Simulink®

Reference

MATLAB® & SIMULINK®

R2018b

MathWorks®

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

Trademarks

Patents

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)
March 2018	Online only	Revised for Simulink 9.1 (Release 2018a)
September 2018	Online only	Revised for Simulink 9.2 (Release 2018b)

Contents

7 Tools and Apps — Alphabetical List

8 Fixed-Point Tool

9 Model Advisor Checks

10 Model Reference Conversion Advisor

11 Performance Advisor Checks

12 Simulink Limits

14 Block Reference Page Examples

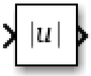
15 Model Parameter Configuration Dialog Box

1

Blocks — Alphabetical List

Abs

Output absolute value of input
Library: Simulink / 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	• 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	• 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.

Ports

Input

`Port_1` — Input signal

scalar | vector

Input signal to the absolute value block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Absolute value output signal

scalar | vector

Absolute value of the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Main

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than -1

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Output minimum` — Minimum output value for range checking

`[ ]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Same as input` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>` | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use

Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Same as input'`, `'Inherit: Inherit via back propagation'`, `'single'`, `'int8'`, `'uint8'`, `int16`, `'uint16'`, `'int32'`, `'uint32'`, `fixdt(1,16,0)`, `fixdt(1,16,2^0,0)`, `fixdt(1,16,2^0,0)`. `'<data type expression>'`
Default: `'Inherit: Same as input'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Choose the behavior when integer overflow occurs

`off` (default) | `on`

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.

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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`Yes`

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™.

For more information about HDL code generation, see Abs.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Sign | Sum

Introduced before R2006a

Action Port

Add control port for action signal to subsystem
Library: Ports & Subsystems

Description

The Action Port block controls the execution of these subsystem blocks:

- If Action Subsystem blocks connected to If blocks.
- Switch Case Action Subsystem blocks connected to Switch Case blocks.
- Simulink based states in Stateflow®charts. See "Create and Edit Simulink Based States" (Stateflow).

Parameters

`States when execution is resumed` — Select handling of internal states

`held` (default) | `reset`

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

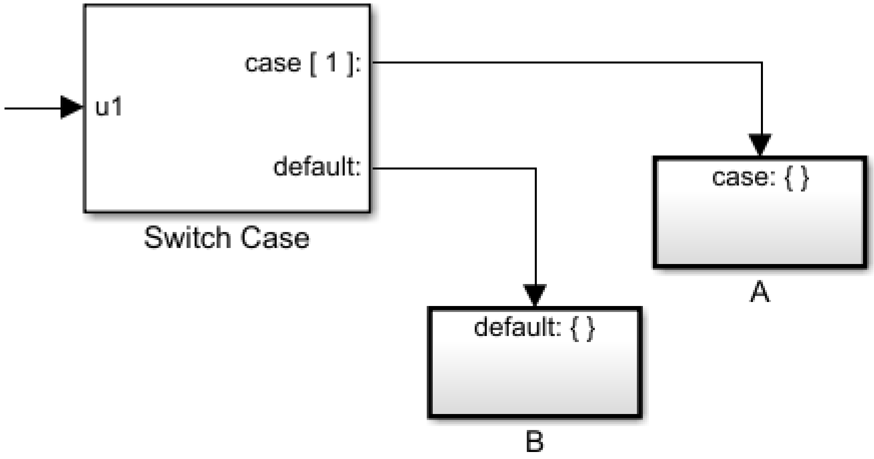
`held`

When the subsystem reenables, retain the previous state values of the subsystem. Previous state values between calls are retained even if you call other subsystem blocks connected to the If or Switch Case block.

`reset`

When the subsystem reenables, reinitialize the state values.

A subsystem reenables when the logical expression for its action port evaluates to true after having been previously false. In the following example, the Action Port blocks for both subsystems A and B have the **States when execution is resumed** parameter set to `reset`.



When **case[1]** is `true`, subsystem **A** is executed. Because the condition of **case[1]** was not previously false, repeated calls to subsystem **A**, while **case [1]** continues to be `true`, does not reset its state values. . The same behavior applies to subsystem **B**.

Programmatic Use
Block Parameter: `InitializeStates`
Type: character vector
Value: `'held'` | `'reset'`
Default: `'held'`

`Propagate sizes of variable-size signals` — Select when to propagate a variable-size signal

`Only when execution is resumed` (default) | `During execution`

Select when to propagate a variable-size signal.

`Only when execution is resumed`

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

`During execution`

Propagate variable-size signals at each time step.

Programmatic Use
Block Parameter: `PropagateVarSize`

Type: character vector
Values: `'Only when execution is resumed'` | `'During execution'`
Default: `'Only when execution is resumed'`

See Also

If | If Action Subsystem | Switch Case | Switch Case Action Subsystem

Topics

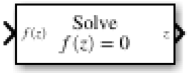
Select Subsystem Execution

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 solve for `f(z) = 0` or `f(z) = z`

Programmatic Use
Block Parameter: `Constraint`
Type: character vector
Values: `'f(z) = 0' | 'f(z) = z'`
Default: `'f(z) = 0'`

`Solver` — Algebraic Loop Solver

`auto` (default) | `Trust region` | `Line search`

Choose 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

Programmatic Use
Block Parameter: `Solver`
Type: character vector
Values: `'auto' | 'Trust region' | 'Line search'`
Default: `'auto'`

`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`.

Programmatic Use
Block Parameter: `Tolerance`
Type: character vector
Values: `'auto' |` positive scalar
Default: `'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

Programmatic Use
Block Parameter: `InitialGuess`
Type: character vector
Values: scalar
Default: `'0'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`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.

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

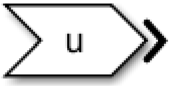
See Also

“Algebraic Loops”

Introduced before R2006a

Argument Inport

Argument input port for Simulink Function block
Library: User-Defined Functions



Description

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

Ports

Input

u — Argument input
scalar | vector | matrix

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 complexity and data type of the block output are the same as the argument 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.

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

Output

out — Block output
scalar | vector | matrix

Block output signal from this block.

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

Parameters

`Port number` — Specify port number of block

`1` (default) | integer

Specify the order in which the port that corresponds to the block appears in the parent subsystem or model block.

`1`

Specify the first port location for this block.

integer

Specify location of port.

Programmatic Use

Block parameter: `Port`
Type: character vector
Value: `'1'` | `'<integer>'`
Default: `'1'`

`Argument name` — Spedify input argument name

`u` (default) | character vector

Specify input argument name for the function prototype displayed on the face of the Simulink Function block.

`u`

Defaut name for the input argument.

character vactor

Name of the input argument.

Programmatic Use

Block parameter: `ArgumentName`
Type: character vector
Value: `'u'` | `'<character vector>'`
Default: `'u'`

`Minimum` — Specify minimum value for block output

`[]` (default) | number

Specify the minimum value for the block output signal.

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 and automatic scaling of fixed-point data types.

`[]`

Minimum value not specified.

number

Finite real double scalar value.

Programmatic Use
Block parameter: `OutMin`
Type: character vector
Value: `'[]'` | `'<number>'`
Default: `'[]'`

`Maximum` — Specify maximum value for block output

`[]` (default) | number

Specify the maximum value for the block output signal.

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 and automatic scaling of fixed-point data types.

[]

Maxmum value not specified.

number

Finite real double scalar value.

Programmatic Use
Block parameter: OutMax
Type: character vector
Value: '[]' | '<number>'
Default: '[]'

Data type — Specify block output data type
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^,0) | Enum: <class name> | Bus: <object name> | <data type expression>

Specify the block output 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.

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'`

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data type

off (default) | on

Control changes to data type settings from the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

☑ 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.

Programmatic Use
Block parameter: `LockScale`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Port dimensions` — Specify port dimensions

`1` (default) | `n` | `[m n]`

Specify the dimensions of the argument input signal to the block. For more information, see Outport.

`1`

Inherit port dimensions.

`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'`

`Signal type` — Select real or complex signal

`real` (default) | `complex`

Select real or complex signal.

`real`

Specify the singnal type as a real number.

`complex`

Specify the signal type as a complex number.

Programmatic Use
Block parameter: `SignalType`
Type: character vector
Value: `'real'` | `'complex'`

Default: `'real'`

See Also

Argument Outport | Simulink Function

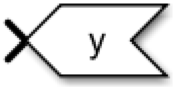
Topics

“Simulink Functions”
“Argument Specification for Simulink Function Blocks”

Introduced in R2014b

Argument Outport

Argument output port for Simulink Function block
Library: User-Defined Functions



Description

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

Ports

Input

in — Block input
scalar | vector | matrix

Block input signal to this block.

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

Output

y — Argument output
scalar | vector | matrix

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

The complexity and data type of the block input are the same as the argument output. You can specify the signal type and data type of an output argument from an Argument

Outport block using the **Signal type** and **Data type** parameters. For more information, see "Data Types Supported by Simulink".

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

Parameters

`Port number` — Specify port number of block

`1` (default) | integer

Specify the order in which the port that corresponds to the block appears in the parent subsystem or model block.

`1`

Specify the first port location for this block.

integer

Specify location of port.

Programmatic Use
Block parameter: `Port`
Type: character vector
Value: `'1'` | `'<integer>'`
Default: `'1'`

`Argument name` — Specify output argument name

`u` (default) | character vector

Specify output argument name for the function prototype displayed on the face of the Simulink Function block.

`u`

Default name of the output argument.

character vector

Name of the output argument.

Programmatic Use
Block parameter: `ArgumentName`
Type: character vector

Value: `'u'` | `'<character vector>'`
Default: `'u'`

`Minimum` — Specify minimum value for block input

`[ ]` (default) | number

Specify the minimum value for the block input signal.

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”) and automatic scaling of fixed-point data types.

`[ ]`

Minimum value not specified.

number

Finite real double scalar value.

Programmatic Use
Block parameter: `OutMin`
Type: character vector
Value: `'[ ]'` | `'<number>'`
Default: `'[ ]'`

`Maximum` — Specify maximum value for block input

`[ ]` (default) | number

Specify the maximum value for the block input signal.

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”) and automatic scaling of fixed-point data types.

`[]`

Max mum value not specified.

number

Finite real double scalar value.

Programmatic Use
Block parameter: `OutMax`
Type: character vector
Value: `'[]'` | `'<number>'`
Default: `'[]'`

Data type — Specify block input data type
`double` (default) | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^,0)` | `Enum: <class name>` | `Bus: <object name>` | `<data type expression>`

Specify block input data type. For more information. For more information, see

- Outport
- “Specify Data Types Using Data Type Assistant”

`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.

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'`

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data type
off (default) | on

Control changes to data type settings from the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

☑ 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.

Programmatic Use
Block parameter: `LockScale`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Port dimensions` — Specify port dimensions

`1` (default) | `n` | `[m n]`

Specify the dimensions of the output argument signal from the block. For more information, see Outport.

`1`

Inherit port dimensions.

`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'`

`Signal type` — Select real or complex signal

`real` (default) | `complex`

Select real or complex signal. For more information, see Outport.

`real`

Specify the signal type as a real number.

`complex`

Specify the signal type as a complex number.

Programmatic Use
Block parameter: `SignalType`
Type: character vector
Value: `'real'` | `'complex'`
Default: `'real'`

See Also

Argument Inport | Function Caller | Simulink Function

Topics

“Simulink Functions”
“Argument Specification for Simulink Function Blocks”

Introduced in R2014b

ASCII to String

Uint8 vector signal to string signal
Library: Simulink / String

Description

The ASCII to String block converts uint8 vector signals to string signals. The block treats each element in the input vector as an ASCII value during the conversion. For example, the block converts an input vector of `[72 101 108 108 111]` to the string `"Hello"`.

Ports

Input

`Port_1` — ASCII signal
vector

ASCII signal, specified as a vector.

While using dynamic strings, if the length of the input vector exceeds the number of characters specified in the configuration parameter **Buffer size of dynamically-sized string (bytes)** (`256` by default), the ASCII to String block truncates the string output to the buffer size-1 (for example, `255`), for generated code. To avoid truncation, increase the value of the **Buffer size of dynamically-sized string (bytes)** configuration parameter.

Example: [088 099]

Data Types: `uint8`

Output

`Port_1` — Converted string signal
scalar

Converted string signal from input ASCII signal, specified as a scalar. The block converts each ASCII element in the vector into its alphanumeric equivalent and outputs all elements concatenated into one string.

Data Types: `string`

Block Characteristics

Data Types	`base integer` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

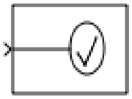
Topics

"String Data Type Conversions"
"Simulink Strings"

Introduced in R2018a

Assertion

Check whether signal is zero
Library: Simulink / Model Verification



Description

The Assertion block checks whether any of the elements of the input signal are 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 displays a warning message when the assertion fails but allows the simulation to continue.
- Specify a MATLAB expression to evaluate when the assertion fails.
- Enable or disable the assertion.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal to the assertion check.

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

Parameters

`Enable assertion` — Enable or disable the check

`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Assertion.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

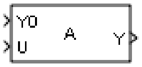
Check Dynamic Lower Bound | Check Dynamic Upper Bound | Check Static Lower Bound | Check Static Upper Bound

Introduced before R2006a

Assignment

Assign values to specified elements of signal

Library: Simulink / Math Operations



Description

The Assignment block assigns values to specified elements of the signal. You 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, `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. To enable an external index port, in the corresponding row of the table, set **Index Option** to `Index vector (port)`.

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:

Row	Index Option	Index
1	`Assign all`	
2	`Index vector (dialog)`	`[1 3 5]`
3	`Starting index (dialog)`	`4`
4	`Starting index (port)`	
5	`Index vector (port)`	

The assigned values are `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.

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.

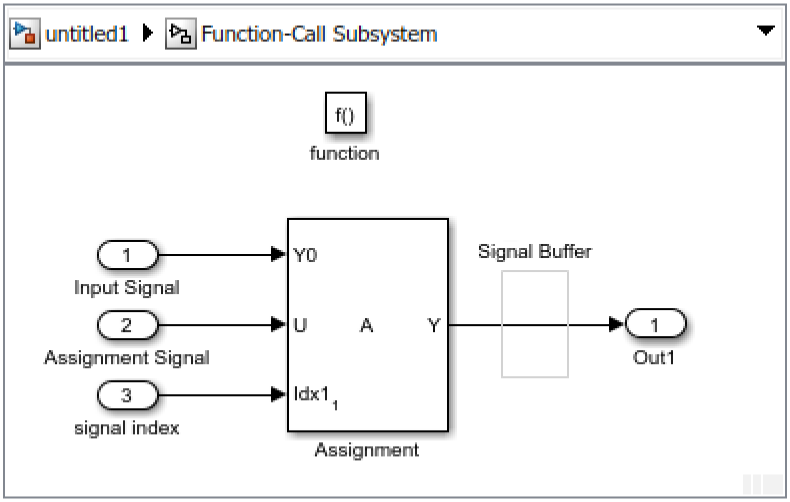
You can use the block to assign values to vector, matrix, or multidimensional signals.

You can use an array of buses as an input signal to an Assignment block.

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 outport is virtual** check box for the conditional subsystem Outport block, such cases are supported and partial writes to arrays using Assignment blocks are possible.



Ports

Input

`Y0` — Input initialization signal

scalar | vector

The initialization signal for the output signal. If an element is not assigned another value, then the value of the output element matches this input signal value.

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

`U` — Input data port

scalar | vector

Value assigned to the output element when specified.

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

`IndxN` — `N`th index signal

scalar | vector

External port specifying an index for the assignment of the corresponding output element.

Dependencies

To enable an external index port, in the corresponding row of the **Index Option** table, set **Index Option** to `Index vector (port)` or `Starting index (port)`.

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

Output

`Y` — Output signal with assigned values

scalar | vector

The output signal with assigned values for the specified elements.

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

Parameters

Number of output dimensions — Number of dimensions of the output signal

1 (default) | integer

Enter the number of dimensions of the output signal.

Command-Line Information
Parameter: NumberOfDimensions
Type: character vector
Values: integer
Default: '1'

Index mode — Index mode

One-based (default) | Zero-based

Select the indexing mode. If One-based is selected, an index of 1 specifies the first element of the input vector. If Zero-based is selected, an index of 0 specifies the first element of the input vector.

Command-Line Information
Parameter: IndexMode
Type: character vector
Values: 'Zero-based' | 'One-based'
Default: 'One-based'

Index Option — Index method for elements

Index vector (dialog) (default) | Assign all | Index vector (port) | Starting index (dialog) | Starting index (port)

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

Menu Item	Action
Assign all	This is the default. All elements are assigned.

Menu Item	Action
`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 one of these values for the **Initialize output (Y)** parameter:

- `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.

Command-Line Information
Parameter: `IndexOptionArray`
Type: character vector
Values: `'Assign all'` | `'Index vector (dialog)'` | `'Index option (port)'` | `'Starting index (dialog)'` | `'Starting index (port)'`
Default: `'Index vector (dialog)'`

Index — Index of elements

1 (default) | integer

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`.

Command-Line Information
Parameter: `IndexParamArray`
Type: character vector
Values: cell array
Default: `'{ }'`

`Output Size` — Width of the block output signal

1 (default) | integer

Enter the width of the block output signal.

Dependencies

To enable this column, select `Specify size for each dimension in table` for the **Initialize output (Y)** parameter.

Command-Line Information
Parameter: `OutputSizeArray`
Type: character vector
Values: cell array
Default: `'{ }'`

`Initialize output (Y)` — How to initialize the output signal

`Initialize using input port <Y0>` (default) | `Specify size for each dimension in the table`

Specify how to initialize the output signal.

- `Initialize using input port <Y0>` – Signal at the input port `Y0` initializes the output.
- `Specify size for each dimension in table` – 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.

Dependency

Enabled when you set **Index Option** to `Index vector (port)` or `Starting index (port)`.

Command-Line Information
Parameter: `OuputInitialize`
Type: character vector

Values: `'Initialize using input port <Y0>'` | `'Specify size for each dimension in table'`
Default: `'Initialize using input port <Y0>'`

`Action if any output element is not assigned` — Specify whether to produce a warning or error if you have not assigned all output elements

`Error` (default) | `Warning` | `None`

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.

Command-Line Information
Parameter: `DiagnosticForDimensions`
Type: character vector
Values: `'Error'` | `'Warning'` | `'None'`
Default: `'None'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
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™.

For more information on HDL code generation, see Assignment.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

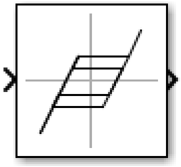
“Combine Buses into an Array of Buses” | Bus Assignment

Introduced before R2006a

Backlash

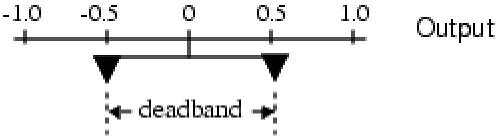
Model behavior of system with play

Library: Simulink / Discontinuities



Description

The Backlash block implements a system in which a change in input causes an equal change in output, except when the input changes direction. When the input changes direction, the 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 an 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.

Mode	Input	Output
Disengaged	Inside deadband zone.	Remains constant.
Engaged-positive direction	Outside deadband zone and increasing.	Equals input *minus* half of deadband width.
Engaged-negative direction	Outside deadband zone and decreasing.	Equals input *plus* half of deadband width.

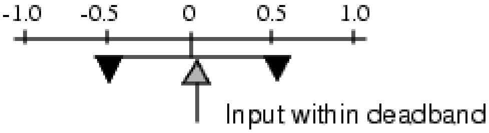
The **Initial output** parameter value defines the initial center of the deadband zone.

This table shows output values when initial conditions are: **Deadband width** = 2 and **Initial output** = 5.

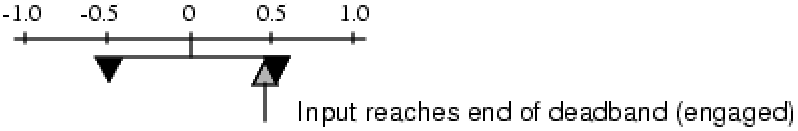
Output Value	Condition
5	4 < input < 6
input + 1	input < 4
input - 1	input > 6

For example, you can use the Backlash block to model the meshing of two gears. The input and output are both shafts with a gear on one end, and the input shaft drives the output 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 **Initial output** parameter defines the output.

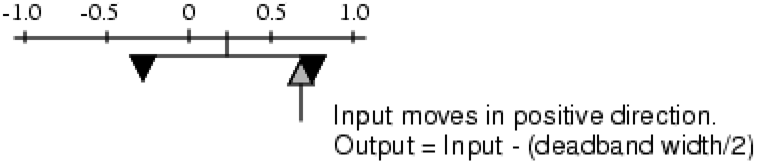
These figures illustrate operation when the initial input is within the deadband and the system begins in disengaged mode.



When the input increases and reaches the end of the deadband, it engages the output. The output remains at its previous value.



After the input engages the output, the output changes by the same amount as the input.


-1.0
-0.5
0
0.5
1.0
Input moves in positive direction.
Output = Input - (deadband width/2)

If the input reverses direction, it disengages from the output. The output remains constant until the input reaches the end of the deadband and engages again.

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal to the backlash algorithm. The value of this signal is either in the deadband or engaging the output in a positive or negative direction.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32

Output

Port_1 — Output signal
scalar | vector

Output signal after the backlash algorithm is applied to the input signal. When the input is in the deadband, then the output remains unchanged. If the input is engaged with the output, then the output changes an equal amount as the input.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32

Parameters

Deadband width — Specify the width of the deadband
1 (default) | scalar | vector

Specify the size of the deadband zone centered on the output value. When the input signal is inside the deadband, then a change in input does not cause a change in output. When the input signal is outside of the deadband, then the output changes an equal amount as the input.

Programmatic Use
Block Parameter: `BacklashWidth`
Type: character vector
Values: real scalar or vector
Default: `'1'`

`Initial output` — Specify the initial output value

0 (default) | scalar | vector

Specify the initial center of the deadband zone. If the initial input value is in the deadband zone, then the output value is equal to **Initial output**. If the initial input value is outside of the deadband zone, then the output value is **Initial output** plus or minus half of the deadzone width.

Programmatic Use
Block Parameter: `InitialOutput`
Type: character vector
Values: real scalar or vector
Default: `'0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `base integer`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`

Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` function (`string.h`) in certain conditions.

HDL Code Generation

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

For more information on HDL code generation, see Backlash.

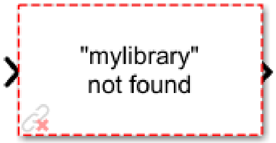
See Also

Dead Zone

Introduced before R2006a

Unresolved Link

Indicate unresolved reference to library block



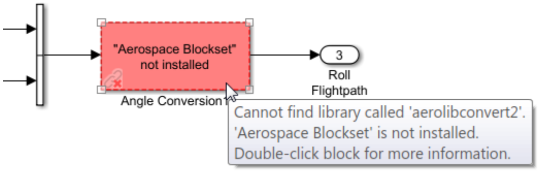
Description

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.

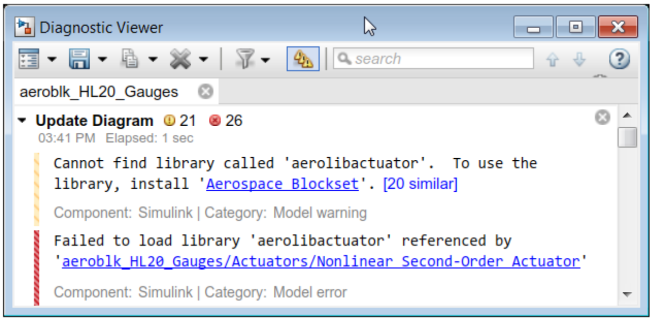
Parameters

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.



Cannot find library called 'aerolibconvert2'.
'Aerospace Blockset' is not installed.
Double-click block for more information.



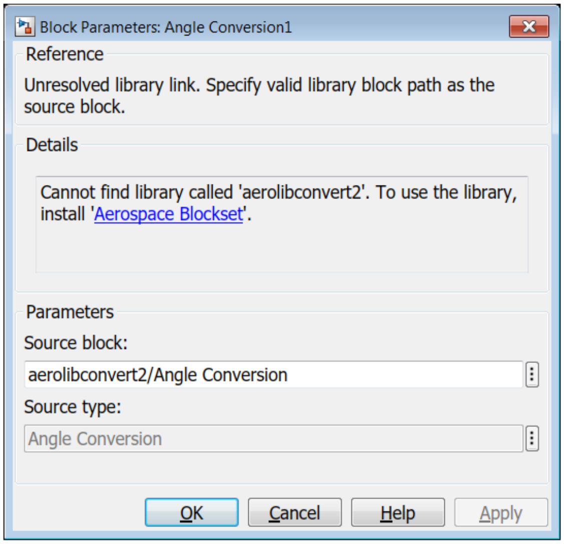
Update Diagram 21 26
03:41 PM Elapsed: 1 sec

```
Cannot find library called 'aerolibactuator'.  To use the
library, install 'Aerospace Blockset'. [20 similar]
```

Component: Simulink | Category: Model warning

```
Failed to load library 'aerolibactuator' referenced by
'aeroblk_HL20_Gauges/Actuators/Nonlinear Second-Order Actuator'
```

Component: Simulink | Category: Model error



Reference

Unresolved library link. Specify valid library block path as the source block.

Details

Cannot find library called 'aerolibconvert2'. To use the library, install 'Aerospace Blockset'.

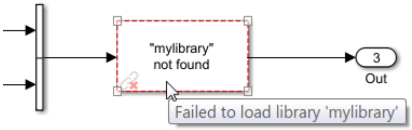
Parameters

Source block:

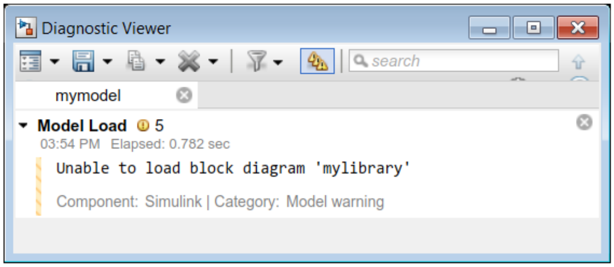
aerolibconvert2/Angle Conversion

Source type:

Angle Conversion



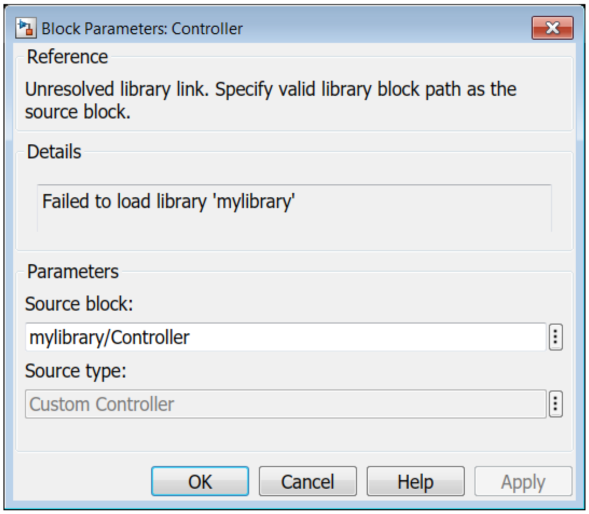
Failed to load library 'mylibrary'



Model Load 5
03:54 PM Elapsed: 0.782 sec

```
Unable to load block diagram 'mylibrary'
```

Component: Simulink | Category: Model warning



Reference

Unresolved library link. Specify valid library block path as the source block.

Details

Failed to load library 'mylibrary'

Parameters

Source block:

mylibrary/Controller

Source type:

Custom Controller

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

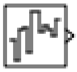
Topics

"Linked Blocks"
"Fix Unresolved Library Links"

Introduced in R2014a

Band-Limited White Noise

Introduce white noise into continuous system
Library: Simulink / Sources



Description

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

Simulation of White Noise

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: mag^2/(rad/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)(2π) = 1/π, 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.

Ports

Output

`Port_1` — Normally distributed random numbers
scalar | vector | matrix | N-D array

Normally distributed random numbers specified as a scalar, vector, matrix, or N-D array.

Data Types: `double`

Parameters

`Noise power` — Height of PSD of white noise
`[0.1]` (default) | scalar | vector | matrix | N-D array

Specify the height of the PSD of the white noise as a scalar, vector, matrix, or N-D array of positive values.

Programmatic Use
Block Parameter: `Cov`
Type: character vector
Values: `scalar | vector | matrix | N-D array`
Default: `'[0.1]'`

`Sample time` — Correlation time of noise

`0.1` (default) | scalar | vector

Correlation time of the noise. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `Ts`
Type: character vector
Values: `scalar | vector`
Default: `'0.1'`

Seed — Starting seed

`[23341]` (default) | scalar | vector | matrix | N-D array

Specify the starting seed for the random number generator as a scalar, vector, matrix, or N-D array. Values must be positive, real-valued, and finite.

Programmatic Use
Block Parameter: `seed`
Type: character vector
Values: `scalar | vector | matrix | N-D array`
Default: `'[23341]'`

`Interpret vector parameters as 1-D` — Treat vector parameters as 1-D

`on` (default) | `off`

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`. For more information, see “Determining the Output Dimensions of Source Blocks”.

Programmatic Use
Block Parameter: `VectorParams1D`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Algorithms

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*.

Extended Capabilities

C/C++ Code Generation

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

Cannot use inside a triggered subsystem hierarchy.

See Also

Random Number

Topics

“Sample Time”

Introduced before R2006a

Bias

Add bias to input

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector

Input signal to which the bias is added to create the output signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector

Output signal resulting from adding the bias to the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Bias — Offset to add to the input signal

0.0 (default) | scalar | vector

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

Programmatic Use
Block Parameter: Bias
Type: character vector
Values: real, finite
Default: '0.0'

Saturate on integer overflow — Choose the behavior when integer overflow occurs

on (default) | boolean

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 int8 (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.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`

Multidimensional Signals	`No`
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™.

For information about HDL Code Generation, see Bias.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

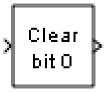
See Also

Add | Divide | Gain

Introduced before R2006a

Bit Clear

Set specified bit of stored integer to zero
Library: Simulink / 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.

Ports

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.

Input

`Port_1` — Input signal
scalar or vector

The input signal is the specified bit of the stored integer.

Data Types: `single` | `double` | `Boolean` | `fixed point`

Output

`Port_1` — Output signal
scalar or vector

The output consists of the specified bit set to zero.

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

Parameters

`Index of bit` — Index of bit

0 (default) | scalar or vector

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

Programmatic Use
Block Parameter: `iBit`
Type: scalar or vector
Values: {'0'}
Default: '0'

Block Characteristics

Data Types	`Boolean`[a] | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[a] Bit operations are not recommended for use with Boolean signals.

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™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

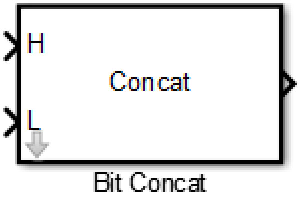
See Also

Bit Rotate | Bit Set | Bit Shift | Bitwise Operator

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

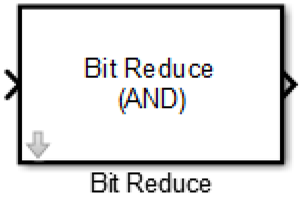
See Also

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`

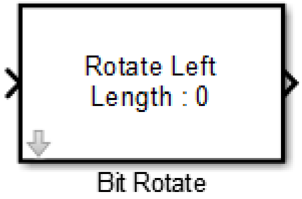
See Also

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

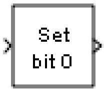
Bit Shift | Bit Concat | Bit Reduce | Bit Slice

Introduced in R2014a

Bit Set

Set specified bit of stored integer to one

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar or vector

Input signal with the specified bit of the stored integer.

Data Types: `single` | `double` | `Boolean` | `fixed point`

Output

`Port_1` — Output signal
scalar or vector

Output signal with the specified bit set to 1.

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

Parameters

`Index of bit` — Index of bit

0 (default) | scalar or vector

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

Programmatic Use
Block Parameter: `iBit`
Type: character vector
Values: positive integer
Default:`'0'`

Block Characteristics

Data Types	`Boolean`[a] | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[a] Bit operations are not recommended for use with Boolean signals.

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™.

For more information about HDL code generation, see Bit Set.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

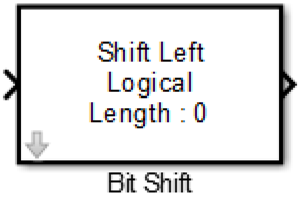
See Also

Bit Clear | Bit Rotate | Bit Shift | Bitwise Operator

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

Slice
(7 downto 0)
Bit Slice

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 `(MSB Position - 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: Simulink / Logic and Bit Operations

Description

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.

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.

Ports

Input

`Port_1` — Input signal

scalar | vector

Input signal, specified as a scalar or vector.

- 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.
- 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.

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

Output

`Port_1` — Output signal

scalar | vector

The output signal specified as 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. 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.

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

Parameters

`Operator` — Bitwise logical operator

AND (default) | OR | NOR | NAND | XOR | NOT

Specify the bitwise logical operator for the block operands.

You can select one of these 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

Bitwise Operation	Description
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)

Programmatic Use

Block Parameter: `logicop`
Type: character vector
Values: `'AND'|'OR' |'NAND'|'NOR' |'XOR' |'NOT'`
Default: `'AND'`

`Use bit mask` — Select to use bit mask

checked (default) | unchecked

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

Programmatic Use

Block Parameter: `UseBitMask`
Type: character vector
Values: `'off'|'on'`
Default: `'on'`

`Number of input ports` — Number of input signals

1 (default) | integer

Specify the number of inputs. You can have more than one input ports.

Dependency

Clearing the **Use bit mask** check box enables **Number of input ports** and disables **Bit Mask** and **Treat mask as**.

Programmatic Use

Block Parameter: `NumInputPorts`
Type: character vector
Values: positive integer
Default: `'1'`

`Bit Mask` — Bit mask to associate with a single input

bin2dec (default)

Specify the bit mask to associate with a single input.

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.

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

Dependency

This parameter is available only when you select **Use bit mask**.

Programmatic Use
Block Parameter: `BitMask`
Type: character vector
Values: positive integer
Default: `'bin2dec('11011001')'`

`Treat mask as` — Treat the mask as a real-world value or a stored integer

`Stored Integer` (default) | `Real World Value`

Specify whether to treat the mask as a real-world value or a stored integer.

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.

Dependency

This parameter is available only when you select **Use bit mask**.

Programmatic Use
Block Parameter: `BitMaskRealWorld`
Type: character vector
Values: `'Real World Value'` | `'Stored Integer'`
Default: `'Stored Integer'`

Block Characteristics

Data Types	`Boolean`[a] | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[a] Bit operations are not recommended for use with Boolean signals.

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™.

For more information about HDL code generation, see Bitwise Operator.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Compare To Constant | Compare To Zero | Logical Operator | Shift Arithmetic

Introduced before R2006a

Block Support Table

View data type support for Simulink blocks
Library: Simulink / Model-Wide Utilities

Description

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

Block Characteristics

Data Types	
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Tips

There are two alternate ways to open the Block Support Table:

- To access the Simulink Block Support Table from the model menu, select **Help > Simulink > Block Data Types & Code Generation Support > Simulink**.

- To open the Block Support Table from the command line, you can enter `showblockdatatypetable` at the MATLAB command prompt.

Extended Capabilities

C/C++ Code Generation

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

The Block Support Table block is ignored during code generation.

See Also

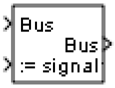
Topics

“Data Types Supported by Simulink”

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. For a nested bus that is nested inside an array of buses, see “Assign into Array of Buses Signals”.
- 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

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`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™.

This block has a single, default HDL architecture. See Bus Assignment.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

See Also

Bus Creator | Bus Selector

Topics

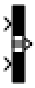
"Getting Started with Buses"
"Combine Buses into an Array of Buses"
"Composite Signals"
"Buses and Libraries"

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 that 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` — Search term used to filter input signals

text

Search term used 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 to the right of the **Filter by name** edit box.

`Enable regular expression` — Option to filter with regular expressions

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 to the right of the **Filter by name** edit box.

`Show filtered results as a flat list` — Option to display filtered results as a flat list

off (default) | on

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

Dependencies

To access this parameter, click the ⏬ button to the right 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` — Option to require input signal names to match names listed in dialog box

off (default) | on

Optional check that input signal names match signal names in the Bus Creator Block Parameters dialog box.

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**, Simulink software ignores the **Require input signal names to match signals above** setting.

If you change the **Number of inputs** programmatically, this parameter reverts to `'off'`.

`Rename selected signal` — Name for currently selected signal

`''` (default) | character vector

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

Dependencies

To display this parameter, enable the **Require input signal names to match signals above** parameter.

`Override bus signal names from inputs` — Bus input signal name override

on (default) | off

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 by checking that input signal names match the bus object element names.

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

Dependencies

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

Programmatic Use
Block Parameter: `InheritFromInputs`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output as nonvirtual bus` — Nonvirtual bus output

off (default) | on

Nonvirtual bus output 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 change the sample time of a signal or bus that has a different sample time than the other nonvirtual bus input signals, use a Rate Transition block. For details, see "Specify Bus Signal Sample Times".

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

Dependencies

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

Programmatic Use
Block Parameter: `NonVirtualBus`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	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™.

This block has a single, default HDL architecture. See Bus Creator.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point 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

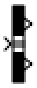
“Getting Started with Buses”
“Simplify Subsystem Bus Interfaces”
“Nest Buses”
“Bus-Capable Blocks”
“Assign Signal Values to a Bus”
“Composite Signals”
“Specify Bus Signal Sample Times”

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 virtual 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` — Input bus signal

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, use a Selector block to select 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` — Selected bus elements of input bus signal

standalone signal (default) | virtual bus | nonvirtual bus | array of buses

Selected bus element signals of an input bus signal. For each output signal, this block uses 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 virtual 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 expressionto 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 virtual 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 virtual 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'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`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™.

This block has a single, default HDL architecture. See Bus Selector.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point 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

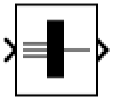
Topics

“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 signals must consist of scalars or 1-D, row, or column vectors that have the same data type, signal type, and sampling mode. If the input bus contains row or column vectors, the output is a row or column vector, respectively. Otherwise, the output is a 1-D array.

Use the Bus to Vector block only to replace an implicit bus-to-vector conversion with an 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 9-29** 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** to save 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 path 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

The input bus signals must consist of scalars or 1-D, row, or column vectors that have 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 output is a row or column vector, respectively. Otherwise, the output is a 1-D array.

Parameters

This block has no user-accessible parameters.

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`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™.

This block has a single, default HDL architecture. See Bus to Vector.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point 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

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.

See Also

Push Button

Topics

"Tune and Visualize Your Model with Dashboard Blocks"
"Decide How to Visualize Simulation Data"

Introduced in R2017b

C Caller

Integrate C code in Simulink
Library: Simulink / User-Defined Functions

Description

The C Caller block integrates your external C code into Simulink. This block imports and lists the functions in your external C code, and allows you to select your resolved C functions to integrate in your Simulink models. The C Caller block standalone supports code generation. For more complex models, code generation depends on the capabilities of your Simulink model.

To use C Caller block, define your source code and any supporting files using **Simulation Target** under **Configuration Parameters**. Then, bring a C Caller block to Simulink canvas, using **Library Browser > Simulink > User Defined Functions**. To change the defined source code file and its dependencies, go to **Simulation Target** tab in **Configuration Parameters** by clicking the ⚙ from the block dialog. After changing your source code or any of its dependencies, refresh the list of functions by clicking the ↻ on the block dialog. To browse and modify the list of functions in your source code, use the » icon to easily access to your header files.

Ports

Input

`Input argument` — Input argument for the corresponding C caller block
scalar | vector | matrix

Input argument to the C Caller block.

Number of input arguments and their names are inferred through the selected function in your externaI C code. To receive data to your block, connect an input signal to the input ports.

Input label has the same name as your input argument unless changed by editing the **Label** column under **Port Specification** from the **Block Dialog**. If you rename the label to an input port, the C Caller block changes the name of the port.

For input variables, you can change the input scope to parameters or constants using the **Scope** column.

Data types supported in MATLAB, but not supported in Simulink cannot be passed between C Caller and other MATLAB blocks.

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

Output

`Output argument` — Output argument for the corresponding C Caller block

scalar | vector | matrix

Output argument from the C Caller block.

Number of output arguments and their names are inferred through the selected function in your externaI C code. To send data from your C Caller block, connect a block to the output port of your C Caller block.

Output port label has the same name as your output argument unless you change it by editing the **Label** column under **Port Specification** from the **Block Dialog**. If you rename the label to an input port, the C Caller block changes the name of the port.

Data types supported in MATLAB, but not supported in Simulink cannot be passed between C Caller and other MATLAB blocks.

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

Parameters

`Function name` — Name of the functions in source code

character vector

The C Caller block in your model imports all functions in your external source code, and shows the names of functions under `Function name` in block dialog. To select and use a function in your block, confirm that their names appear at the `Function name`. In case you are missing one of the functions, reload the source code by clicking ↻ on the block dialog box. To change the name of your functions, modify your source code and click the refresh button to reload your functions.

`Port specification` — Specify port properties

string

`Port specification` table indicates the attributes of each input and output element for the selected block. These properties include Argument name, scope, input/output label, type and size of the input/output variables. Argument name, scope, type, and size are inferred from your source code. If the type of scope is an input, you can modify this variable to a parameter or a constant.

Arg name — Demonstrates the variable name inferred from your source code.

Scope — Indicates the role of the variables from your source code. If the variable is an input argument in the C Caller block source code, you can change the scope type to a constant or a parameter. If the variable is an output argument in the source code, you cannot change the scope type.

Label — Labels the input or output variable for the Simulink model. You can change the labels using this table. If the scope is a parameter, enter the parameter name in this field. If the scope is a constant, enter the constant value.

Type — Indicates the data type coming from the ports.

Size — Indicates the size of the input and output data.

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[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.

See Also

MATLAB Function | MATLAB System | S-Function | S-Function Builder

Topics

"Integrate C Code Using C Caller Blocks"

Introduced in R2018b

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.

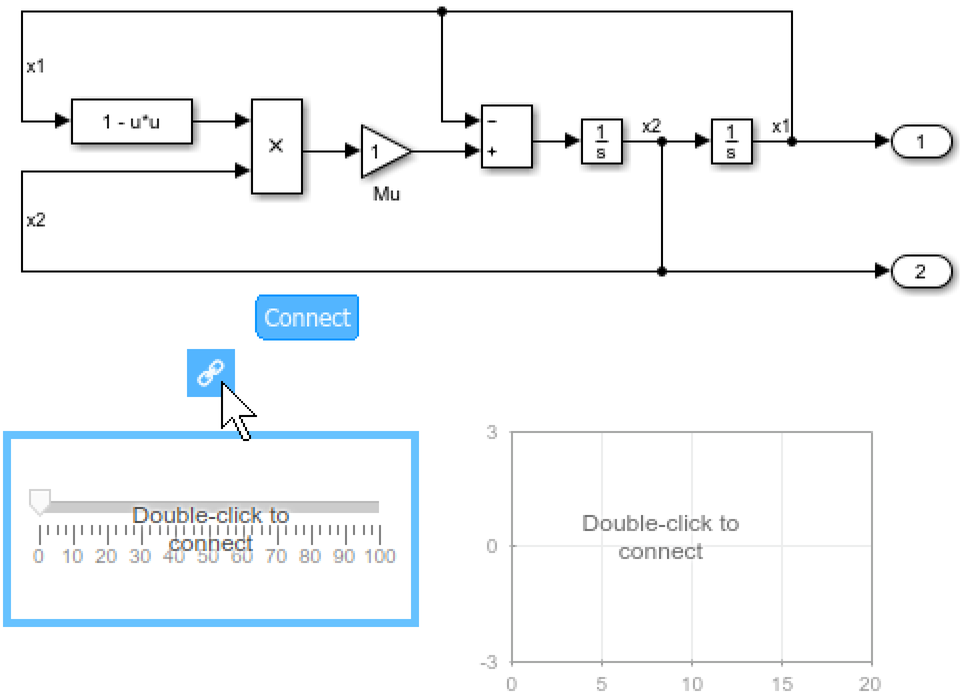
Double-clicking the Check Box block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

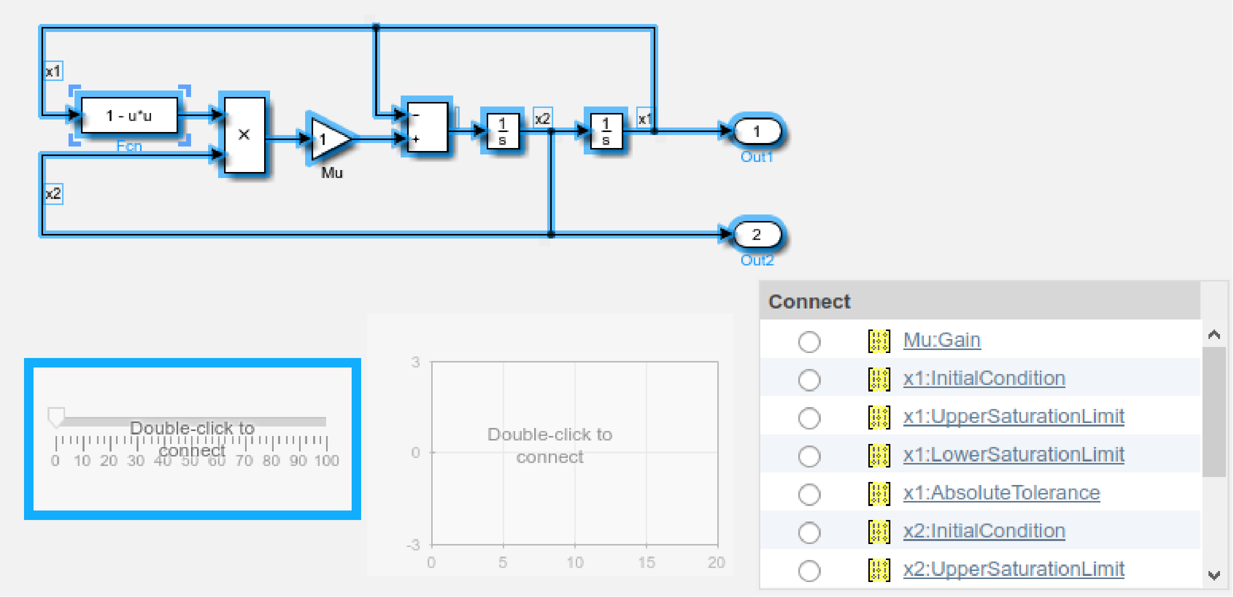
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

- ○ Mu:Gain
- ○ x1:InitialCondition
- ○ x1:UpperSaturationLimit
- ○ x1:LowerSaturationLimit
- ○ x1:AbsoluteTolerance
- ○ x2:InitialCondition
- ○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

See Also

Rocker Switch | Rotary Switch | Slider Switch | Toggle Switch

Topics

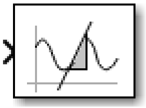
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2017b

Check Discrete Gradient

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

Library: Simulink / 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. Specify the value of the upper bound (1 by default) by setting the **Maximum gradient** parameter. 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.

Enable or disable all model verification blocks by changing the **Model Verification block enabling** setting in the Configuration Parameters.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal the block checks to determine if the difference of each element between sucessive samples is less than the upper bound. Specify the upper bound by setting the **Maximum gradient** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Maximum gradient` — Upper bound of allowed differences

`1` (default) | scalar

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

Command-Line Information
Parameter: `gradient`
Type: character vector
Values: real scalar
Default: `'1'`

`Enable assertion` — Enable or disable the check

`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the

Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Select icon type` — Select icon type

`graphic` | `text`

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: `icon`
Type: character vector
Values: `'graphic'` | `'text'`
Default: `'graphic'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Check Discrete Gradient.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

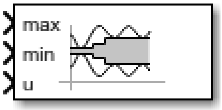
See Also

Check Dynamic Gap | Check Dynamic Range

Introduced before R2006a

Check Dynamic Gap

Check that gap of possibly varying width occurs in range of signal's amplitudes
Library: Simulink / 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 connected to **u**. The inputs **min** and **max** specify the lower and upper bounds of the dynamic gap. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`max` — Upper bound of dynamic gap
scalar | vector | matrix

Signal specifying the upper bound of the gap. All three input signals must be the same data type.

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

`min` — Lower bound of dynamic gap

scalar | vector | matrix

Signal specifying the lower bound of the gap. All three input signals must be the same data type and dimension.

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

`u` — Input signal

scalar | vector | matrix

Input signal checked for a gap of width specified by **max** and **min**. All three input signals must be the same data type and dimension.

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

Output

`Port_1` — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Enable assertion` — Enable or disable the check

on (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the

Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Select icon type` — Select icon type

`graphic` | `text`

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: `icon`
Type: character vector
Values: `'graphic'` | `'text'`
Default: `'graphic'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Check Dynamic Gap.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

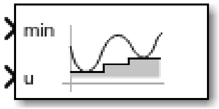
See Also

Check Dynamic Lower Bound | Check Dynamic Range | Check Dynamic Upper Bound

Introduced before R2006a

Check Dynamic Lower Bound

Check that one signal is always less than another signal
Library: Simulink / Model Verification



Description

The Check Dynamic Lower Bound block checks that the amplitude of a reference signal, **min**, is less than the amplitude of a test signal, **u**, 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.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`min` — Lower bound of check
scalar | vector | matrix

Signal specifying the lower bound of the check against the input signal **u** amplitude. Signal data type and dimension must be the same as **u**.

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

u — Input signal
scalar | vector | matrix

Input signal checked against the lower bound specified by **min**. Both input signals must be the same data type and dimension.

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

Output

Port_1 — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: double | Boolean

Parameters

Enable assertion — Enable or disable the check

on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: enabled
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails

' ' (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Select icon type` — Select icon type

`graphic` | `text`

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: `icon`
Type: character vector
Values: `'graphic'` | `'text'`
Default: `'graphic'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

For information about HDL Code Generation, see Check Dynamic Lower Bound.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

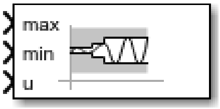
See Also

Introduced before R2006a

Check Dynamic Range

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

Library: Simulink / 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 **u** 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.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`max` — Upper bound of dynamic range check
scalar | vector | matrix

Signal specifying the upper bound of the range that the block checks the input signal **u** amplitude. All three input signals must be the same data type.

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

`min` — Lower bound of dynamic range check

scalar | vector | matrix

Signal specifying the lower bound of the range that the block checks the input signal **u** amplitude. All three input signals must be the same data type and dimension.

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

u — Input signal

scalar | vector | matrix

Input signal checked against the range specified by **max** and **min**. All three input signals must be the same data type and dimension.

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

Output

`Port_1` — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Enable assertion` — Enable or disable the check

`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the

Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Select icon type` — Select icon type

`graphic` | `text`

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: `icon`
Type: character vector
Values: `'graphic'` | `'text'`
Default: `'graphic'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

For information about HDL Code Generation, see Check Dynamic Range.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

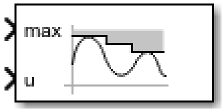
See Also

Introduced before R2006a

Check Dynamic Upper Bound

Check that one signal is always greater than another signal

Library: Simulink / Model Verification



Description

The Check Dynamic Upper Bound block checks that the amplitude of a reference signal, **max**, is greater than the amplitude of a test signal, **u**, 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.

Use the blocks in the Model Verification library 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 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).

Ports

Input

max — Upper bound of range check

scalar | vector | matrix

Signal specifying the lower bound of the range that the block checks the input signal **u** amplitude. Signal data type and dimension must be the same as **u**.

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

u — Input signal
scalar | vector | matrix

Input signal checked against the lower bound specified by **min**. Both input signals must be the same data type and dimension.

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

Output

`Port_1` — Assertion output signal
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Enable assertion` — Enable or disable the check
`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector

Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`

Default: 'off'

Select icon type — Select icon type
graphic | text

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: icon
Type: character vector
Values: 'graphic' | 'text'
Default: 'graphic'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

For information about HDL Code Generation, see Check Dynamic Upper Bound.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

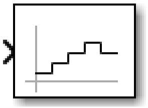
See Also

Check Dynamic Lower Bound | Check Dynamic Range

Introduced before R2006a

Check Input Resolution

Check that input signal has specified resolution
Library: Simulink / Model Verification



Description

The Check Input Resolution block checks whether the input signal has a specified scalar or vector resolution. 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.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal that the block checks the resolution specified by the **Resolution** parameter.

Data Types: `double`

Output

Port_1 — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

Resolution — Resolution

scalar

Specify the resolution requirement for the input signal.

Command-Line Information

Parameter: `resolution`
Type: character vector
Values: `'1'` | `real value`
Default: `'1'`

Enable assertion — Enable or disable the check

`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information

Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`

Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Block Characteristics

Data Types	double
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Check Input Resolution.

PLC Code Generation

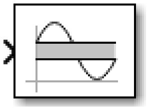
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Introduced before R2006a

Check Static Gap

Check that gap exists in signal's range of amplitudes
Library: Simulink / 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.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`Port_1` — Input
scalar | vector | matrix

Input signal the block checks if the signal value is less than a static lower bound or greater than a static upper bound.

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

Output

`Port_1` — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, set the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Upper bound` — Upper boundary value

scalar | vector | matrix

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

Command-Line Information
Parameter: `max`
Type: character vector
Values: `scalar` | `vector` | `matrix`
Default: `'0'`

`Inclusive upper bound` — Include the upper bound in range

`on` (default) | `off`

Select this check box to make the range of valid input amplitudes include the upper bound.

Command-Line Information
Parameter: `max_included`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Lower bound` — Lower boundary value

scalar | vector | matrix

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

Command-Line Information
Parameter: `min`
Type: character vector
Values: `scalar` | `vector` | `matrix`
Default: `'0'`

`Inclusive lower bound` — Include the lower bound in range

`on` (default) | `off`

Select this check box to make the range of valid input amplitudes include the lower bound.

Command-Line Information
Parameter: `min_included`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Enable assertion` — Enable or disable the check

`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Select icon type` — Select icon type

`graphic` | `text`

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: `icon`
Type: character vector
Values: `'graphic'` | `'text'`
Default: `'graphic'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Check Static Gap.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

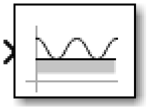
Check Dynamic Range | Check Static Lower Bound | Check Static Upper Bound

Introduced before R2006a

Check Static Lower Bound

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

Library: Simulink / 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. Specify the value of the lower bound with the **Lower bound** parameter. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal checked against the lower bound specified by the **Lower bound** parameter.

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

Output

`Port_1` — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Lower bound` — Lower boundary value

scalar | vector | matrix

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

Command-Line Information
Parameter: `min`
Type: character vector
Values: `scalar` | `vector` | `matrix`
Default: `'0'`

`Inclusive boundary` — Include the lower bound in range

`on` (default) | `off`

Select this check box to make the range of valid input amplitudes include the lower bound.

Command-Line Information
Parameter: `min_included`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Enable assertion` — Enable or disable the check

on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

off (default) | on

Selecting this check box causes the 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.

Command-Line Information
Parameter: export
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Select icon type — Select icon type

graphic | text

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: icon
Type: character vector
Values: 'graphic' | 'text'
Default: 'graphic'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No

Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Check Static Lower Bound.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

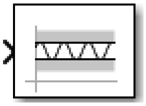
Check Dynamic Lower Bound | Check Dynamic Range | Check Dynamic Upper Bound | Check Static Upper Bound

Introduced before R2006a

Check Static Range

Check that signal falls inside fixed range of amplitudes

Library: Simulink / 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. Specify the upper and lower bounds with the parameters **Upper bound** and **Lower bound**. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal checked against the range specified by the **Upper bound** and **Lower bound** parameters.

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

Output

`Port_1` — Assertion output signal
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, set the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Upper bound` — Upper boundary value
scalar | vector | matrix

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

Command-Line Information
Parameter: `max`
Type: character vector
Values: `scalar` | `vector` | `matrix`
Default: `'0'`

`Inclusive upper boundary` — Include the upper bound in range
on (default) | `off`

Select this check box to make the range of valid input amplitudes include the lower bound.

Command-Line Information
Parameter: `min_included`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Lower bound` — Lower boundary value

scalar | vector | matrix

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

Command-Line Information
Parameter: `min`
Type: character vector
Values: `scalar` | `vector` | `matrix`
Default: `'0'`

`Inclusive lower bound` — Include the lower bound in range

`on` (default) | `off`

Select this check box to make the range of valid input amplitudes include the lower bound.

Command-Line Information
Parameter: `min_included`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Enable assertion` — Enable or disable the check

`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Select icon type` — Select icon type

`graphic` | `text`

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: `icon`
Type: character vector
Values: `'graphic'` | `'text'`
Default: `'graphic'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Check Static Range.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

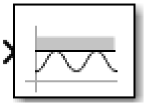
Check Dynamic Range | Check Static Lower Bound | Check Static Upper Bound

Introduced before R2006a

Check Static Upper Bound

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

Library: Simulink / 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.

Use the blocks in the Model Verification library 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 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal checked against the upper bound specified by the **Lower bound** parameter.

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

Output

`Port_1` — Assertion output signal

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a `Boolean`. Otherwise the data type of the signal is a `double`.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: `double` | `Boolean`

Parameters

`Upper bound` — Upper boundary value

scalar | vector | matrix

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

Command-Line Information
Parameter: `max`
Type: character vector
Values: `scalar` | `vector` | `matrix`
Default: `'0'`

`Inclusive boundary` — Include the upper bound in range

`on` (default) | `off`

Select this check box to make the range of valid input amplitudes include the upper bound.

Command-Line Information
Parameter: `max_included`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Enable assertion` — Enable or disable the check

`on` (default) | `off`

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: `enabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`' '` (default) | MATLAB expression

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.

Command-Line Information
Parameter: `callback`
Type: character vector
Values: MATLAB expression
Default: `' '`

`Stop simulation when assertion fails` — Halt simulation when check fails

`on` (default) | `off`

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: `stopWhenAssertionFail`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

Selecting this check box causes the 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`.

Command-Line Information
Parameter: `export`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Select icon type` — Select icon type

`graphic` | `text`

Specify the type of icon used to display this block in a block diagram. 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.

Command-Line Information
Parameter: `icon`
Type: character vector
Values: `'graphic'` | `'text'`
Default: `'graphic'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`

Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

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

Not recommended for production code.

HDL Code Generation

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

For information about HDL Code Generation, see Check Static Upper Bound.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

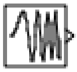
Check Dynamic Lower Bound | Check Dynamic Range | Check Dynamic Upper Bound | Check Static Lower Bound

Introduced before R2006a

Chirp Signal

Generate sine wave with increasing frequency

Library: Simulink / 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.

Limitations

- 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.

Ports

Output

`Port_1` — Chirp signal

scalar | vector | matrix | N-D array

Sine wave whose frequency increases at a linear rate with time. The chirp signal can be a scalar, vector, matrix, or N-D array.

Data Types: `double`

Parameters

`Initial frequency` — Initial frequency (Hz)

`0.1` (default) | scalar | vector | matrix | N-D array

The initial frequency of the signal, specified as a scalar, vector, matrix, or N-D array.

Programmatic Use
Block Parameter: `f1`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'0.1'`

`Target time` — Target time (seconds)

`100` (default) | scalar | vector | matrix | N-D array

Time, in seconds, at which the frequency reaches the **Frequency at target time** parameter value. You specify the **Target time** as a scalar, vector, matrix, or N-D array. After the target time is reached, the frequency continues to change at the same rate.

Programmatic Use
Block Parameter: `T`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'100'`

`Frequency at target time` — Frequency (Hz)

`1` (default) | scalar | vector | matrix | N-D array

Frequency, in Hz, of the signal at the target time, specified as a scalar, vector, matrix, or N-D array.

Programmatic Use
Block Parameter: `f2`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'1'`

`Interpret vector parameters as 1-D` — Treat vector parameters as 1-D

`on` (default) | `off`

When you select this check box, any column or row matrix values you specify 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”.

Programmatic Use
Block Parameter: `VectorParams1D`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

See Also

Sine Wave

Topics

“Creating Signals”

Introduced before R2006a

Clock

Display and provide simulation time
Library: Simulink / 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.

Ports

Output

`Port_1` — Sample time
scalar

Sample time, specified as the current simulation time at each simulation time step.

Data Types: `double`

Parameters

`Display time` — Display simulation time on block icon
`off` (default) | `on`

Select this check box to display the simulation time as part of the Clock block icon. When you clear this check box, the simulation time does not appear on the block icon.

Programmatic Use
Block Parameter: `DisplayTime`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Decimation` — Interval at which to update block icon

`10` (default) | positive integer

Specify the interval at which Simulink updates the Clock icon as a positive integer.

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.

Dependencies

To display the simulation time on the block icon, you must select the **Display time** check box.

Programmatic Use
Block Parameter: `Decimation`
Type: character vector
Value: scalar
Default: `'10'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

See Also

Digital Clock

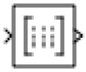
Topics

“Sample Time”

Introduced before R2006a

Combinatorial Logic

Implement truth table
Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
vector

Input signal, specified as a vector. 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`.

Data Types: `double` | `Boolean`

Output

`Port_2` — Output signal
scalar | vector

Output signal, `double` if the truth table contains non-Boolean values of type `double`; `Boolean` otherwise. 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.

Data Types: `double` | `Boolean`

Parameters

`Truth table` — Matrix of outputs

matrix

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

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

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:

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

Programmatic Use
Block Parameter: `TruthTable`
Type: character vector
Values: matrix
Default: `'[0 0;0 1;0 1;1 0;0 1;1 0;1 0;1 1]'`

Block Characteristics

Data Types	`double` | `Boolean`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

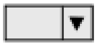
See Also

Bit Clear | Bit Set | Compare To Constant | Compare To Zero

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.

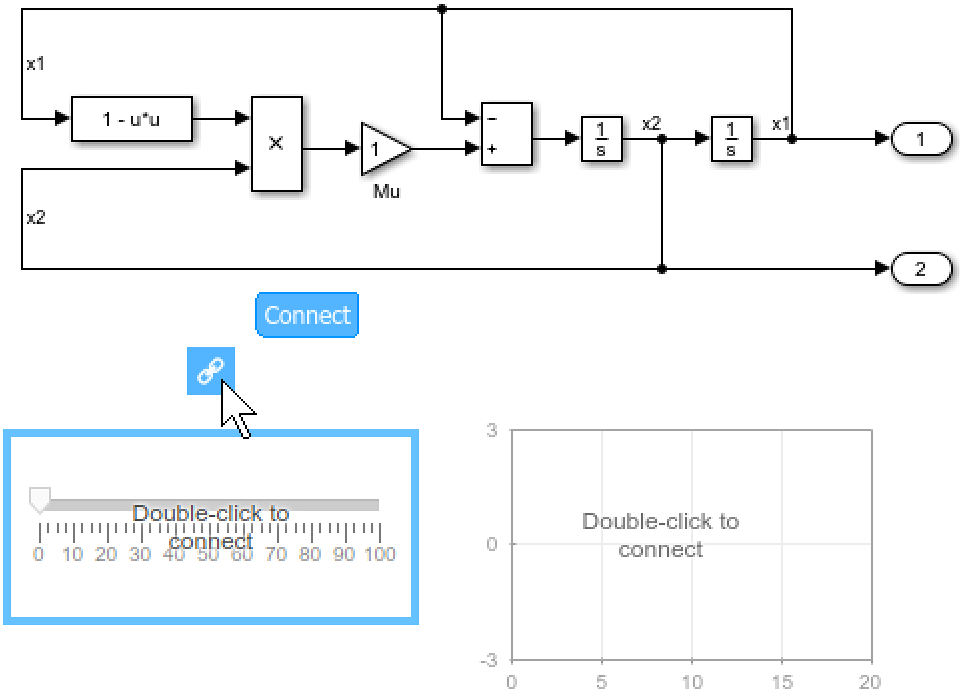
Double-clicking the Combo Box block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

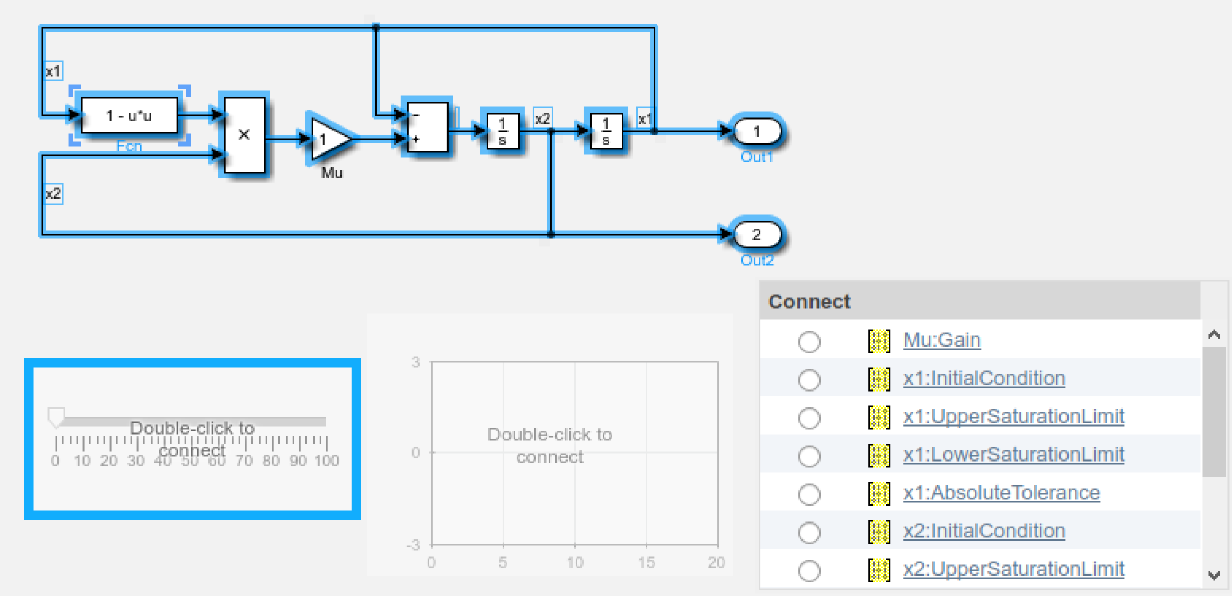
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○ Mu:Gain
○ x1:InitialCondition
○ x1:UpperSaturationLimit
○ x1:LowerSaturationLimit
○ x1:AbsoluteTolerance
○ x2:InitialCondition
○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

See Also

Radio Button | Rotary Switch

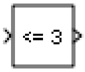
Topics

"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: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar

Input signal, specified as a scalar, is compared with zero.The block first converts its **Constant value** parameter to the input data type, and then performs the specified operation.

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

Output

`Port_1` — Output signal
0 | 1

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

Data Types: `uint8` | `Boolean`

Parameters

`Operator` — Logical operator

<= (default) | == | ~= | < | >= | >

This parameter can have these 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.

Programmatic Use
Block Parameter: `relop`
Type: character vector
Values: `'=='` | `'~='`| `'<'` |`'<='`| `'>='`| `'>'`
Default: `'<='`

`Constant value` — Constant to compare with

constant

Specify the constant value to which the input is compared.

Programmatic Use
Block Parameter: `const`
Type: character vector
Value: scalar | vector | matrix | N-D array
Default: `'3.0'`

`Output data type` — Data type of the output

`boolean` (default) | `uint8`

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

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values:`'boolean'` | `'uint8'`
Default: `'boolean'`

`Enable zero-crossing detection` — Select to enable zero-crossing detection

check (default) | uncheck

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`Yes`

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™.

For more information about HDL code generation, see Compare To Constant.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

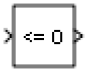
See Also

Combinatorial Logic | Compare To Zero | Logical Operator

Introduced before R2006a

Compare To Zero

Determine how signal compares to zero
Library: Simulink / 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 output is `0` if the comparison is false, and `1` if it is true.

Ports

Input

`Port_1` — Input signal
scalar

Input signal, specified as scalar, is compared with zero. 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.

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

Output

`Port_1` — Output signal
0 | 1

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

The block output is `uint8` or `boolean`, depending on your selection for the **Output data type** parameter.

Data Types: `uint8` | `Boolean`

Parameters

`Operator` — Logical operator
<= (default) | == | ~= | < | >= | >

This 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.

Programmatic Use
Block Parameter: `relop`
Type: character vector
Values: `'=='` | `'~='`| `'<'` |`'<='`| `'>='`| `'>'`
Default: `'<='`

`Output data type` — Data type of the output
`boolean` (default) | `uint8`

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

Programmatic Use
Block Parameter: `OutDataTypeStr`

Type: character vector
Values:'boolean' | 'uint8'
Default: 'boolean'

Enable zero-crossing detection — Select to enable zero-crossing detection
on (default) | off

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

Programmatic Use
Block Parameter: ZeroCross
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes

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™.

For more information about HDL code generation, see Compare To Zero.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

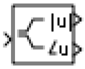
See Also

Bitwise Operator | Compare To Constant | Logical Operator | String Compare

Introduced before R2006a

Complex to Magnitude-Angle

Compute magnitude and/or phase angle of complex signal
Library: Simulink / Math Operations



Description

The Complex to Magnitude-Angle block 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Complex input signal that the block computes and outputs the magnitude and/or the phase angle.

Data Types: `single` | `double`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal that is the magnitude and/or phase angle of the input signal. To choose the output, set the **Output** parameter.

Data Types: `single` | `double`

Parameters

`Output` — Magnitude and/or phase angle output specification

`Magnitude and angle` (default) | `Magnitude` | `Angle`

Specify if the output is the magnitude and/or the phase angle in radians of the input signal.

Command-Line Information
Parameter: `Output`
Type: character vector
Values: `'Magnitude and angle'` | `'Magnitude'` | `'Angle'`
Default: `'Magnitude and angle'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single`

Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

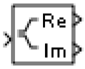
See Also

Complex to Real-Imag | Real-Imag to Complex

Introduced before R2006a

Complex to Real-Imag

Output real and imaginary parts of complex input signal
Library: Simulink / Math Operations



Description

The Complex to Real-Imag block 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Complex input signal that the block computes and outputs the real and/or imaginary part.

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

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal that is the real and/or imaginary part of the input signal. To choose which part is output, set the **Output** parameter.

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

Parameters

`Output` — Real and/or imaginary output specification

`Real and imag` (default) | `Real` | `Imag`

Specify if the output is the real and/or imaginary part of the input signal.

Command-Line Information
Parameter: `Output`
Type: character vector
Values: `'Real and imag'` | `'Real'` | `'Imag'`
Default: `'Real and imag'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`

Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For information about HDL Code Generation, see Complex to Real-Imag.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

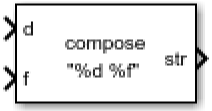
Complex to Magnitude-Angle | Real-Imag to Complex

Introduced before R2006a

Compose String

Compose output string signal based on specified format and input signals

Library: Simulink / String



Description

The Compose String block composes output string signal based on the format specifier listed in the **Format** parameter. The **Format** parameter determines the number of input signals. If there are multiple inputs, the block constructs the string by combining these multiple inputs in order, and applying the associated format specifier, one format specifier for each input. Each format specifier starts with a percent sign, `%`, followed by the conversion character. For example, `%f` formats the input as a floating point output. To supplement the string output, you can also add a character to the format specification. Use this block to compose and format an output string signal from a multiple inputs.

For example, if the **Format** parameter contains `"%s is %f"`, the block expects two inputs, a string signal and a single or double signal. If the first input is the string `"Pi"` and the second input is a double value `3.14`, the output is `"Pi is 3.14"`.

When a MinGW® compiler compiles code generated from the block, running the compiled code may produce nonstandard results for floating-point inputs. For example, a numeric input of `501.987` returns the string `"5.019870e+002"` instead of the expected string `"5.019870e+02"`.

Ports

Input

d — Data for first part of string

scalar

Data for the first part of string, specified as a scalar. The **Format** parameter determines the port label and the format of the input data. For example, if the first item in the **Format** parameter is `%d`, the port label is d.

The data type of the input signal must be compatible with the format specifier in the **Format** parameter. For more information, see the **Format** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

`f` — Data for second part of string

scalar

Data for the second part of string, specified as a scalar. The **Format** parameter determines the port label and the format of the input data. For example, if the first item in the **Format** parameter is `%f`, the port label is f.

The data type of the input signal must be compatible with the format specifier specified in the **Format** parameter. For more information, see the **Format** parameter.

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

`Port_N` — Data for N parts of string

scalar (default)

Data for *N* parts of string, specified as a scalar. The **Format** parameter determines the port label and the format of the input data. For example, if the corresponding item in the **Format** parameter is `%f`, the port label is f.

The data type of the input signal must be compatible with the format specifier in the **Format** parameter. For more information, see the **Format** parameter.

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

Output

`str` — Output string

scalar

Output string composed of inputs, specified as a scalar.

Data Types: `string`

Parameters

`Format` — Format input data
"%d %f" (default) | scalar

Format of input data, specified as a scalar.

For more information about acceptable format specifiers, see the Algorithms section.

`Output data type` — Output data type
`string` (default) | `scalar`

Output data type, specified using the string data type to specify a string with no maximum length.

To specify a string data type with a maximum length, specify `stringtype(`*`N`*`)`. For example, `stringtype(31)` creates a string data type with a maximum length of 31 characters.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Block Characteristics

Data Types	`double` | `single` | `base integer` | `string`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No

Zero-Crossing Detection	No

Algorithms

A formatting specifier starts with a percent sign, %, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype specifiers between % and the conversion character. (Spaces are invalid between specifiers and are shown here only for readability).

The Compose String block uses this format specifier prototype:

```
%[flags][width][.precision][length]specifier
```

Conversion Character

This table shows conversion characters to format numeric and character data as text.

Value Type	Conversion	Details
Integer, signed	%d or %i	Base 10
Integer, unsigned	%u	Base 10
	%o	Base 8 (octal)
	%x	Base 16 (hexadecimal), lowercase letters a–f
	%X	Same as %x, uppercase letters A–F
Floating-point number	%f	Floating-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	%e	Exponential notation, such as 3.141593e+00 (Use a precision operator to specify the number of digits after the decimal point.)

Value Type	Conversion	Details
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
String	`%s`	The type of the output text is the same as the type of **Format**.

Optional Operators

The optional identifier, flags, field width, precision, and operators further define the format of the output text.

- **Flags**

`'–'`	Left-justify. Works with all specifiers. **Example:** `%-5.2f` **Example:** `%-10s`
`'+'`	Always print a sign character (+ or –) for any numeric value. Works with all specifiers except `u`, `o`, `x`, `X`, and `s`. **Example:** `%+5.2f` Right-justify text. **Example:** `%+10s`
`' '`	Insert a space before the value. Works with all specifiers except `u`, `o`, `x`, `X`, and `s`. **Example:** `% 5.2f`

`'0'`	Pad to field width with zeros before the value. Works with all specifiers except `s`. **Example:** `%05.2f`
`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. Works with all specifiers except `d`, `i`, `u`, and `s`. **Example:** `%#5.0f`

- **Field Width**

  Minimum number of characters to print.

  The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`
d, i, o, u, x, X	Minimum number of digits to be written. Outputs shorter than the specified precision are padded with leading zeros. **Example**: `"%.4d"` prints 5 as `'0005'`
For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `'3.142'`
s	Maximum number of characters to be written to the output. **Example:** `"%.2s"` prints `"Hello!"` as `"He"`

Note If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

Text Before or After Formatting Operators

Special Character	Representation
Single quotation mark	''
Percent character	%%
Backslash	\\
Alarm	\a
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v
Character whose Unicode® numeric value can be represented by the hexadecimal number, N	\xN **Example:** sprintf('\x5A') returns 'Z'
Character whose Unicode numeric value can be represented by the octal number, N	\N **Example:** sprintf('\132') returns 'Z'

Format can also include additional text before a percent sign, %, or after a conversion character. The text can be:

- Ordinary text to print.
- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in formatSpec.

Length Specifiers

The Format String block supports the h and l length subspecifiers. These specifiers can change according to the **Configuration Parameters > Hardware Implementation > Number of bits** settings

Length	d i	u o x X	f e E g G	s
No length specifier	`int`	`unsigned int`	`double` (default), `single`	`string`
`h`	`short`	`unsigned short`	—	—
`l`	`long`	`unsigned long`	—	—

Note for Specifiers that Specify Integer Data Types (`d`, `i`, `u`, `o`, `x`, `X`)

Target `int`, `long`, and `short` type sizes are controlled by settings in the **Configuration Parameters** > **Hardware Implementation** pane. For example, if the target `int` is 32 bits and the specifier is `%u`, then the expected input type will be `uint32`. However, the input port accepts any built-in integer type of that size or smaller with the `%u` specifier

Notes for Specifiers that Specify Floating Point Data Types (`f`, `e`, `E`, `g`, `F`)

- Do not use `l` and `h` with these specifiers. Do not use the length subspecifier (for example, `%f` is allowed, but `%hf` and `%lf` are not allowed) .
- Ports that correspond with these specifiers accept both `single` and `double` data types.

Note for Specifiers that Specify the String Data Type (`s`)

- The `s` specifier does not work with the `l` or `h` subspecifiers, and only accepts a string input data type.

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String to Double | String to Single | Substring | To String | `sprintf`

Topics

"Display and Extract Coordinate Data"
"Simulink Strings"

Introduced in R2018a

Configurable Subsystem

Represent any block selected from user-specified library of blocks

Library: Simulink / Ports & Subsystems

Description

The Configurable Subsystem block represents one of a set of blocks contained in a specified library of blocks. The context menu of the Configurable Subsystem block 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 box.

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.

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 so, 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 box by double-clicking it. The dialog box 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. To apply the changes and close the dialog box, click the **OK** button.
7. Select **Block Choice** from the context menu of the Configurable Subsystem block.

   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 context menu of that Configurable Subsystem instance.
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 its 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', ...);
```

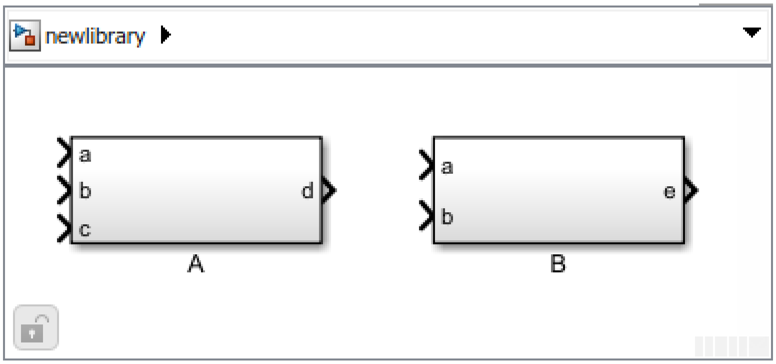
Mapping I/O Ports

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.

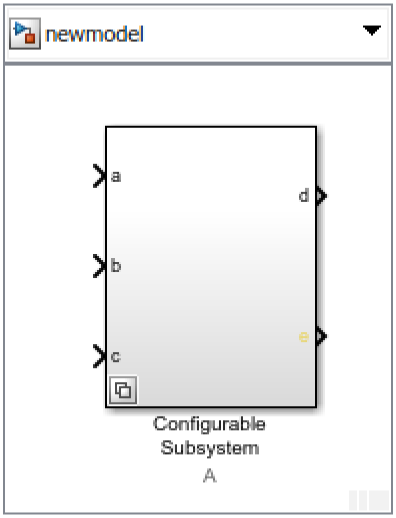
This mapping allows a user to change the library block represented by a Configurable Subsystem block without having to rewire connections to the Configurable Subsystem block.

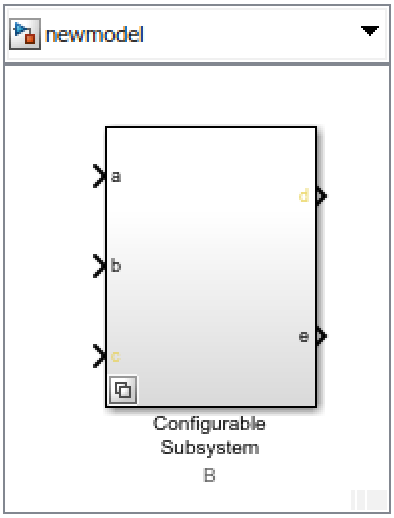
For example, suppose that a library contains two blocks A and B and that block A has input ports labeled a, b, and c and an output port labeled d and that block B has input ports labeled a and b and an output port labeled e.



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. 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array | bus

Input signal to the Configurable Subsystem. The block that the Configurable Subsystem represents determines the supported data types and dimensions of the input signal.

Dependencies

The number of input ports depends on the blocks in the library that the Configurable Subsystem represents. For more information, see “Mapping I/O Ports” on page 1-214.

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

Output

`Port_1` — Output signal

scalar | vector | matrix | N-D array | bus

Output signal from the Configurable Subsystem. The block that the Configurable Subsystem represents determines the output data types and dimensions.

Dependencies

The number of output ports depends on the blocks in the library that the Configurable Subsystem represents. For more information, see “Mapping I/O Ports” on page 1-214.

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

Parameters

List of block choices — Block members of the configurable subsystem

no default

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

Programmatic Use
Block Parameter: MemberBlocks
Type: cell array of character vectors
Values: cell array of block names as character vectors
Default: {''}

Port names — Port names

no default

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.

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

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 code generation support depends on block implementation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem | Variant Subsystem

Topics

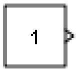
"Model Discretizer"

Introduced before R2006a

Constant

Generate constant value

Library: Simulink / Commonly Used Blocks
Simulink / 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** check box. Otherwise, if you specify a vector for the **Constant value** parameter, the block treats that vector as a matrix.

Tip To output a constant enumerated value, consider using the Enumerated Constant block instead. The Constant block provides block parameters that do not apply to enumerated types, such as **Output minimum** and **Output maximum**.

Using Bus Objects as the Output Data Type

The Constant block supports nonvirtual buses as the output data type. Using a bus object as the output data type can help simplify your model. If you use a bus object as the output data type, set the **Constant value** to `0` or to a MATLAB structure that matches the bus object.

Using Structures for the Constant Value of a Bus

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”.

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”.

Ports

Output

`Port_1` **— Constant value**
scalar | vector | matrix | N-D array

Constant value, specified as a real or complex valued scalar, vector, matrix, or N-D array. By default, the Constant block outputs a signal whose dimensions, 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.

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 more information, see `Simulink.BusElement`.

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

Parameters

Main

`Constant value` — Constant output value

`1` (default) | scalar | vector | matrix | N-D array

Specify the constant value output of the block.

- 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 one of these options:

  - 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 “Using Bus Objects as the Output Data Type” on page 1-220.
- For nonbus data types, Simulink converts this parameter from its value data type to the specified output data type offline, using a rounding method of nearest and overflow action of saturate.

Programmatic Use
Block Parameter: `Value`
Type: character vector
Value: scalar | vector | matrix | N-D array
Default: `'1'`

`Interpret vector parameters as 1-D` — Treat vectors as 1-D

`on` (default) | `off`

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.

- When you select this check box, the block 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.
- When you clear this check box, the block does not output a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector.

Programmatic Use
Block Parameter: VectorParams1D
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Sample time — Sampling interval

inf (default) | scalar | vector

Specify the interval between times that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

The default value of inf 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.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar | vector
Default: 'inf'

Signal Attributes

Output minimum — Minimum output value for range checking

[] (default) | scalar

Specify the lower value of the output range that Simulink checks 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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[ ]` (default) | scalar

Specify the upper value of the output range that Simulink checks 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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[ ]'`

`Output data type` — Output data type

`Inherit: Inherit from 'Constant value'` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `boolean` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class name>` | `Bus: <object name>` | `<data type expression>`

Specify the output data type. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector

Values: '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>' | 'Bus: <object name>'
Default: 'Inherit: Inherit from 'Constant value''

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated | bus
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
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™.

For more information about HDL code generation, see Constant.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Enumerated Constant | `Simulink.BusElement` | `Simulink.Parameter`

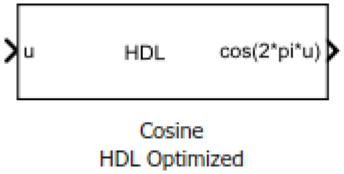
Topics

"Set Block Parameter Values"
"When to Use Bus Objects"
"Specify Initial Conditions for Bus Signals"
"Group Constant Signals into an Array of Buses"

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(2π*u*)
- cos(2π*u*)
- exp(*i*2π*u*)
- sin(2π*u*) and cos(2π*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

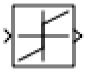
See Also

Sine HDL Optimized | Sine, Cosine | Trigonometric Function

Introduced in R2016b

Coulomb and Viscous Friction

Model discontinuity at zero, with linear gain elsewhere
Library: Simulink / 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 = sign(x) .* (Gain .* abs(x) + 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

The input signal to the model of Coulomb and viscous friction.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal

scalar | vector | matrix

The output signal calculated by applying the friction models to the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

To edit the parameters for the Coulomb and Viscous Friction block, double-click the block icon.

`Coulomb friction value` — Static friction offset

`[1320]` (default) | real values

Specify the offset that applies to all input values.

Programmatic Use
Block Parameter: `offset`
Type: character vector
Value: real values
Default: `'[1 3 2 0]'`

`Coefficient of viscous friction` — Dynamic friction coefficient

1 (default) | real values

Specify the signal gain for nonzero input values.

Programmatic Use
Block Parameter: `gain`
Type: character vector
Value: real values
Default: `'1'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
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™.

For more information on HDL code generation, see Coulomb and Viscous Friction.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

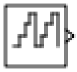
Backlash | Dead Zone

Introduced before R2006a

Counter Free-Running

Count up and overflow back to zero after reaching maximum value for specified number of bits

Library: Simulink / 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.

Note 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.

Ports

Output

`Port_1` — Count value
scalar

Count value, specified as an unsigned integer of 8 bits, 16 bits, or 32 bits.

Data Types: `uint8` | `uint16` | `uint32`

Parameters

`Number of Bits` — Number of bits

16 (default) | scalar

Specify the number of bits as a finite, real-valued. When you specify:

- A positive integer, for example `8`, the block counts up to $2^8 - 1$, which is `255`.
- An unsigned integer expression, for example `uint8(8)`, the block counts up to `uint8(`$2^{\mathtt{uint8(8)}} - 1$`)`, which is `254`.

Programmatic Use
Block Parameter: `NumBits`
Type: character vector
Values: scalar
Default: `'16'`

`Sample time` — Interval between samples

`-1` (default) | scalar | vector

Specify the time interval between samples as a scalar (`sampling period`), or a two-element vector (`[sampling period, initial offset]`). To inherit the sample time, set this parameter to `-1`. For more information, see "Specify Sample Time".

Programmatic Use
Block Parameter: `tsamp`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Block Characteristics

Data Types	`base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`

Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

HDL Code Generation

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

For more information about HDL code generation, see Counter Free-Running.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

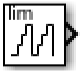
See Also

Counter Limited | MATLAB Function

Introduced before R2006a

Counter Limited

Count up and wrap back to zero after outputting specified upper limit
Library: Simulink / 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.

Note 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.

Ports

Output

`Port_1` — Count value
scalar

Count value, specified as an unsigned integer of 8 bits, 16 bits, or 32 bits. The block uses the smallest number of bits required to represent the upper limit.

Data Types: `uint8` | `uint16` | `uint32`

Parameters

`Upper limit` — Upper limit
7 (default) | scalar

Specify the upper limit for the block to count to as a finite, real-valued scalar.

Programmatic Use
Block Parameter: `uplimit`
Type: character vector
Values: scalar
Default: `'7'`

`Sample time` — Interval between samples

`-1` (default) | scalar | vector

Specify the time interval between samples as a scalar (`sampling period`), or a two-element vector (`sampling period, initial offset`). To inherit the sample time, set this parameter to `-1`. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `tsamp`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Block Characteristics

Data Types	`Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

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.
- Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

HDL Code Generation

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

For more information about HDL code generation, see Counter Limited.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Counter Free-Running

Introduced before R2006a

Custom Gauge

Display an input value on a customized gauge

Library: Simulink / Dashboard

Description

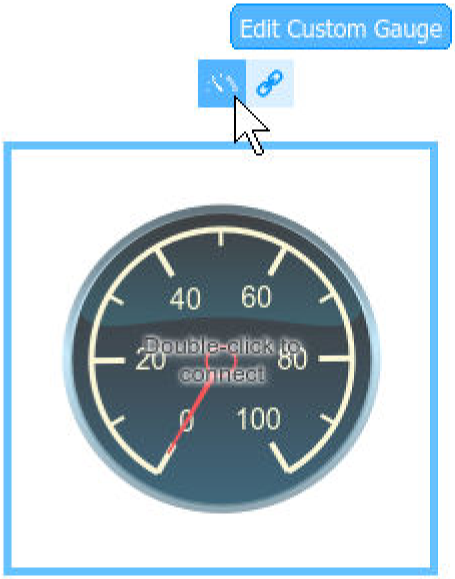
The Custom Gauge block displays the value of the connected signal on a gauge face that you can customize to look like a gauge in a real system. For example, you could create an engine rpm gauge or fuel indicator for an automotive model.

1
2
3
4
5
6
7
8
9
10
x1000

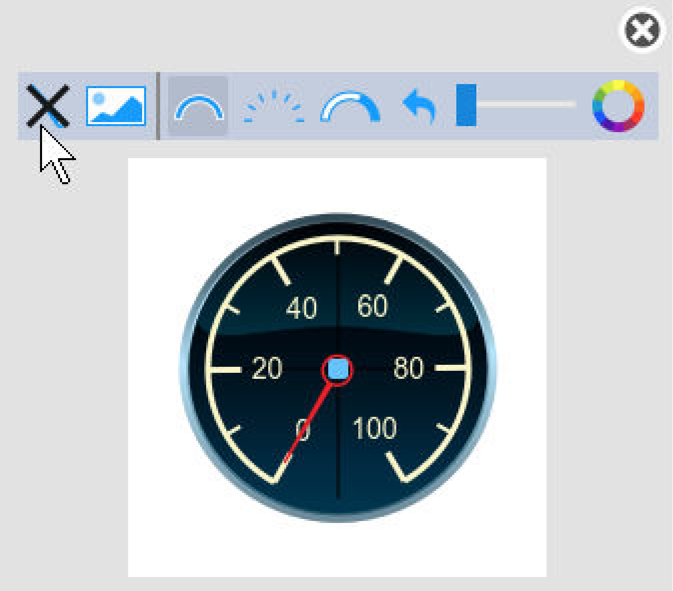
The Custom Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range and tick values on the Custom Gauge block to fit your data. Use the Custom Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Create A Custom Gauge

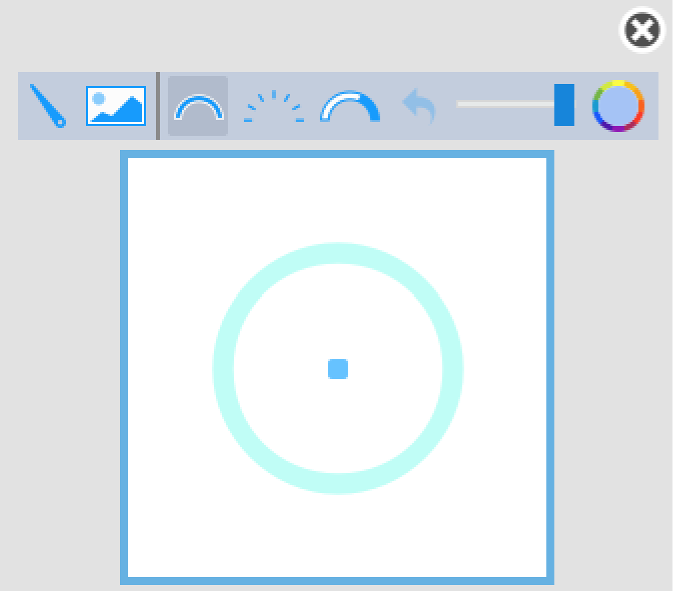
When you add a Custom Gauge block to your model, the block is preconfigured with a default design. You can use the preconfigured Custom Gauge block like you would other Dashboard blocks, or you can enter Custom Gauge edit mode to customize the appearance. The **Edit Custom Gauge** button appears above the Custom Gauge block when you pause on it.



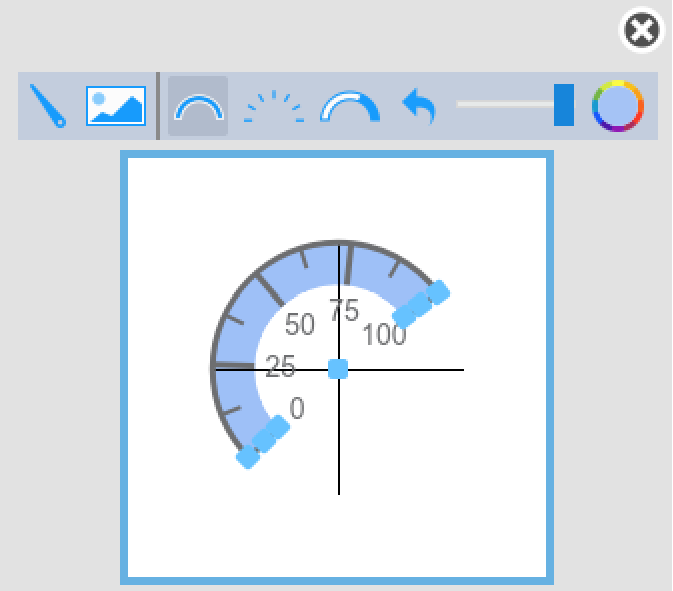
In Custom Gauge edit mode, the customization menu appears above the block. Separate images specify the appearance of the needle and the background. You can also customize the size of the dial arc and the color and transparency of the arc and tick marks. To build from the default design, you can individually select the needle image, background image, and arc to reposition and resize each feature. If you want to start from a blank canvas, you can remove the arc with the **Clear Arc** ↶ button. You can remove the needle and background images by clicking the **X** that appears when you pause on the needle or background image buttons.



The blank canvas for the Custom Gauge block shows a circle with its center marked to help guide alignment as you build your customized gauge. You can drag the center marker to align the center of your arc with the center of the arc in your background image. To add an image for the needle or the gauge background, click the button for the type of image you want to add. Then, select the image you want to use. When you add a gauge background image, the Custom Gauge block **Lock Aspect Ratio** parameter is selected. You can resize the block without distorting the image. If you want to change the aspect ratio of the block after adding a gauge background image, clear the **Lock Aspect Ratio** parameter.



To draw the arc for your gauge, click and drag along the guide circle. As you draw the arc, crosshairs appear to assist you with aligning the start and end points of the arc. After you release the mouse, you can continue to adjust the arc position and size. The arc options on the toolbar control the appearance of the arc, tick marks, and the value arc that indicates the value of the connected signal or parameter during simulation. The slider adjusts the transparency of the selected feature, and the color wheel allows you to specify the color. You can configure the arc as fully transparent when your background image includes the representation of the gauge face you want to use.

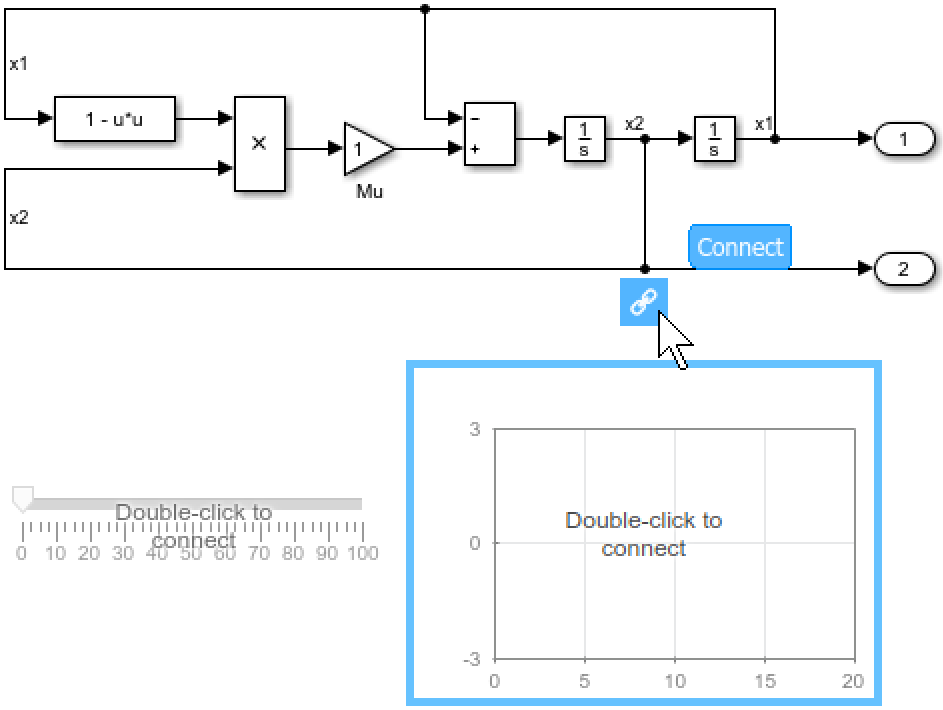


When you finish editing the appearance of your Custom Gauge block, click the **Exit** button in the upper-right of the model canvas or press **Esc** to exit Custom Gauge edit mode.

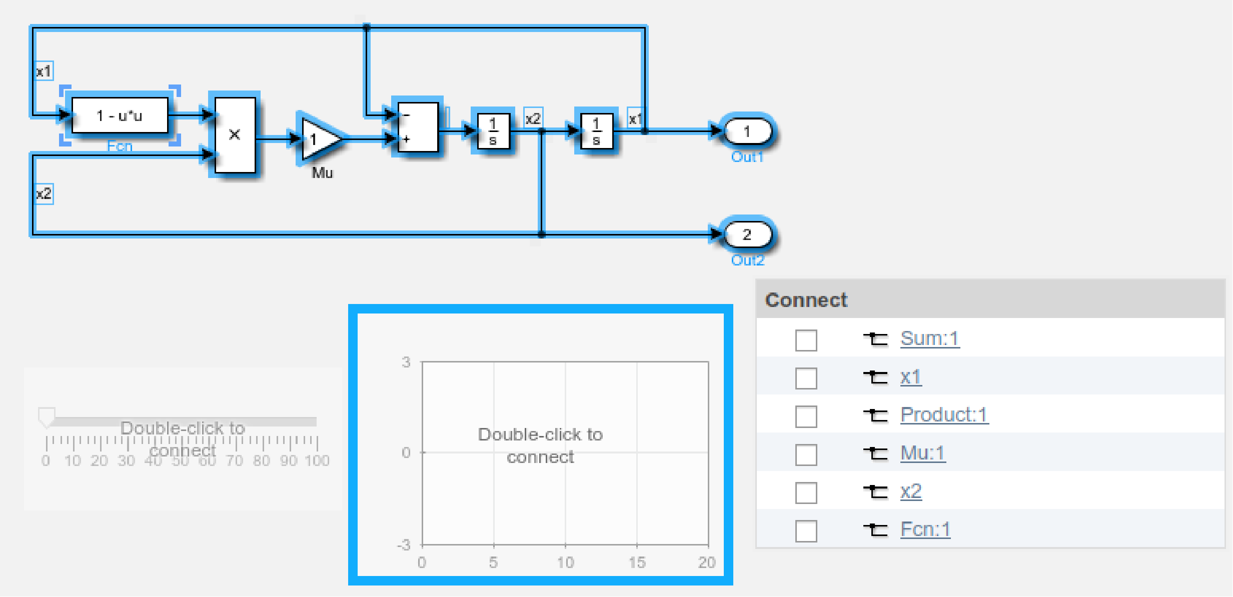
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect	
□	Sum:1
□	x1
□	Product:1
□	Mu:1
□	x2
□	Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced 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.

Parameters

`Connection` — Select a signal to connect and display

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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 arc. 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 arc. 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 arc. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

`Scale Colors` — Color indications on gauge arc

colors for arc ranges

Color specifications for ranges on the arc. 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.

`Lock Aspect Ratio` — Maintain background image aspect ratio

`on` (default) | `off`

Maintain the background image aspect ratio when resizing the block. By default, when you specify a background image for the block, the **Lock Aspect Ratio** check box is selected.

`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.

See Also

Gauge | Half Gauge | Linear Gauge | Quarter Gauge

Topics

“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2018b

Dashboard Scope

Trace signals during simulation
Library: Simulink / Dashboard

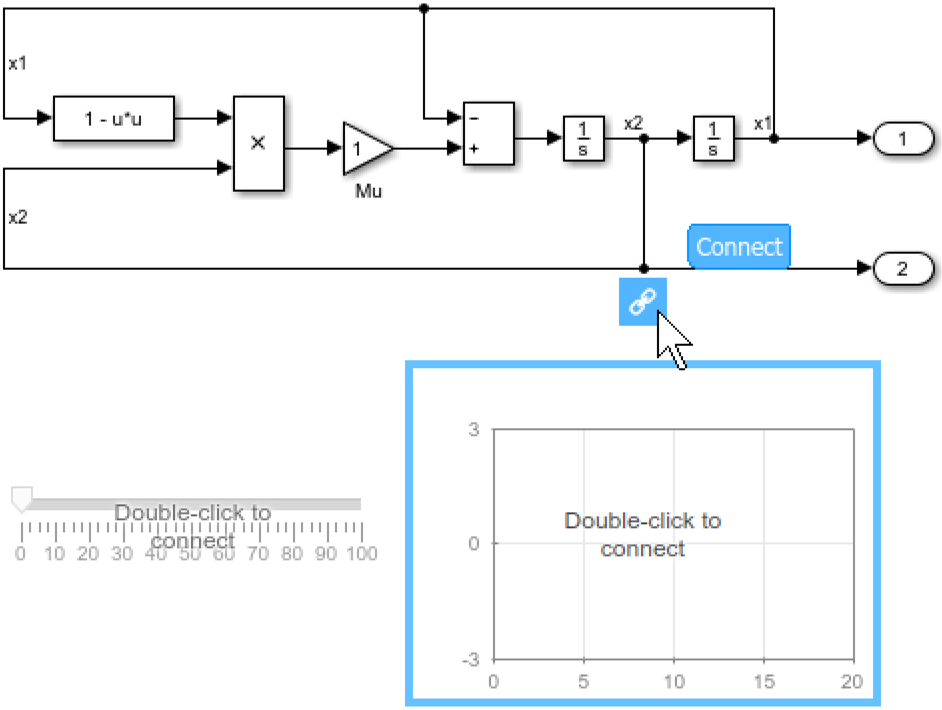
Description

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 signals of any data type that Simulink supports, including enumerated data types. The Dashboard Scope block can display up to eight signals from a matrix or bus.

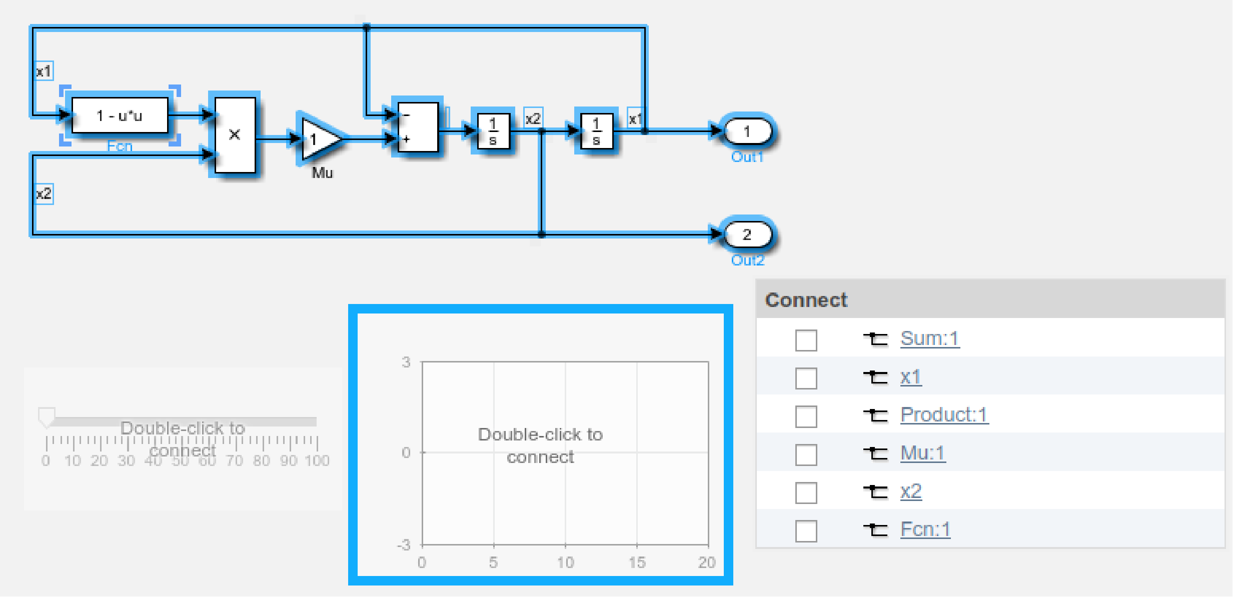
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



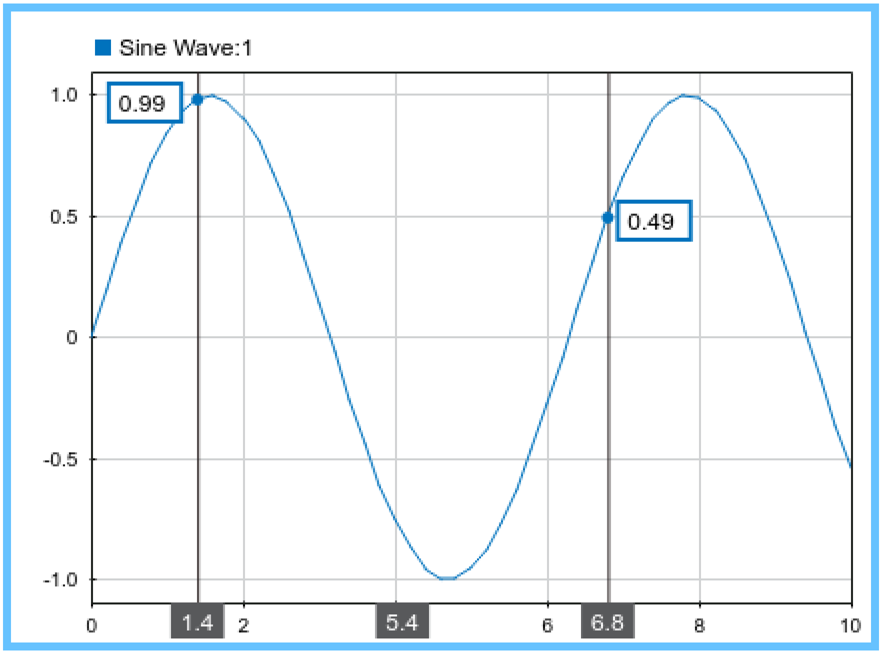
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Data Cursors

You can add data cursors to the Dashboard Scope to inspect the displayed signals. The data cursors show when the Dashboard Scope block is selected. With the Dashboard Scope block selected, you can move cursors along the displayed signals to see the data values corresponding to each time sample. When you display two cursors, a box between the cursors along the time axis displays the time difference between the two cursors.



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.

Complex Signals

The Dashboard Scope block displays complex signals according to their **Complex Format**. You can configure the **Complex Format** for a signal using the **Instrumentation Properties** dialog box, accessible when you right-click the logging badge for the signal and select **Properties**. A signal can have a **Complex Format** of **Real-Imaginary**, **Magnitude-Phase**, **Magnitude**, or **Phase**. When you set the **Complex Format** for a signal to **Real-Imaginary** or **Magnitude-Phase**, the Dashboard Scope block displays both components of the signal together. The real or magnitude component displays in the color indicated in the **Connection** table. The imaginary or phase component displays in a different shade of the color indicated in the **Connection** table.

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 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.

Parameters

`Connection` — Select signals to connect and display

signal connection options

Select one or more signals to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select 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 scope display.

When **Time Span** is set to `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 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 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 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.

See Also

Topics

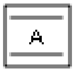
"Tune and Visualize Your Model with Dashboard Blocks"
"Decide How to Visualize Simulation Data"

Introduced in R2015a

Data Store Memory

Define data store

Library: Simulink / 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 For Each subsystem.

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.

Parameters

Main

`Data store name` — Name for the data store

A (default) | character vector | string

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.

Programmatic Use
Block Parameter: `DataStoreName`
Type: character vector
Values: `'A' | ...`
Default: `'A'`

`Rename All` — Rename this data store throughout the model

button

Rename this data store everywhere the Data Store Read and Data Store Write blocks use it in a model.

Limitations

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` — Path to connected Data Store Read/Write blocks

block path

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. Clicking a block path displays and highlights that block in your model.

Signal Attributes

`Initial value` — Initial value of data store

`0` (default) | scalar | vector | matrix | N-D array

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”.

Programmatic Use

Block Parameter: `InitialValue`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'0'`

`Minimum` — Minimum output value for range checking

`[]` (default) | scalar

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[ ]'`

`Maximum` — Maximum output value for range checking

`[]` (default) | scalar

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[ ]'`

`Data type` — Output data type

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

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.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

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>'`

Default: `'Inherit: auto'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Dimensions (-1 to infer from Initial value)` — Dimensions of data store

`-1` (default) | scalar | vector | matrix

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.

Programmatic Use
Block Parameter: `Dimensions`
Type: character vector
Values: scalar | vector | matrix
Default: `'-1'`

`Interpret vector parameters as 1-D` — Interpret vectors as 1-D

`on` (default) | `off`

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”.

Programmatic Use

Block Parameter: `VectorParams1D`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Signal type` — Complexity of data store values

`auto` (default) | `real` | `complex`

Specify the numeric type, real or complex, of the values in the data store.

Programmatic Use

Block Parameter: `SignalType`
Type: character vector
Values: `'auto' | 'real' | 'complex'`
Default: `'auto'`

`Share across model instances` — Allow Model blocks to read from the same data store

`off` (default) | `on`

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 Store Between Instances of a Reusable Algorithm”. For more information, see “Share Data Among Referenced Model Instances”.

Programmatic Use
Block Parameter: `ShareAcrossModelInstances`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Data store name must resolve to Simulink signal object` — Require data store name resolve to Simulink signal object

`off` (default) | `on`

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.

Programmatic Use
Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package

`Simulink.Signal` (default) | object of a class that is derived from `Simulink.Signal`

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 "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder). For information about custom storage classes, see "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder).

Programmatic Use
Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | ...`
Default: `'Simulink.Signal'`

Storage class — Storage class for code generation

Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Volatile (Custom) | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder). For information about custom storage classes, see "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (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 "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder).

Programmatic Use

Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

TypeQualifier — Storage type qualifier

'' (default) | const | volatile | ...

Specify a storage type qualifier such as `const` or `volatile`.

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.

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 “Data Objects”.

Dependencies

To enable this parameter, set **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `Model default`. This parameter is hidden unless you previously set its value.

Programmatic Use
Block Parameter: `RTWStateStorageTypeQualifier`
Type: character vector
Values: `'' | 'const' | 'volatile' | ...`
Default: `''`

Diagnostics

`Detect Read Before Write` — Action when model attempts to read data before writing in current time step

`warning` (default) | `none` | `error`

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.

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `ReadBeforeWriteMsg`
Type: character vector
Values: `'none' | 'warning' | 'error'`
Default: `'warning'`

`Detect Write After Read` — Action when block attempts to write after reading in same time step

`warning` (default) | `none` | `error`

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.

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `WriteAfterReadMsg`
Type: character vector
Values: `'none' | 'warning' | 'error'`
Default: `'warning'`

`Detect Write After Write` — Action when model writes twice in same time step

`warning` (default) | `none` | `error`

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.

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `WriteAfterWriteMsg`
Type: character vector
Values: `'none' | 'warning' | 'error'`
Default: `'warning'`

Logging

`Log data store data` — Log data store data

`off` (default) | `on`

Select this option to save the values of this signal to the MATLAB workspace during simulation. See “Signal Logging” for details.

Programmatic Use
Block Parameter: `DataLogging`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Logging name` — Name associated with logged signal data

`Use data store name` (default) | `Custom`

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.

Programmatic Use
Block Parameter: `DataLoggingNameMode`
Type: character vector
Values: `'SignalName' | 'Custom'`
Default: `''`

Note If you set `DataLoggingNameMode` to `Custom`, you must specify the name associated with logged signal data using the `DataLoggingName` parameter.

Block Parameter: `DataLoggingName`
Type: character vector
Values: character vector
Default: `''`

`Limit data points to last` — Discard all but the last N data points

`5000` | non-zero integer

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”.

Programmatic Use
Block Parameter: `DataLoggingMaxPoints`
Type: character vector
Values: nonzero integer

Default: `'5000'`

Decimation — Log every Nth data point

2 (default) | integer

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”.

Programmatic Use
Block Parameter: `DataLoggingLimitDataPoints`
Type: character vector
Values: non-zero integer
Default: `'2'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

To generate PLC code for a model that uses a Data Store Memory block, first define a `Simulink.Signal` in the base workspace. Then in the **Signal Attributes** tab of the block parameters, set the data store name to resolve to that of the `Simulink.Signal` object.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Store Read | Data Store Write

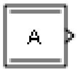
Topics

"Storing Data Using Data Store Memory Blocks"
"Data Stores"
"Choosing How to Store Global Data"
"Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder)
"Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder)
"Access Data Stores with Simulink Blocks"
"Log Data Stores"

Introduced before R2006a

Data Store Read

Read data from data store
Library: Simulink / 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.

Ports

Output

`Port_1` — Values from specified data store
scalar | vector | matrix | N-D array

Values from the specified data store, output with the same data type and dimensions as in the data store. The block supports both real and complex signals. You can choose whether to output the entire data store, or only selected elements from the data store.

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”.

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

Parameters

Arguments

`Data store name` — Name of data store from which this block reads

`A` (default) | name of data store

Specifies the name of the data store from which this block reads data. The adjacent drop-down list provides 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 list also includes all `Simulink.Signal` objects in the base and model workspaces. To change the name, select a name from the list or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink 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 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.

Programmatic Use
Block Parameter: `DataStoreName`
Type: character vector
Values: data store name
Default: `'A'`

`Data store memory block` — Data Store Memory block name

block path

This field lists the Data Store Memory block that initialized the store from which this block reads.

`Data store write blocks` — Corresponding Data Store Write blocks

block path

This field lists the path to all 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. Click any entry in this list to highlight the corresponding block in your model.

`Sample time` — Sample time

-1 (default) | scalar | vector

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.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Element Selection

`Elements in the array (Signals in the bus)` — Elements in the associated data store

no default

List of elements in the associated data store. For data stores containing arrays, you can read the whole data store, or you can specify one or more elements of the data store. For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

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 elements that you want to select for reading. Then click **Select>>**.

To refresh the display and reflect modifications to the array or bus used in the data store, click **Refresh**.

Dependencies

The prompt for this section (**Elements in the array** or **Signals in the bus**) depends on the type of data in the data store.

Programmatic Use

Block Parameter: `DataStoreElements`
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)
Default: `''`

`Specify element(s) to select` — MATLAB expression defining the elements to select

no default

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** or **Apply**.

Programmatic Use

Block Parameter: `DataStoreElements`
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)
Default: `''`

`Selected element(s)` — List of selected elements

no default

Displays the elements that you select from the data store. 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**.

Programmatic Use

Block Parameter: `DataStoreElements`
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)

Default: `''`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Store Memory | Data Store Write

Topics

"Data Stores"
"Rename Data Stores"

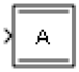
“Order Data Store Access”
“Access Data Stores with Simulink Blocks”
“Data Store Diagnostics”

Introduced before R2006a

Data Store Write

Write data to data store

Library: Simulink / 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”.

Ports

Input

`Port_1` — Values to write to data store

scalar | vector | matrix | N-D array

Values to write to the specified data store. The Data Store Write block accepts a real or complex signal.

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”.

To assign a subset of the bus or matrix elements to the associated data store, use the **Element Assignment** pane. The Data Store Write block icon reflects the elements that you specify. For details, see “Accessing Specific Bus and Matrix Elements”.

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

Parameters

Arguments

`Data store name` — Name of data store from which this block writes

A (default) | name of data store

Specifies the name of the data store to which this block writes data. The adjacent drop-down list provides 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 drop-down list also includes all `Simulink.Signal` objects in the base and model workspaces. To change the name, select a name from the drop-down or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink 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 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.

Programmatic Use
Block Parameter: `DataStoreName`
Type: character vector
Values: data store name
Default: `'A'`

`Data store memory block` — Data Store Memory block name

block path

This field lists the Data Store Memory block that initialized the store to which this block writes.

`Data store read blocks` — Corresponding Data Store Read blocks

block path

This field lists the path to all 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. Click any entry in this list to highlight the corresponding block in your model.

`Sample time` — Sample time

-1 (default) | scalar | vector

The sample time, which 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.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Element Assignment

`Elements in the array (Signals in the bus)` — Elements in the associated data store

no default

List of elements in the associated data store. 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. For data stores with a bus data type, you can expand the tree to view the bus elements. The list displays the maximum dimensions for each element, in parentheses.

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 elements that you want to select for writing. Then click **Select>>**.

To refresh the display and reflect modifications to the array or bus used in the data store, click **Refresh**.

Dependencies

The prompt for this section (**Elements in the array** or **Signals in the bus**) depends on the type of data in the data store.

Programmatic Use
Block Parameter: `DataStoreElements`
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)
Default: `''`

`Specify element(s) to assign` — MATLAB expression defining the elements to assign

no default

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** or **Apply**.

Programmatic Use
Block Parameter: `DataStoreElements`
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)
Default: `''`

`Assigned element(s)` — List of selected elements

no default

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**.

Programmatic Use
Block Parameter: `DataStoreElements`
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)
Default: `''`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Store Memory | Data Store Read

Topics

"Data Stores"
"Rename Data Stores"
"Order Data Store Access"
"Access Data Stores with Simulink Blocks"
"Log Data Stores"
"Data Store Diagnostics"

Introduced before R2006a

Data Type Conversion

Convert input signal to specified data type

Library: Simulink / Commonly Used Blocks
Simulink / Signal Attributes

Description

The Data Type Conversion block converts an input signal of any Simulink data type to the data type that you specify.

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. This parameter does not change the behavior of the block when:

- The input and output do not have a fixed-point data type.
- The input or output has a fixed-point data type with trivial scaling.

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 limits imposed by the scaling of the input and output and 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. Doing so ensures 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 “Convert Data Types in Simulink Models” on page 14-136.

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 `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 **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 these cases:

- To cast a noninteger numeric signal to an enumerated signal.
- To cast a complex signal to an enumerated signal, regardless of the data types of the real and imaginary parts of the complex signal.

See “Simulink Enumerations” for information on working with enumerated types.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. 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. The block converts the input signal to the **Output data type** you specify.

When you are converting fixed-point data types, use the **Input and output to have equal** parameter to determine whether the conversion happens based on the `Real World Value (RWV)` or `Stored Integer (SI)` value of the signal. For more information, see “Convert Fixed-Point Signals” on page 1-284.

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

Output

`Port_1` — Output signal

scalar | vector | matrix | N-D array

Output signal, converted to the data type you specify, with the same dimensions as the input signal.

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

Parameters

`Output minimum` — Minimum output value for range checking

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Output data type

`Inherit: Inherit via back propagation` (default) | `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>` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'`, `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'Enum: <class name>''<data type expression>'`
Default: `'Inherit: Inherit via back propagation'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Input and output to have equal` — Constraint for converting fixed-point data types

`Real World Value (RWV)` (default) | `Stored Integer (SI)`

Specify which type of input and output must be equal, in the context of fixed-point data representation.

- `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.

Programmatic Use
Block Parameter: `ConvertRealWorld`
Type: character vector
Values: `'Real World Value (RWV)' | 'Stored Integer (SI)'`
Default: `'Real World Value (RWV)'`

`Integer rounding mode` — Specify the rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Choose one of these rounding modes.

`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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

See Also

For more information, see “Rounding” (Fixed-Point Designer).

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Data Type Conversion.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Type Conversion Inherited | Data Type Propagation | Data Type Scaling Strip

Topics

“Control Signal Data Types”
“About Data Types in Simulink”
“Simulink Enumerations”
“Fixed Point”

Introduced before R2006a

Data Type Conversion Inherited

Convert from one data type to another using inherited data type and scaling

Library: Simulink / 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. The second input, **u**, is converted to the reference type by inheriting the data type and scaling information. (For a description of the port order for various block orientations, see “Port Location After Rotating or Flipping”.)

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.

Ports

Input

`Port_1` — Reference signal
scalar | vector | matrix | N-D array

Reference signal, defining the data type used to convert input signal **u**.

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

u — Input signal to convert
scalar | vector | matrix | N-D array

Input signal to convert to the reference data type, specified as a scalar, vector, matrix, or N-D array.

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

Output

y — Output signal
scalar | vector | matrix | N-D array

Output is the input signal **u**, converted to the reference data type.

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

Parameters

`Input and Output to have equal` — Constraint for converting fixed-point data types
`Real World Value (RWV)` (default) | `Stored Integer (SI)`

Specify which type of input and output must be equal, in the context of fixed-point data representation.

- `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.

Programmatic Use
Block Parameter: `ConvertRealWorld`
Type: character vector
Values: `'Real World Value (RWV)' | 'Stored Integer (SI)'`
Default: `'Real World Value (RWV)'`

`Integer rounding mode` — Rounding mode for fixed-point operations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Type Conversion | Data Type Propagation

Topics

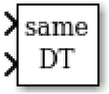
“Control Signal Data Types”
“About Data Types in Simulink”
“Fixed Point”

Introduced before R2006a

Data Type Duplicate

Force all inputs to same data type

Library: Simulink / Signal Attributes



Description

The Data Type Duplicate block forces all inputs to have the same data type. Other attributes of input signals, such as dimension, complexity, and sample time, are 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 backpropagation.

The block can also be useful 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.

Ports

Input

`Port_1` — First input signal

scalar | vector | matrix | N-D array

First input signal, specified as a scalar, vector, matrix, or N-D array. If all signals do not have the same data type, the block returns an error message.

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

`Port_N` — Nth input signal

scalar | vector | matrix | N-D array

Nth input signal, specified as a scalar, vector, matrix, or N-D array. If all signals do not have the same data type, the block returns an error message.

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

Parameters

`Number of input ports` — Number of block inputs

2 (default) | real-valued positive integer

Specify the number of inputs to this block as a real-valued positive integer.

Programmatic Use
Block Parameter: `NumInputPorts`
Type: character vector
Values: real-valued positive integer
Default: `'2'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Data Type Duplicate.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Type Conversion | Data Type Propagation

Topics

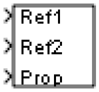
“Control Signal Data Types”
“About Data Types in Simulink”
“Fixed Point”

Introduced before R2006a

Data Type Propagation

Set data type and scaling of propagated signal based on information from reference signals

Library: Simulink / 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 along 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 many choices for propagating data type and scaling information. For example, you can use:

- The number of bits from the **Ref1** reference signal or the number of bits from widest reference signal
- The range from the **Ref2** reference signal or the range of the reference signal with the greatest range
- A bias of zero, regardless of the biases used by the reference signals
- The precision of the reference signal with the least precision

You specify how data type information is propagated using the **Propagated data type** parameter:

- If you select `Specify via dialog`, then you manually specify the data type via the **Propagated data type** edit field.

- If you select `Inherit via propagation rule`, then you must use the parameters described in “Parameters” on page 1-304.

You specify how scaling information is propagated using the **Propagated scaling** parameter:

- If you select `Specify via dialog`, then you manually specify the scaling via the **Propagated scaling** edit field.
- If you select `Inherit via propagation rule`, then you must use the parameters described in “Parameters” on page 1-304.

After you use the information from the reference signals, you can apply a second level of adjustments to the data type and scaling. To do so, use individual multiplicative and additive adjustments. This flexibility has various uses. For example, if you are targeting a DSP, then you can configure the block so that the number of bits associated with a multiply and accumulate (MAC) operation is twice as wide as the input signal, and has a specific 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 forces these three choices to be used. For example, suppose that 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 method makes it easier to create designs that are easily retargeted between fixed-point chips and floating-point chips.

The Data Type Propagation block allows you to set up libraries of useful subsystems that are properly configured based on the connected signals. Without this data type propagation process, subsystems from a library are unlikely to work as desired with most integer or fixed-point signals. Manual intervention would be required to configure the data type and scaling. In many situations, this block can eliminate the manual intervention.

Precedence Rules

The precedence of the dialog box parameters decreases from top to bottom. Also:

- 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.
- `PosRange` is one bit higher than the exact maximum positive range of the signal.
- The computed number-of-bits are promoted to the smallest allowable value that is greater than or equal to the computation. If none exists, then the block returns an error.

Ports

Input

`Ref1` — First reference signal

scalar | vector | matrix | N-D array

First reference signal, from which to gather data type and scaling information.

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

`Ref2` — Second reference signal

scalar | vector | matrix | N-D array

Second reference signal from which to gather data type and scaling information.

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

`Prop` — Propagated data type and scaling

data type and scaling

Data type and scaling information, back-propagated to the model. After the block gathers data type and scaling information from the reference signals, you can apply a second level of adjustments to the data type and scaling. To do so, specify individual multiplicative and additive adjustments in the block dialog box.

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

Parameters

Propagated Type

`1. Propagated data type` — Mode of specifying propagated data type

`Inherit via propagation rule` (default) | `Specify via dialog`

Specify whether to propagate the data type via the dialog box, or inherit the data type from the reference signals.

Dependencies

Setting this parameter to `Specify via dialog` enables the **1.1. Propagated data type (e.g. fixdt(1,16), fixdt('single'))**.

Programmatic Use
Block Parameter: `PropDataTypeMode`
Type: character vector
Values: `'Specify via dialog' | 'Inherit via propagation rule'`
Default: `'Inherit via propagation rule'`

`1.1. Propagated data type (e.g. fixdt(1,16), fixdt('single'))` — Propagated data type

`fixdt(1,16)` (default) | data type string

Specify the data type to propagate.

Dependencies

To enable this parameter, set **1. Propagated data type** to `Specify via dialog`.

Programmatic Use
Block Parameter: `PropDataTypeMode`
Type: character vector
Values: `'Specify via dialog' | 'Inherit via propagation rule'`
Default: `'Inherit via propagation rule'`

1.1 If any reference input is double, output is — Output data type when a reference input is double

double (default) | single

Specify the output data type as single or double. This parameter makes it easier to create designs that are easily retargeted between fixed-point chips and floating-point chips.

Dependencies

To enable this parameter, set **Propagated data type** to Inherit via propagation rule.

Programmatic Use

Block Parameter: IfRefDouble
Type: character vector
Values: 'double' | 'single'
Default: 'double'

1.2 If any reference input is single, output is — Output data type when a reference input is single

single (default) | double

Specify the output data type as single or double. This parameter makes it easier to create designs that are easily retargeted between fixed-point chips and floating-point chips.

Dependencies

To enable this parameter, set **Propagated data type** to Inherit via propagation rule.

Programmatic Use

Block Parameter: IfRefSingle
Type: character vector
Values: 'double' | 'single'
Default: 'single'

1.3 Is-Signed — Signedness of propagated data type

IsSigned1 or IsSigned2 (default) | IsSigned1 | IsSigned2 | TRUE | FALSE

Specify the sign of **Prop** as one of the following values.

Parameter Value	Description
IsSigned1	**Prop** is a signed data type if **Ref1** is a signed data type.
IsSigned2	**Prop** is a signed data type if **Ref2** is a signed data type.
IsSigned1 or IsSigned2	**Prop** is a signed data type if either **Ref1** or **Ref2** are signed data types.
TRUE	**Ref1** and **Ref2** are ignored, and Prop is always a signed data type.
FALSE	**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 **IsSigned** parameter is `IsSigned1 or IsSigned2`, then **Prop** is forced to be a signed data type.

Dependencies

To enable this parameter, set **Propagated data type** to `Inherit via propagation rule`.

Programmatic Use

Block Parameter: `IsSigned`
Type: character vector
Values: `'IsSigned1' | 'IsSigned2' | 'IsSigned1 or IsSigned2' | 'TRUE' | 'FALSE'`
Default: `'IsSigned1 or IsSigned2'`

1.4.1 `Number-of-bits: Base` — Number of bits for the base of the propagated data type

`max([NumBits1 NumBits2])` (default) | `NumBits1` | `NumBits2` | `min([NumBits1 NumBits2])` | `NumBits1+NumBits2`

Specify the number of bits used by **Prop** for the base data type as one of the following values.

Parameter Value	Description
NumBits1	The number of bits for **Prop** is given by the number of bits for **Ref1**.

Parameter Value	Description
NumBits2	The number of bits for **Prop** is given by the number of bits for **Ref2**.
max([NumBits1 NumBits2])	The number of bits for **Prop** is given by the reference signal with largest number of bits.
min([NumBits1 NumBits2])	The number of bits for **Prop** is given by the reference signal with smallest number of bits.
NumBits1+NumBits2	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).

Dependencies

To enable this parameter, set **Propagated data type** to `Inherit via propagation rule`.

Programmatic Use
Block Parameter: `NumBitsBase`
Type: character vector
Values: `'NumBits1' | 'NumBits2' | 'max([NumBits1 NumBits2])' | 'min([NumBits1 NumBits2])' | 'NumBits1+NumBits2'`
Default: `'max([NumBits1 NumBits2])'`

1.4.2 Number-of-bits: Multiplicative adjustment — Number of bits for multiplicative adjustment of propagated data type

1 (default) | positive integer

Specify the number of bits used by **Prop** by including a multiplicative adjustment that uses a data type of `double`. For example, suppose that 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, set this parameter to 2.

Dependencies

To enable this parameter, set **Propagated data type** to `Inherit via propagation rule`.

Programmatic Use
Block Parameter: `NumBitsMult`

Type: character vector
Values: positive integer
Default: `'1'`

1.4.3 `Number-of-bits: Additive adjustment` — Number of bits for additive adjustment of propagated data type

`0` (default) | positive integer

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 could overflow. To prevent overflow, you can associate guard bits with the propagated data type. To associate four guard bits, you specify the value `4`.

Dependencies

To enable this parameter, set **Propagated data type** to `Inherit via propagation rule`.

Programmatic Use
Block Parameter: `NumBitsAdd`
Type: character vector
Values: scalar
Default: `'0'`

1.4.4 `Number-of-bits: Allowable final values` — Allowable number of bits in propagated data type

`'1:128'` (default) | scalar or vector of positive integers

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 bits, 16 bits, 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`.

Dependencies

To enable this parameter, set **Propagated data type** to `Inherit via propagation rule`.

Programmatic Use
Block Parameter: `NumBitsAllowFinal`
Type: character vector
Values: scalar or vector of positive integers

Default: '1:128'

Propagated Scaling

2. `Propagated scaling` — Propagated scaling mode

`Inherit via propagation rule` (default) | `Specify via dialog` | `Obtain via best precision`

Choose to propagate the scaling via the dialog box, inherit the scaling from the reference signals, or calculate the scaling to obtain best precision.

Programmatic Use
Block Parameter: `PropScalingMode`
Type: character vector
Values: `Inherit via propagation rule` | `Specify via dialog` | `Obtain via best precision`
Default: `Inherit via propagation rule`

2.1. `Propagated scaling (Slope or [Slope Bias])` — Slope or slope and bias

`2^-10` | `Slope` | `[Slope Bias]`

Specify the scaling as either a slope or a slope and bias.

Dependencies

To enable this parameter, set **Propagated scaling** to `Specify via dialog`.

Programmatic Use
Block Parameter: `PropScaling`
Type: character vector
Values: `Slope` | `[Slope Bias]`
Default: '2^-10'

2.1. `Values used to determine best precision scaling` — Values to constrain precision

`[5 -7]` (default)

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 block selects a scaling such that these values can be represented with no overflow error and minimum quantization error.

Dependencies

To enable this parameter, set **Propagated scaling** to `Obtain via best precision`.

Programmatic Use
Block Parameter: `ValuesUsedBestPrec`
Type: character vector
Values: vector of values
Default: `'[5 -7]'`

2.1.1. Slope: Base — Slope for base of the propagated data type

`min([Slope1 Slope2])` (default) | `Slope1` | `Slope2` | `min([Slope1 Slope2])` | `max([Bias1 Bias2])` | `Slope1*Slope2` | `Slope1/Slope2` | `PosRange1` | `PosRange2` | `max([PosRange1 PosRange2])` | `min([PosRange1 PosRange2])` | `PosRange1*PosRange2` | `PosRange1/PosRange2`

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.

Parameter Value	Description
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. Also, PosRange1 and PosRange2 are one bit higher than the maximum positive range of the associated reference signal.

Dependencies

To enable this parameter, set **Propagated scaling** to Inherit via propagation rule.

Programmatic Use
Block Parameter: SlopeBase
Type: character vector
Values: 'Slope1' | 'Slope2' | 'max([Slope1 Slope2])' | 'min([Slope1 Slope2])' | 'Slope1*Slope2' | 'Slope1/Slope2' | 'PosRange1' | 'PosRange2' | 'max([PosRange1 PosRange2])' | 'min([PosRange1 PosRange2])' | 'PosRange1*PosRange2' | 'PosRange1/PosRange2'
Default: 'min([Slope1 Slope2])'

2.1.2. Slope: Multiplicative adjustment — Slope of multiplicative adjustment of propagated data type

1 (default) | scalar

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.

Dependencies

To enable this parameter, set **Propagated scaling** to Inherit via propagation rule.

Programmatic Use
Block Parameter: SlopeMult
Type: character vector
Values: scalar
Default: '1'

2.1.3. Slope: Additive adjustment — Slope of additive adjustment of propagated data type

0 (default) | scalar

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.

Dependencies

To enable this parameter, set **Propagated scaling** to `Inherit via propagation rule`.

Programmatic Use
Block Parameter: `SlopeAdd`
Type: character vector
Values: scalar
Default: `'0'`

2.2.1. Bias: Base — Base bias for Prop

Bias1 (default) | Bias2 | max([Bias1 Bias2]) | min([Bias1 Bias2]) | Bias1*Bias2 | Bias1/Bias2 | Bias1+Bias2 | Bias1-Bias2

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.
`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.

Parameter Value	Description
`Bias1-Bias2`	The bias of **Prop** is given by the difference of the reference biases.

Dependencies

To enable this parameter, set **Propagated scaling** to `Inherit via propagation rule`.

Programmatic Use
Block Parameter: `BiasBase`
Type: character vector
Values: `'Bias1' | 'Bias2' | 'max([Bias1 Bias2])' | 'min([Bias1 Bias2])' | 'Bias1*Bias2' | 'Bias1/Bias2' | 'Bias1+Bias2' | 'Bias1-Bias2'`
Default: `'Bias1'`

2.2.2. `Bias: Multiplicative adjustment` — Multiplicative bias for propagated data type

`1` (default) | scalar

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`.

Programmatic Use
Block Parameter: `BiasMult`
Type: character vector
Values: scalar
Default: `'0'`

2.3.2. `Bias: Additive adjustment` — Additive bias for propagated data type

`0` (default) | scalar

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, configure both the multiplicative adjustment and the additive adjustment to `0`.

Dependencies

To enable this parameter, set **Propagated scaling** to `Inherit via propagation rule`.

Programmatic Use
Block Parameter: `BiasAdd`
Type: character vector
Values: scalar
Default: `'0'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Type Conversion | Data Type Conversion Inherited | Data Type Duplicate

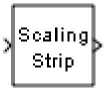
Topics

"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: Simulink / Signal Attributes



Description

The Data Type 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 distinguish between real world value and stored integer value.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The block strips the scaling off a fixed-point input signal, and outputs the stored integer value with the smallest possible built-in data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Input signal mapped to built-in data type
scalar | vector | matrix

Stored integer value of the input signal with the smallest possible built-in data type, and the same dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Data Type Conversion | Data Type Duplicate | Data Type Propagation

Topics

“About Data Types in Simulink”

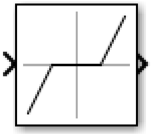
“Fixed Point”

Introduced before R2006a

Dead Zone

Provide region of zero output

Library: Simulink / 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. The block output depends on the input (U) and the values for the lower and upper limits.

Input	Output
U >= LL and U <= UL	Zero
U > UL	U – UL
U < LL	U – LL

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal to the dead-zone algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

`Port_1` — Output signal

scalar | vector

Output signal after the dead-zone algorithm is applied to the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Start of dead zone` — Specify the lower bound of the dead zone

`'-0.5'` (default) | scalar | vector

Specify dead zone lower limit. Set the value for **Start of dead zone** less than or equal to **End of dead zone**. When the input value is less than **Start of dead zone**, then the block shifts the output value down by the **Start of dead zone** value.

Programmatic Use
Block Parameter: `LowerValue`
Type: character vector
Value: scalar or vector less than or equal to `UpperValue`.
Default: `'-0.5'`

`End of dead zone` — Specify the upper limit of the dead zone

`'0.5'` (default) | scalar | vector

Specify dead zone upper limit. Set the value for **End of dead zone** greater than or equal to **Start of dead zone**. When the input value is greater than **End of dead zone**, then the block shifts the output value down by the **End of dead zone** value.

Programmatic Use
Block Parameter: `UpperValue`
Type: character vector
Value: scalar or vector greater than or equal to `LowerValue`.
Default: `'0.5'`

`Saturate on integer overflow` — Choose the behavior when integer overflow occurs

`on` (default) | boolean

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 `int8` (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.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'on'`

`Treat as gain when linearizing` — Specify the gain value

`On` (default) | boolean

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. Clear the box to have the commands treat the gain as 0.

Programmatic Use
Block Parameter: `LinearizeAsGain`
Type: character vector
Value: `'off'` | `'on'`
Default: `'on'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

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™.

For more information on HDL code generation, see Dead Zone.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

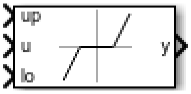
See Also

Backlash | Dead Zone Dynamic

Introduced before R2006a

Dead Zone Dynamic

Provide dynamic region of zero output
Library: Simulink / Discontinuities



Description

The Dead Zone Dynamic block generates a region of zero output based on dynamic input signals that specify the upper and lower limit. The block output depends on the input **u**, and the values of the input signals **up** and **lo**.

Input	Output
`u >= lo` and `u <= up`	Zero
`u > up`	`u` – `up`
`u < lo`	`u` – `lo`

The Dead Zone Dynamic block is a masked subsystem and does not have any parameters.

Ports

Input

u — Input signal
scalar | vector

Input signal to the dead zone algorithm.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`lo` — Lower limit for the dead zone
scalar | vector

Dynamic value providing the lower bound of the region of zero output. When the input is less than **lo** then the output value is shifted down by value of **lo**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

up — Upper limit for the dead zone
scalar

Dynamic value providing the upper bound of the region of zero output. When the input is greater than **up** then the output value is shifted down by value of **up**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

y — Output signal
scalar | vector

Output signal after the dynamic dead zone algorithm is applied to the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
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™.

For more information on HDL code generation, see Dead Zone Dynamic.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

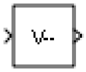
Dead Zone | Triggered Subsystem

Introduced before R2006a

Decrement Real World

Decrease real-world value of signal by one

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal

scalar | vector | matrix

Output is the real-world value of the input signal decreased by one. Overflows always wrap. The output has the same data type and dimensions as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Block Characteristics

Data Types	double | single | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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™.

For more information about HDL code generation, see Decrement Real World.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Decrement Stored Integer | Decrement Time To Zero | Decrement To Zero | Increment Real World

Topics

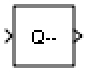
“Fixed-Point Numbers”

Introduced before R2006a

Decrement Stored Integer

Decrease stored integer value of signal by one

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output is the stored integer value of the input signal decreased by one. Floating-point signals also decrease by one, and overflows always wrap. The output has the same data type and dimensions as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

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™.

For more information about HDL code generation, see Decrement Stored Integer.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Decrement Real World | Decrement Time To Zero | Decrement To Zero | Increment Stored Integer

Topics

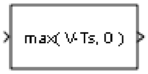
"Fixed-Point Numbers"

Introduced before R2006a

Decrement Time To Zero

Decrease real-world value of signal by sample time, but only to zero

Library: Simulink / 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, `Ts`. The output never goes below zero.

Limitations

The Decrement Time To Zero block works only with fixed sample rates and does not work inside a triggered subsystem.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output is the real-world value of the input signal decreased by the sample time, `Ts`. The output never goes below zero. The output has the same data type and dimensions as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Decrement Real World | Decrement Stored Integer | Decrement To Zero

Topics

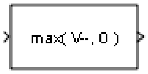
"Fixed-Point Numbers"

Introduced before R2006a

Decrement To Zero

Decrease real-world value of signal by one, but only to zero

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output is the real-world value of the input signal decreased by one. The output never goes below zero. The output has the same data type and dimensions as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Decrement Real World | Decrement Stored Integer | Decrement Time To Zero

Topics

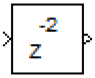
"Fixed-Point Numbers"

Introduced before R2006a

Delay

Delay input signal by fixed or variable sample periods

Library: Simulink / Commonly Used Blocks
Simulink / 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 several factors such as the **Initial condition** parameter and the simulation start time. For more information, see “Initial Block Output” on page 1-340. 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 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”.

The table shows the Delay block output for the first few time steps with these settings. The block inherits a discrete sample time as `[*Tsampling*,*Toffset*]`, where *`Tsampling`* is the sampling period and *`Toffset`* is the initial time offset. *n* is the value of the **Delay length** parameter and *`Tstart`* is the simulation start time for the model

Simulation Time Range	Block Output
(*Tstart*) to (*Tstart + Toffset*)	Zero
(*Tstart + Toffset*) to (*Tstart + Toffset + n * Tsampling*)	**Initial condition** parameter
After (*Tstart + Toffset + n * Tsampling*)	Input signal

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)`	• The signal dimensions change only during state reset when the block is enabled. • The initial condition must be scalar.
`Columns as channels (frame based)`	• No support
`Inherited` (where input is a sample-based signal)	• 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)	• 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, specify the initial condition on the dialog box. The initial condition cannot come from the input port x0. 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

Ports

Input

u — Data input signal

scalar | vector

Input data signal delayed according to parameters settings.

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

d — Delay length

scalar

Delay length specified as inherited from an input port. Enabled when you select the **Delay length: Source** parameter as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Enable` — External enable signal

scalar

Enable signal that enables or disables execution of the block. To create this port, select the **Show enable port** parameter.

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

`External reset` — External reset signal

scalar

External signal that resets execution of the block to the initial condition. To create this port, select the **External reset** parameter.

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

`x0` — Initial condition

scalar | vector

Initial condition specified as inherited from an input port. Enabled when you select the **Initial Condition: Source** parameter as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal

scalar | vector

Output signal that is the input signal delayed by the length of time specified by the parameter **Delay length**. The initial value of the output signal depends on several conditions. See “Initial Block Output” on page 1-340.

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

Parameters

Main

`Delay length` — Delay length

`Dialog` (default) | `Input port`

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 noninteger value from an input port (variable delay) truncates it to the integer.

Programmatic Use
Block Parameter: `DelayLengthSource`
Type: character vector
Values: `'Dialog'` | `'Input port'`
Default: `'Dialog'`
Block Parameter: `DelayLength`
Type: character vector
Values: scalar
Default: `'2'`
Block Parameter: `DelayLengthUpperLimit`
Type: character vector
Values: scalar

Default: `'100'`

`Initial condition` — Initial condition

`Dialog` (default) | `Input port`

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.

Programmatic Use

Block Parameter: `InitialConditionSource`
Type: character vector
Values: `'Dialog'` | `'Input port'`
Default: `'Dialog'`
Block Parameter: `InitialCondition`
Type: character vector
Values: scalar
Default: `'0.0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use

Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Use circular buffer for state` — Circular buffer for storing state

`off` (default) | `on`

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.

Programmatic Use

Block Parameter: `UseCircularBuffer`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Prevent direct feedthrough` — Prevent direct feedthrough

`off` (default) | `on`

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.

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use

Block Parameter: `PreventDirectFeedthrough`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Remove delay length check in generated code` — Remove delay length out-of-range check

`off` (default) | `on`

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

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use

Block Parameter: `RemoveDelayLengthCheckInGeneratedCode`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Diagnostic for delay length` — Diagnostic checks for delay length

`None` (default) | `Warning` | `Error`

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.

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use
Block Parameter: `DiagnosticForDelayLength`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

`Show enable port` — Create enable port

`off` (default) | `on`

Select to control execution of this block with an enable port. 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.

Programmatic Use
Block Parameter: `ShowEnablePort`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`External reset` — External state reset

`None` (default) | `Rising` | `Falling` | `Either` | `Level` | `Level hold`

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: • 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

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector
Values: `'None'` | `'Rising'` | `'Falling'` | `'Either'` | `'Level'` | `'Level hold'`
Default: `'None'`

`Sample time (-1 for inherited)` — Discrete interval between sample time hits

`-1` (default) | `scalar`

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.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Value: real scalar
Default: `'-1'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: `StateName`
Type: character vector

Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, specify a value for **State name**. 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**.

Programmatic Use

Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default) | `<StorageClass.PackageName>`

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use

Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`

Default: `'Simulink.Signal'`

`Code generation storage class` — State storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `BitField (Custom)` | `Model default` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `AutoScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation.

- `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 "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder). For information about custom storage classes, see "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

To enable this parameter, specify a value for **State name**.

Programmatic Use

Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto' | 'SimulinkGlobal' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`Yes`

Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Consider using the Model Discretizer to map these continuous blocks into discrete equivalents that support code generation. From a model, select **Analysis** > **Control Design** > **Model Discretizer**.

Not recommended for production code.

HDL Code Generation

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

For more information about HDL code generation, see Delay.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Resettable Delay | Tapped Delay | Unit Delay | Variable Integer Delay

Topics

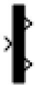
“Using Enabled Subsystems”

Introduced before R2006a

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

Vector input signal from which the Demux block selects scalar signals or smaller vectors.

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 mod 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 mod 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 `...` p_m`]` $p_1+p_2+...+p_m=n$ $p_i > 0$	m vector signals having widths p_1, p_2, ... 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 `...` p_m`]` $p_1+p_2+...+p_m!=n$ $p_i = > 0$	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 examples, see “Extract Vector Elements and Distribute Evenly Across Outputs” on page 14-171 and “Extract Vector Elements Using the Demux Block” on page 14-172.

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`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`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™.

This block has a single, default HDL architecture. See Demux.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

See Also

Bus Creator | Bus to Vector | Mux

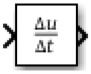
Topics

“Virtual Signals”
“Simplify Subsystem Bus Interfaces”
“Composite Signals”
feedbacksystem

Introduced before R2006a

Derivative

Output time derivative of input
Library: Simulink / 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:

$$y(t) = \frac{\Delta u}{\Delta t} = \left.\frac{u(t) - u(T_{previous})}{t - T_{previous}}\right| t > T_{previous},$$

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 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 for 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 the step size and improve simulation accuracy. 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 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.

If 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:

$$y(k) = \frac{1}{\Delta t}(u(k) - u(k-1))$$

.

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.

Ports

Input

`Port_1` — Input signal

real scalar or vector

Signal to be differentiated, specified as a real scalar or vector.

Data Types: `double`

Output

`Port_1` — Time derivative of input signal
real scalar or vector

Time derivative of input signal, specified as a real scalar or vector. The input signal is differentiated with respect to time as:

$$y(t) = \frac{\Delta u}{\Delta t} = \left.\frac{u(t) - u(T_{previous})}{t - T_{previous}}\right| t > T_{previous},$$

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.

Data Types: `double`

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
`inf` (default)

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 block 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`.

Tips

- As a best practice, change the value of c to $\frac{1}{f_b}$, where f_b is the break frequency of the filter.
- The parameter must be a finite positive value.

Programmatic Use
Block Parameter: `CoefficientInTFapproximation`
Type: character vector, string
Values: `'inf'`
Default: `'inf'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Consider using the Model Discretizer to map the continuous blocks into discrete equivalents that support code generation. From a model, select **Analysis** > **Control Design** > **Model Discretizer** to access the Model Discretizer

Not recommended for production code.

See Also

Discrete Derivative

Topics

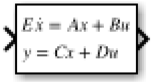
“Improved Linearization with Transfer Fcn Blocks” on page 14-24

Introduced before R2006a

Descriptor State-Space

Model linear implicit systems

Library: Simulink / Continuous



Description

The Descriptor State-Space block allows you to model linear implicit systems that can be expressed in the form $E\dot{x} = Ax + Bu$ where *E* is the mass matrix of the system. When *E* is nonsingular and therefore invertible, the system can be written in its explicit form $\dot{x} = E^{-1}Ax + E^{-1}Bu$ and modeled using the State-Space block.

When the mass matrix *E* is singular, one or more derivatives of the dependent variables of the system are not present in the equations. These variables are called algebraic variables. Differential equations that contain such algebraic variables are called differential algebraic equations. Their state space representation is of the form

$$\begin{aligned} E\dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

where the variables have the following meanings:

- *x* is the state vector
- *u* is the input vector
- *y* is the output vector

Ports

Input

`Input 1` — Input signal

scalar | vector

Real-valued input vector of type `double` whose width is the number of columns in the **B** and **D** matrices.

Data Types: `double`

Output

`Output 1` — Output vector

scalar | vector

Real-valued input vector of type `double` whose width is the number of rows in the **C** and **D** matrices.

Data Types: `double`

Parameters

E — Mass matrix

1 (default) | scalar | matrix | sparse matrix

Specify the mass matrix *E* as a real-valued *n*-by-*n* matrix, where *n* is the number of states in the system. **E** must be the same size as **A**. **E** can be singular or non-singular.

Programmatic Use
Block Parameter: `E`
Type: character vector, string
Values: scalar | matrix
Default: `'1'`

A — Matrix coefficient, A

1 (default) | scalar | matrix | sparse matrix

Specify the matrix coefficient *A* as a real-valued *n*-by-*n* matrix, where *n* is the number of states in the system. **A** must be the same size as **E**.

Programmatic Use
Block Parameter: `A`
Type: character vector, string
Values: scalar | matrix
Default: `'1'`

`B` — Matrix coefficient, B

1 (default) | scalar | vector | matrix | sparse matrix

Specify the matrix coefficient *B* as a real-valued *n*-by-*m* matrix, where *n* is the number of states in the system and *m* is the number of inputs.

Programmatic Use
Block Parameter: `B`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'1'`

`C` — Matrix coefficient, C

1 (default) | scalar | vector | matrix | sparse matrix

Specify the matrix coefficient *C* as a real-valued *r*-by-*n* matrix, where *n* is the number of states in the system and *r* is the number of outputs.

Programmatic Use
Block Parameter: `C`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'1'`

`D` — Matrix coefficient, D

1 (default) | scalar | vector | matrix | sparse matrix

Specify the matrix coefficient *D* as a real-valued *r*-by-*m* matrix, where *r* is the number of outputs of the system and *m* is the number of inputs to the system.

Programmatic Use
Block Parameter: `D`
Type: character vector, string
Values: scalar | vector | matrix

Default: `'1'`

`Initial conditions` — Initial state vector

`0` (default) | scalar | vector

Specify the initial state vector.

Limitations

The initial conditions of this block cannot be `inf` or `NaN`.

Programmatic Use

Block Parameter: `X0`
Type: character vector
Values: scalar | vector
Default: `'0'`

`Direct feedthrough` — Set output signal dependency on input

true (default) | false

Specify whether the output of the block directly depends on the input signal. Use this parameter for systems having more than 500 continuous states in order to speed up simulation. For systems with 500 continuous states or less, Simulink automatically determines this setting.

Programmatic Use

Block Parameter: `DirectFeedthrough`
Type: character vector, string
Values: 'True' | 'False'
Default: `'True'`

`Absolute tolerance` — Absolute tolerance for computing block states

`auto` (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify `auto` or `-1`.

- 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.

- 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.

Programmatic Use
Block Parameter: `AbsoluteTolerance`
Type: character vector, string
Values: `'auto'` | `'-1'` | any positive real-valued scalar or vector
Default: `'auto'`

`State Name (e.g., 'position')` — Assign unique name to each state

`' '` (default) | `'position'` | `{'a', 'b', 'c'}` | `a` | ...

Assign a unique name to each state. If this field is blank (`' '`), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'position'`.
- 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 in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

Limitations

- 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.

Programmatic Use
Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

1. Consider using the Model Discretizer to map these continuous blocks into discrete equivalents that support code generation. From a model, select **Analysis** > **Control Design** > **Model Discretizer** to access the Model Discretizer.
2. Not recommended for production code.

See Also

Blocks

Algebraic Constraint | State-Space

Functions

`dss`

Topics

“Solve Differential Algebraic Equations (DAEs)” (MATLAB)

"Model Differential Algebraic Equations"

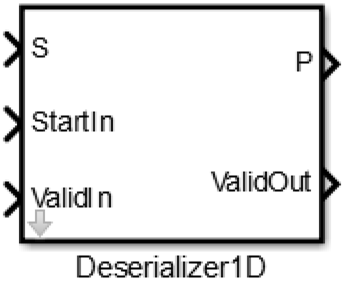
External Websites

https://www.mathworks.com/matlabcentral/fileexchange/7481-manuscript-of--solving-index-1-daes-in-matlab-and-simulink-

Introduced in R2018b

Deserializer1D

Convert scalar stream or smaller vectors to vector signal



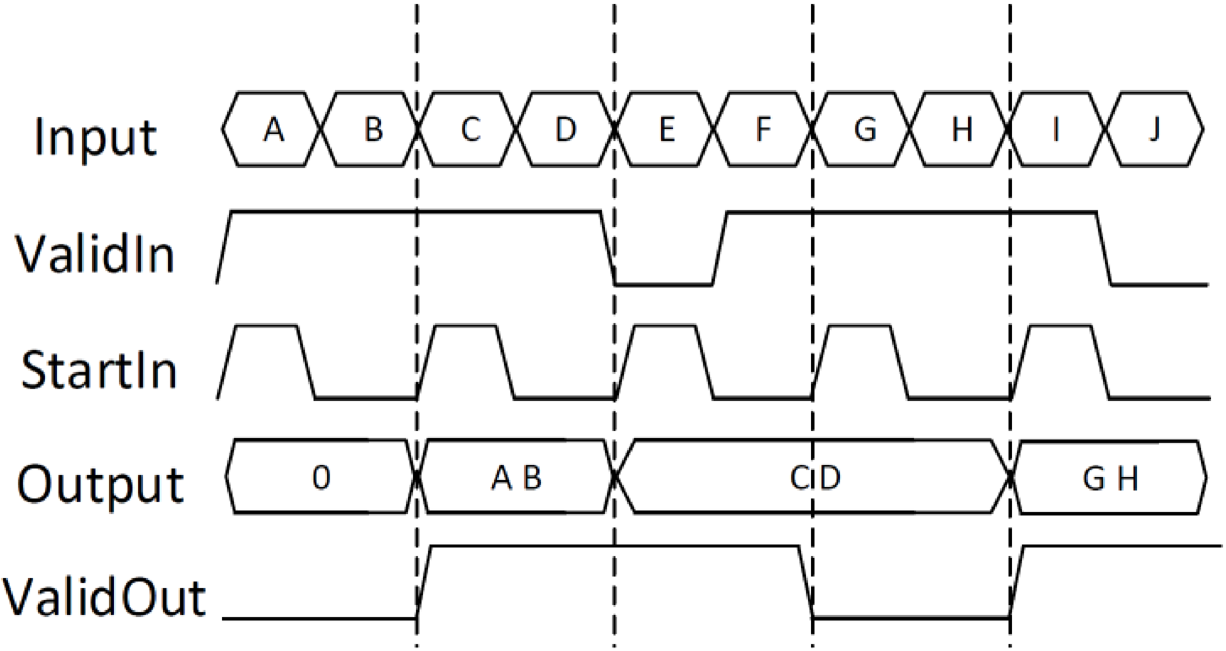
Library

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:

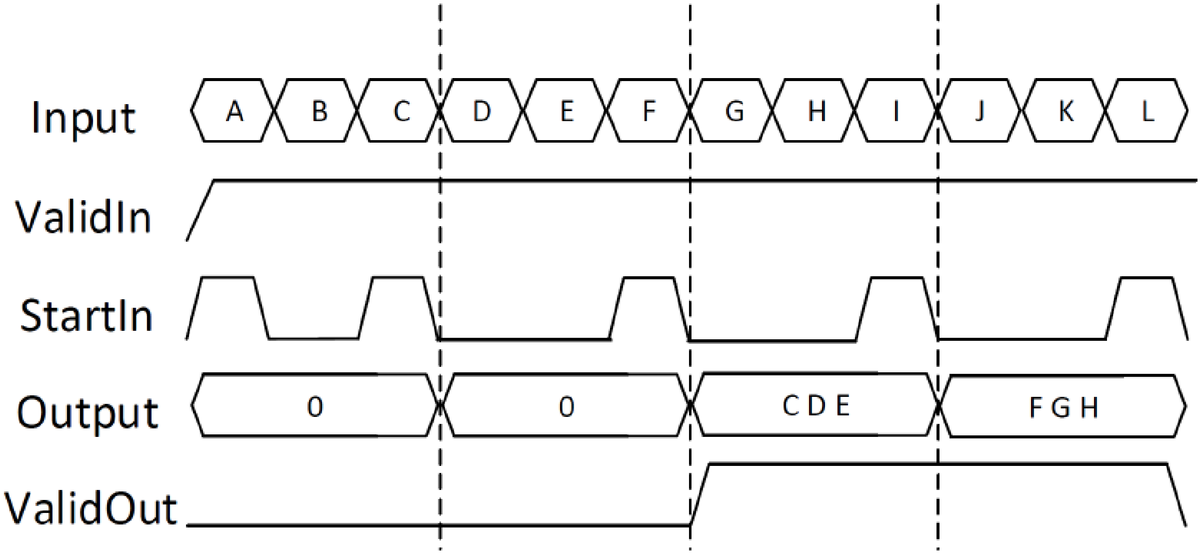


- **Ratio** is 2 and **Idle Cycles** is 0, so each output cycle is two input signals long with all data points considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

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 Serializer1D block. This port is available when you select the **ValidIn** check box.

Data type: Boolean

`StartOut`

Indicates where to start deserialization. Use with the Serializer1D 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

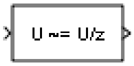
See Also

Serializer1D

Introduced in R2014b

Detect Change

Detect change in signal value
Library: Simulink / Logic and Bit Operations



Description

The Detect Change block determines if an input signal does not equal its previous value. The initial condition determines the initial value of the previous input U/z.

Ports

Input

`Port_1` — Input signal
signal value

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `built-in integer` | `floating point`

Output

`Port_1` — Output signal
0 | 1

Output signal, true (equal to 1) when the input signal does not equal its previous value; false (equal to 0) when the input signal equals its previous value.

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

Parameters

`Initial condition` — Initial condition for the previous input

0 (default) | scalar | vector

Set the initial condition for the previous input U/z.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar | vector
Default:`'0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Output data type` — Data type of the output

`boolean` (default) | `uint8`

Set the output data type to `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean'` | `'uint8'`
Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`

Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

HDL Code Generation

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

For information on HDL code generation support, see Detect Change.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

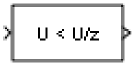
Detect Decrease | Detect Fall Negative | Detect Fall Nonpositive | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a

Detect Decrease

Detect decrease in signal value

Library: Simulink / Logic and Bit Operations



Description

The Detect Decrease block determines if an input is strictly less than its previous value.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

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

Output

`Port_1` — Output signal

0 | 1

Output signal, true (equal to 1) when the input signal is less than its previous value; false (equal to 0) when the input signal is greater than or equal to its previous value.

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

Parameters

Initial condition — Initial condition for the previous input

0 (default) | scalar | vector

Set the initial condition for the previous input U/z.

Programmatic Use
Block Parameter: vinit
Type: character vector
Values: scalar | vector
Default: '0'

Input processing — Specify sample- or frame-based processing

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Output data type` — Data type of the output
`boolean` (default) | `uint8`

Set the output data type to `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean'` | `'uint8'`
Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`

Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

HDL Code Generation

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

For information on HDL code generation support, see Detect Decrease.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

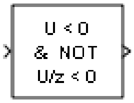
Detect Change | Detect Fall Negative | Detect Fall Nonpositive | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a

Detect Fall Negative

Detect falling edge when signal value decreases to strictly negative value, and its previous value was nonnegative

Library: Simulink / Logic and Bit Operations



Description

The Detect Fall Negative block determines if the input is less than zero, and its previous value is greater than or equal to zero.

Ports

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

Input

`Port_1` — Input signal
signal value

Input signal, specified as a scalar, vector, or matrix.

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

Output

`Port_1` — Output signal
0 | 1

Output signal, true (equal to 1) when the input signal is less than zero, and its previous value was greater than or equal to zero; 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 Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

`Initial condition` — Initial condition for the previous input

0 (default) | scalar | vector

Set the initial condition of the Boolean expression U/z < 0.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar | vector
Default:`'0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Output data type` — Data type of the output

`boolean` (default) | `uint8`

Set the output data type to `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean'` | `'uint8'`
Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`

Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

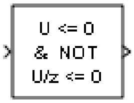
Detect Change | Detect Decrease | Detect Fall Nonpositive | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a

Detect Fall Nonpositive

Detect falling edge when signal value decreases to nonpositive value, and its previous value was strictly positive

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal that detects a falling edge, specified as a scalar, vector, or matrix.

- 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 Types: `uint8` | `Boolean`

Parameters

`Initial condition` — Initial condition of Boolean expression `U/z <= 0`

0 (default) | scalar | vector | matrix

Set the initial condition of the Boolean expression `U/z <= 0`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Output data type` — Output data type

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean' | 'uint8'`
Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

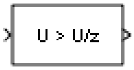
Detect Change | Detect Decrease | Detect Fall Negative | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a

Detect Increase

Detect increase in signal value

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

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

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal, detecting an increase in signal value, specified as a scalar, vector, or matrix.

- 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 Types: `uint8` | `Boolean`

Parameters

`Initial condition` — Initial condition of previous input

`0.0` (default) | scalar | vector | matrix

Set the initial condition for the previous input `U/z`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar | vector | matrix
Default: `'0.0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Output data type` — Output data type

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean' | 'uint8'`
Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`

Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

HDL Code Generation

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

For information on HDL code generation support, see Detect Increase.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

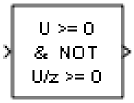
Detect Change | Detect Decrease | Detect Fall Negative | Detect Fall Nonpositive | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a

Detect Rise Nonnegative

Detect rising edge when signal value increases to nonnegative value, and its previous value was strictly negative

Library: Simulink / Logic and Bit Operations



Description

The Detect Rise Nonnegative block determining 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal that indicates a rising edge whenever the signal value increases to a nonnegative value, and its previous value was strictly negative. The output can be a scalar, vector, or matrix.

- 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 Types: `uint8` | `Boolean`

Parameters

`Initial condition` — Initial condition of Boolean expression `U/z >= 0`

`0` (default) | scalar | vector | matrix

Set the initial condition of the Boolean expression `U/z >= 0`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Output data type` — Output data type

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean' | 'uint8'`
Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

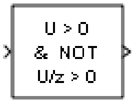
Detect Change | Detect Decrease | Detect Fall Negative | Detect Fall Nonpositive | Detect Increase | Detect Rise Positive

Introduced before R2006a

Detect Rise Positive

Detect rising edge when signal value increases to strictly positive value, and its previous value was nonpositive

Library: Simulink / Logic and Bit Operations



Description

The Detect Rise Positive block detects a rising edge by determining 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

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

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal that detects a rising edge whenever the input is strictly positive, and its previous value was nonpositive. The output can be a scalar, vector, or matrix.

- 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 Types: `uint8` | `Boolean`

Parameters

`Initial condition` — Initial condition of Boolean expression `U/z > 0`
`0` (default) | scalar | vector | matrix

Set the initial condition of the Boolean expression `U/z > 0`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`Input processing` — Specify sample- or frame-based processing
`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Output data type` — Output data type

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean' | 'uint8'`
Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

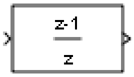
See Also

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: Simulink / Discrete



Description

The Difference block outputs the current input value minus the previous input value.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.

Dependencies

When you set **Input processing** to `Columns as channels (frame based)`, the input signal must have two dimensions or less.

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

Output

`Port_1` — Current input minus previous input
scalar | vector | matrix | N-D array

Current input minus previous input, specified as a scalar, vector, matrix, or N-D array.

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

Parameters

Main

`Initial condition for previous input` — Initial condition

`0.0` (default) | scalar | vector | matrix | N-D array

Set the initial condition for the previous input.

Programmatic Use
Parameter: `ICPrevInput`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'0.0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

Signal Attributes

`Output minimum` — Minimum output value for range checking

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[ ]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Output data type

`Inherit: Inherit via internal rule` (default) | `Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

Programmatic Use
Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | '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)' | '<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	double | single | Boolean[a] | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	Yes

Zero-Crossing Detection	No

a. This block is not recommended for use with Boolean signals.

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.
- Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

`diff`

Topics

“Sample- and Frame-Based Concepts” (DSP System Toolbox)

Introduced before R2006a

Digital Clock

Output simulation time at specified sampling interval
Library: Simulink / 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, use 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.

Ports

Output

`Port_1` — Sample time
scalar

Sample time, in seconds, at the specified sampling interval. At other times, the block holds the output at the previous value.

Data Types: `double`

Parameters

`Sample time` — Sampling interval
`1` (default) | scalar | vector

Specify the sampling interval in seconds. You can specify the sampling interval in one of two ways:

- As the period, specified as a real-valued scalar with data type double.
- As the period and offset, specified as a real-valued vector of length 2 with data type double. The period and offset must be finite and non-negative, and the offset value must be less than the period.

For more information, see Specifying Sample Time.

Tip 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.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'1'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

See Also

Clock

Topics

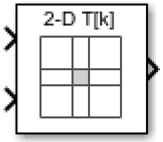
“Sample Time”

Introduced before R2006a

Direct Lookup Table (n-D)

Index into n-dimensional table to retrieve element, vector, or 2-D matrix

Library: Simulink / Lookup Tables



Description

The Direct Lookup Table (n-D) block indexes into an n-dimensional table to retrieve an element, vector, or 2-D matrix. The first selection index corresponds to the top (or left) input port. You can choose to provide the table data as an input to the block, or define the table data on the block dialog box. The number of input ports and the size of the output depend on the number of table dimensions and the output slice you select.

If you select a vector from a 2-D table, the output vector can be a column or a row, depending on the model configuration parameter setting **Math and Data Types > Use algorithms optimized for row-major array layout**. The block inputs are zero-based indices (for more information, see the **Inputs select this object from table** parameter.

The Direct Lookup Table block supports symbolic dimensions.

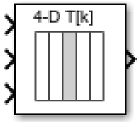
Block Inputs and Outputs

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, vector, or 2-D matrix.

You define a set of output values as the **Table data** parameter. For the default column-major algorithm behavior, 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
Vector	1	2
Matrix	2	3

Suppose that you want to select a vector of values from a 4-D table.



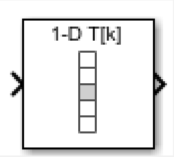
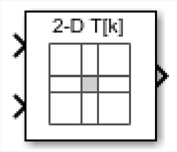
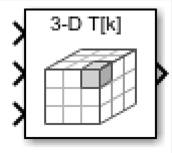
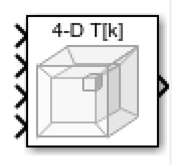
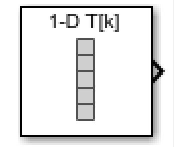
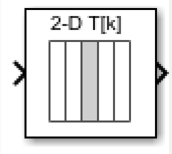
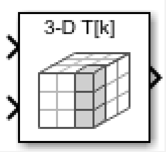
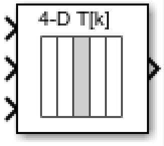
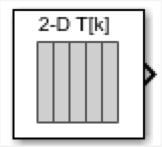
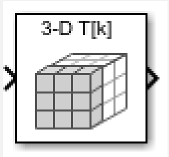
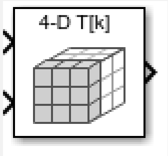
The following mapping of block input port to table dimension applies.

This input port...	Is the index for this table dimension...
1	2
2	3
3	4

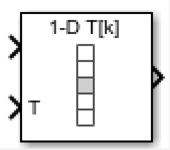
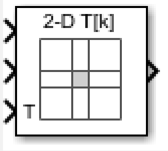
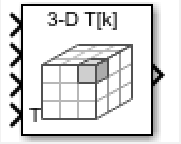
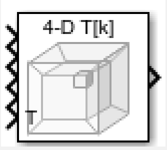
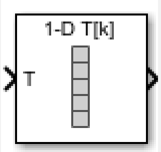
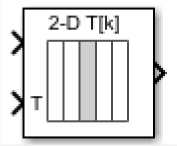
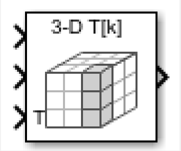
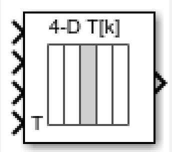
Changes in Block Icon Appearance

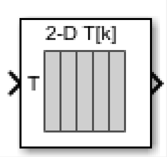
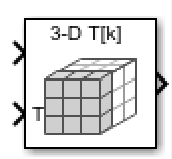
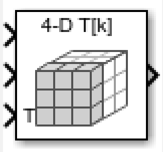
Depending on parameters you set, the block icon changes appearance. For table dimensions higher than 4, the icon matches the 4-D version but shows the exact number of dimensions at the top.

When you use the **Table data** parameter, you see these icons for the default column-major behavior. Some icons are different when you select the configuration parameter **Math and Data Types > Use algorithms optimized for row-major array layout**.

Object that Inputs Select from the Table	Number of Table Dimensions			
	1	2	3	4
Element				
Vector				
2-D Matrix	Not applicable			

When you use the table input port, you see these icons.

Object that Inputs Select from the Table	Number of Table Dimensions			
	1	2	3	4
Element				
Vector				

Object that Inputs Select from the Table	**Number of Table Dimensions**			
	1	**2**	**3**	**4**
2-D Matrix	Not applicable			

Ports

Input

`Port_1` — Index `i1` input values

scalar | vector

For the default column-major algorithm, the first input port, specifying the zero-based index to the table dimension that is one higher than the output dimensionality (0, 1, or 2). The next input specifies the zero-based index to the next table dimension, and so on. All index inputs must be real-valued.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `enumerated`

`Port_N` — Index `N` input values

scalar | vector

For the default column-major algorithm, the N-th input port, specifying the zero-based index to the table dimension that is N higher than the output dimensionality (0, 1, or 2). The number of inputs varies with the shape of the output. All index inputs must be real-valued.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `enumerated`

`T` — Table data

vector | matrix | N-D array

Table data, specified as a vector, matrix, or N-D array. The table size must match the dimensions of the **Number of dimensions** parameter. The block's output data type is the same as the table data type.

Dependencies

To enable this port, select the **Make table an input** check box.

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

Output

`Port_1` — Output element, vector, or 2-D matrix

scalar | vector | 2-D matrix

Output slice, provided as a scalar, vector, or 2-D matrix. The size of the block output is determined by the setting of the **Inputs select this object from table** parameter. The output data type is the same as the table data type.

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

Parameters

Main

Table

`Number of table dimensions` — Number of dimensions of table data

2 (default) | 1 | 3 | 4

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.

To specify...	Do this...
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.

Programmatic Use
Block Parameter: `NumberOfTableDimensions`
Type: character vector
Values: `'1' | '2' | '3' | '4' | ... |'30'|`
Default: `'2'`

`Make table an input` — Provide table data as a block input

`off` (default) | `on`

Select this check box to provide table data to the Direct Lookup Table (n-D) block as a block input. When you select this check box, a new input port, `T`, appears. Use this port to input the table data.

Programmatic Use
Block Parameter: `TableIsInput`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Table data` — Table of output values

`[4 5 6; 16 19 20; 10 18 23]` (default) | `scalar, vector, matrix, or N-D array`

Specify the table of output values. The table size must match the dimensions of the **Number of table dimensions** parameter.

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).

Click **Edit** to open the Lookup Table Editor. For more information, see “Edit Lookup Tables”.

Dependencies

To enable the **Table data** field, clear the **Make table an input** check box.

Programmatic Use
Block Parameter: `Table`
Type: character vector
Values: `scalar, vector, matrix, or N-D array`
Default: `'[4 5 6;16 19 20;10 18 23]'`

Algorithm

`Inputs select this object from table` — Specify whether output is an element, vector, or 2-D matrix

`Element` (default) | `Vector` | `2-D Matrix`

Specify whether the output data is a single element, a vector, 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
`Vector`	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	`array(1,2,3,4)`
A vector	Three indices	`array(:,2,3,4)` (default column-major algorithm)
A 2-D matrix	Two indices	`array(:,:,3,4)` (default column-major algorithm)

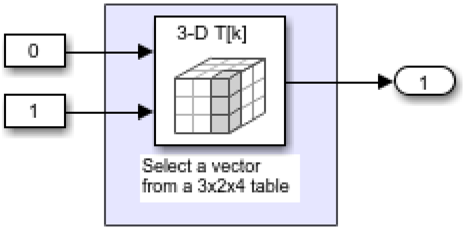
Tips

When the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter is set, the Direct Lookup Table block behavior changes from column-major to row-major. For this block, the column-major and row-major algorithms may differ semantically in output calculations, resulting in different numerical values. This capability requires a Simulink Coder or Embedded Coder license. For example, assume that **Inputs select this object from table** parameter is set to `Vector`. The elements of the selected vector are contiguous in the table storage memory. This table shows the column-major and row-major algorithm depending on the table dimension:

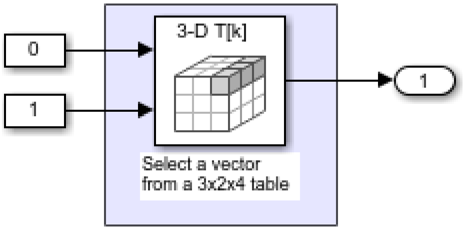
Table Dimension	Column-Major Algorithm	Row-Major Algorithm
2-D table	Column vector is selected	Row vector is selected
3-D and higher table	Output vector is selected from the first dimension of the table	Output vector is selected from the last dimension of the table

Consider the row-major and column-major direct lookup algorithms with vector output from a 3-D table. The last dimension is the third dimension of a 3-D table. Due to semantic changes, column-major and row-major direct lookup may output different vector size and numerical values.

This figure shows a Direct Lookup Table (n-D) block configured with a 3-D table and a vector output. When the model that contains this block is configured for column-major layout, the block icon shows the column-major algorithm.



To have the same block use the row-major algorithm, change the **Use algorithms optimized for row-major array layout** configuration parameter of the model and recompile. The block icon changes to reflect the change to the algorithm optimized for row-major layout.



For more information on row-major support, see “Row-Major Array Layout: Simplify integration with external C/C++ code for Lookup Table and other blocks” (Simulink Coder).

Programmatic Use
Block Parameter: `InputsSelectThisObjectFromTable`
Type: character vector
Values: `'Element' | 'Vector' | '2-D Matrix'`
Default: `'Element'`

`Diagnostic for out-of-range input` — Block action when input is out of range

`Warning` (default) | `None` | `Error`

Specify whether to show a warning or error when an index is out of range with respect to the table dimension. Options include:

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display 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.

Programmatic Use
Block Parameter: `DiagnosticForOutOfRangeInput`
Type: character vector
Values: `'None' | 'Warning' | 'Error'`
Default: `'Warning'`

`Sample time` — Specify sample time as a value other than -1

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Table Attributes

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` — Minimum value table data can have

`[]` (default) | `finite, real, double, scalar`

Specify the minimum value for table data. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `TableMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Table maximum` — Maximum value table data can have

`[]` (default) | `finite, real, double, scalar`

Specify the maximum value for table data. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `TableMax`
Type: character vector
Values: scalar

Default: `'[]'`

`Table data type` — Data type of table data

`Inherit: Inherit from 'Table data'` (default) | `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>` | `<data type expression>`

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 class, for example, an enumerated data type class
- 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 data type attributes. For more information, see "Specify Data Types Using Data Type Assistant".

Programmatic Use
Block Parameter: `TableDataTypeStr`
Type: character vector
Values: `'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)'|'Enum: <class name>'|'<data type expression>'`
Default: `'Inherit: Inherit from 'Table data''`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see "Lock the Output Data Type Setting" (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`

Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `enumerated`
Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
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™.

For more information about HDL code generation, see Direct Lookup Table (n-D).

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

This block supports fixed-point data types for **Table data** only.

See Also

n-D Lookup Table

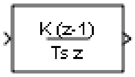
Topics

“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”
“Direct Lookup Table Algorithm for Row-Major Array Layout” (Simulink Coder)
“Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks” (Simulink Coder)

Introduced before R2006a

Discrete Derivative

Compute discrete-time derivative
Library: Simulink / 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 nonperiodic trigger (for example, nonperiodic function-call subsystems). This configuration produces inaccurate results.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Scaled discrete time derivative

scalar | vector | matrix

Optionally scaled discrete-time derivative, specified as a scalar, vector, or matrix. For more information on how the block computes the discrete-time derivative, see “Description” on page 1-433. You specify the data type of the output signal with the **Output data type** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Main

Gain value — Scaling factor

1.0 (default) | scalar

Scaling factor used to weight the block's input at the current time step, specified as a real-valued scalar.

Programmatic Use
Block Parameter: gainval
Type: character vector

Values: scalar
Default: '1.0'

Initial condition for previous weighted input K*u/Ts — Initial condition

0.0 (default) | scalar

Initial condition for the previous scaled input, specified as a scalar.

Programmatic Use
Block Parameter: ICPrevScaledInput
Type: character vector
Values: scalar
Default: '0.0'

Input processing — Specify sample- or frame-based processing

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

Signal Attributes

`Output minimum` — Minimum output value for range checking

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)`

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

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'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'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`

Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

- Depends on absolute time when used inside a triggered subsystem hierarchy.
- Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

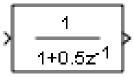
Derivative | Discrete-Time Integrator

Introduced before R2006a

Discrete Filter

Model Infinite Impulse Response (IIR) filters

Library: Simulink / 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 `Direct form I`, `Direct form I transposed`, `Direct form II`, or `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 the input dimensions, 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 d1 and d2.	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 three-channel input signal and a filter with two delay elements: $[d_1 d_2 D_1 D_2 d_1 d_2]$ or $\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$ • 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: • 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`
	• `number of zeros - 1`
	• `number of poles - 1`

`Direct form II`
`Direct form II transposed`
	`max(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)
• Column vector (*K*-by-1) • Unoriented vector (*K*)	1	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)	• Scalar • Column vector (*M*-by-1)
• Row vector (1-by-*N*) • Matrix (*K*-by-*N*)	*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-*N*)	• Scalar • Matrix (*M*-by-*N*)

Sample-Based Processing

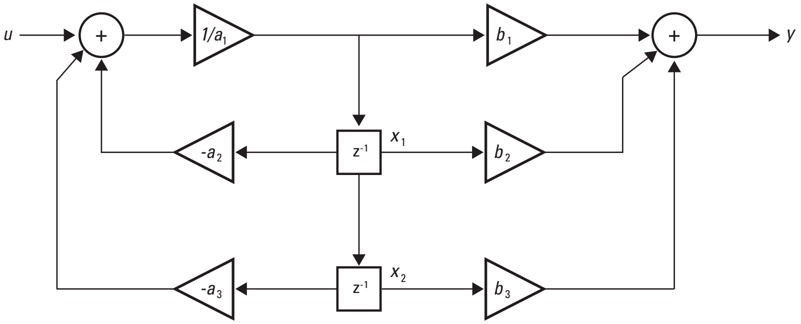
Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
• Scalar	1	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)
• Row vector (1-by-*N*) • Column vector (*N*-by–1) • Unoriented vector (*N*)	*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-*N*)	• Scalar
• Matrix (*K*-by-*N*)	*K* × *N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-(*K*×*N*))	• 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(number of zeros, number of poles)`.
- The matrix must have the same number of rows as the number of delay elements in the filter, `M = max(number of zeros, 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:

$$y_1 = b_1 \left[\frac{(u_1 - a_2 x_1 - a_3 x_2)}{a_1} \right] + b_2 x_1 + b_3 x_2$$



Ports

Input

u — Input signal

scalar | vector | matrix

Input signal to filter, specified as a scalar, vector, or matrix.

Dependencies

The name of this port depends on the source you specify for the numerator coefficients, denominator coefficients and initial states. When you set **Numerator**, **Denominator**, and **Initial states** to `Dialog`, there is only one input port, and the port is unlabeled. When you set **Numerator**, **Denominator**, or **Initial states** to `Input port`, this port is labeled **u**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

`Num` — Numerator coefficients

scalar | vector | matrix

Numerator coeffcients of the discrete filter, specified as descending powers of *z*. Use a row vector to specify the coefficients for a single numerator polynomial.

Dependencies

To enable this port, set **Numerator** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

`Den` — Denominator coefficients

scalar | vector

Specify the denominator coefficients of the discrete filter as descending powers of z. Use a row vector to specify the coefficients for a single denominator polynomial.

Dependencies

To enable this port, set **Denominator** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

`x0` — Initial states

scalar | vector | matrix

Initial states, specified as a scalar, vector, or matrix. For more information about specifying states, see “Specifying Initial States” on page 1-442.

Dependencies

To enable this port, set the **Filter structure** to `Direct form II` or `Direct form II transposed`, and set **Initial states** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

Output

`Port_1` — Filtered output signal

scalar | vector | matrix

Filtered output signal. The output dimensions equal the input dimensions, 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

Parameters

Main

Filter structure — Filter structure

Direct form II (default) | Direct form I transposed | Direct form I | Direct form II transposed

Specify the discrete IIR filter structure.

Dependencies

To use any filter structure other than Direct form II, you must have an available DSP System Toolbox license.

Programmatic Use
Block Parameter: FilterStructure
Type: character vector
Values: 'Direct form II' | 'Direct form I transposed' | 'Direct form I' | 'Direct form II transposed'
Default: 'Direct form II'

Numerator Source — Source of numerator coeffcients

Dialog (default) | Input port

Specify the source of the numerator coefficients as Dialog or Input port.

Programmatic Use
Block Parameter: NumeratorSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Numerator Value — Numerator coefficients

[1] (default) | scalar | vector | matrix

Specify the numerator coefficients of the discrete filter as descending powers of z. Use a row vector to specify the coefficients for a single numerator polynomial.

Dependencies

To enable this parameter, set the **Numerator Source** to Dialog.

Programmatic Use
Block Parameter: `Numerator`
Type: character vector
Values: scalar | vector | matrix
Default: `'[1]'`

`Denominator Source` — Source of denominator coefficients

`Dialog` (default) | `Input port`

Specify the source of the denominator coefficients as `Dialog` or `Input port`.

Programmatic Use
Block Parameter: `DenominatorSource`
Type: character vector
Values: `'Dialog' | 'Input port'`
Default: `'Dialog'`

`Denominator Value` — Denominator coeffcients

`[1 0.5]` (default) | vector

Specify the denominator coefficients of the discrete filter as descending powers of z. Use a row vector to specify the coefficients for a single denominator polynomial.

Dependencies

To enable this parameter, set the **Denominator Source** to `Dialog`.

Programmatic Use
Block Parameter: `Denominator`
Type: character vector
Values: scalar | vector
Default: `'[1 0.5]'`

`Initial states Source` — Source of initial states

`Dialog` (default) | `Input port`

Specify the source of the initial states as `Dialog` or `Input port`.

Programmatic Use
Block Parameter: `InitialStatesSource`
Type: character vector
Values: `'Dialog' | 'Input port'`
Default: `'Dialog'`

`Initial states Value` — Initial filter states

`0` (default) | scalar | vector | matrix

Specify the initial filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-442.

Dependencies

To enable this parameter, set the **Filter structure** to `Direct form II` or `Direct form II transposed`, and set **Initial states Source** to `Dialog`.

Programmatic Use

Block Parameter: `InitialStates`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`Initial states on numerator side` — Initial numerator states

`0` (default) | scalar | vector | matrix

Specify the initial numerator filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-442.

Dependencies

To enable this port, set the **Filter structure** to `Direct form I` or `Direct form I transposed`.

Programmatic Use

Block Parameter: `InitialStates`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`Initial states on denominator side` — Initial denominator states

`0` (default) | scalar | vector | matrix

Specify the initial denominator filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-442.

Dependencies

To enable this port, set the **Filter structure** to `Direct form I` or `Direct form I transposed`.

Programmatic Use
Block Parameter: `InitialDenominatorStates`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`External reset` — External state reset

`None` (default) | `Rising` | `Falling` | `Either` | `Level` | `Level hold`

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: • 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

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector
Values: `'None'` | `'Rising'` | `'Falling'` | `'Either'` | `'Level'` | `'Level hold'`
Default: `'None'`

`Input processing` — Sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

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.

Dependencies

Frame-based processing requires a DSP System Toolbox license.

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

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)' | 'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

Optimize by skipping divide by leading denominator coefficient (a0) — Skip divide by a0

`off` (default) | `on`

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. It performs a divide-by-a_0 in both simulation and code generation.

Programmatic Use
Block Parameter: `a0EqualsOne`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Sample time (-1 for inherited) — Interval between samples

`-1` (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `SampleTime`

Type: character vector
Values: scalar | vector
Default: `'-1'`

Data Types

`State` — State data type

`Inherit: Same as input` (default) | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `StateDataTypeStr`
Type: character vector
Values: `'Inherit: Same as input'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Same as input'`

`Numerator coefficients` — Numerator coefficient data type

`Inherit: Inherit via internal rule` (default) | `int8` | `int16` | `int32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

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 signed 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 data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: `NumCoeffDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Numerator coefficient minimum` — Minimum value of numerator coefficients

`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `NumCoeffMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Numerator coefficient maximum` — Maximum value of numerator coefficients

`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `NumCoeffMax`

Type: character vector
Values: scalar
Default: `'[]'`

`Numerator product output` — Numerator product output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use

Block Parameter: `NumProductDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via interal rule'`

`Numerator accumulator` — Numerator accumulator data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `NumAccumDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Same as product output' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via interal rule'`

`Denominator coefficients` — Denominator coefficient data type
`Inherit: Inherit via internal rule` (default) | `int8` | `int16` | `int32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `DenCoeffDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Denominator coefficient minimum — Minimum value of denominator coefficients

[] (default) | scalar

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

Programmatic Use
Block Parameter: DenCoeffMin
Type: character vector
Values: scalar
Default: '[]'

Denominator coefficient maximum — Maximum value of denominator coefficients

[] (default) | scalar

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

Programmatic Use
Block Parameter: DenCoeffMax
Type: character vector
Values: scalar
Default: '[]'

Denominator product output — Denominator product output data type

Inherit: Inherit via internal rule (default) | Inherit: Same as input | int8 | int16 | int32 | fixdt(1,16,0) | <data type expression>

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

Programmatic Use
Block Parameter: `DenProductDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Denominator accumulator` — Denominator accumulator data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `DenAccumDataTypeStr`

Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Same as product output' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Output` — Output data type

`Inherit: Inherit via internal rule` (default) | `int8` | `int16` | `int32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use

Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Output minimum` — Minimum value of output

`[]` (default) | scalar

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

Programmatic Use

Block Parameter: `OutMin`

Type: character vector
Values: scalar
Default: `'[]'`

`Output maximum` — Maximum value of output

`[]` (default) | scalar

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

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Multiplicand data type` — Multiplcand data type

`Inherit: Same as input` (default) | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

Specify the multiplicand data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- 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 data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Dependencies

To enable this parameter, set the **Filter structure** to `Direct form I transposed`

Programmatic Use

Block Parameter: `MultiplicandDataTypeStr`

Type: character vector
Values: `'Inherit: Same as input'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Same as input'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select to lock 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).

Programmatic Use
Block Parameter: `LockScale`
Values: `'off'` | `'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations. For more information, see "Rounding" (Fixed-Point Designer).

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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 (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Check for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

`State name` — Assign unique name to each state

`' '` (default) | `'position'` | `{'a', 'b', 'c'}` | `a` | ...

Assign a unique name to each state. If this field is blank (`' '`), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'position'`.
- 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 in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Limitations

- 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.

Dependencies

To enable this parameter, set **Filter structure** to `Direct form II`.

Programmatic Use
Block Parameter: `StateName`
Type: character vector
Values: `' '` | user-defined
Default: `' '`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, set **Filter structure** to `Direct form II` and specify a value for **State name**. 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**.

Programmatic Use

Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Signal object class

`Simulink.Signal` (default) | object of a class that is derived from `Simulink.Signal`

Specify the signal object class.

Dependencies

To enable this parameter, set **Filter structure** to `Direct form II` and specify a value for **State name**.

Programmatic Use

Block Parameter: `StateSignalObject`
Type: character vector
Values: object of a class that is derived from `Simulink.Signal`
Default: `'Simulink.Signal'`

`Code generation storage class` — State storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `BitField (Custom)` | `Volatile (Custom)` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `Localizable (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation.

- `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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

To enable this parameter, set **Filter structure** to `Direct form II`, and specify a value for **State name**.

Programmatic Use

Block Parameter: `StateStorageClass`
Type: character vector
Values:`'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

`Code generation storage type qualifier` — Storage type qualifier

`''` (default)

Specify a storage type qualifier such as `const` or `volatile`.

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.

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 “Data Objects”.

Dependencies

To enable this parameter, set **Filter structure** to `Direct form II`. This parameter is hidden unless you previously set its value.

Programmatic Use
Block Parameter: `RTWStateStorageTypeQualifier`
Type: character vector
Values: `'' | 'const' | 'volatile' | ...`
Default: `''`

Block Characteristics

Data Types	`double` | `single` | `base integer`[a] | `fixed point`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[a] This block only supports signed fixed-point data types.

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

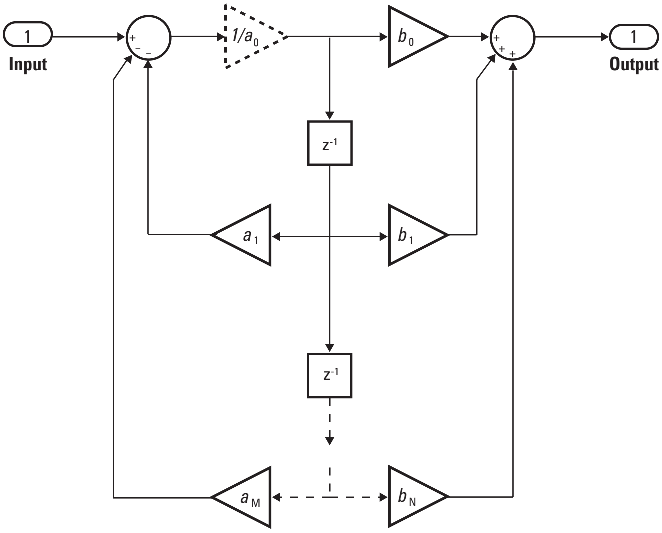
Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

This block only supports signed fixed-point data types.

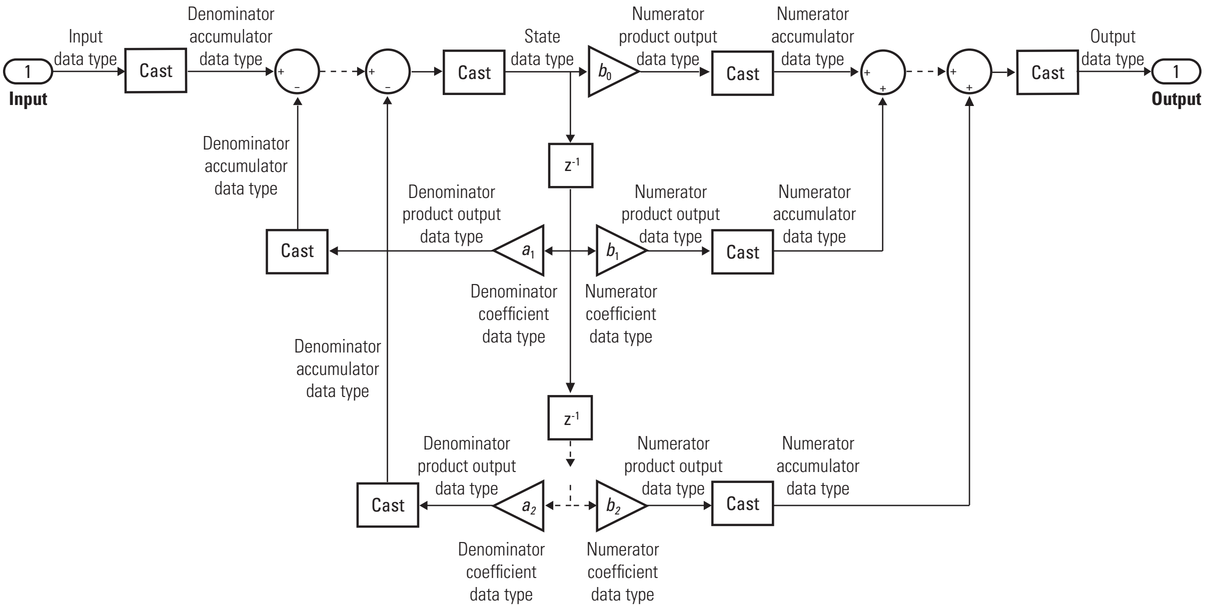
The Discrete Filter block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

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.


Input data type
Denominator accumulator data type
State data type
Numerator product output data type
Numerator accumulator data type
Output data type
1
Input
Cast
Cast
b_0
Cast
Cast
1
Output
z^{-1}
Denominator accumulator data type
Denominator product output data type
Numerator product output data type
Numerator accumulator data type
Cast
a_1
b_1
Cast
Denominator coefficient data type
Numerator coefficient data type
Denominator accumulator data type
z^{-1}
Denominator product output data type
Numerator product output data type
Numerator accumulator data type
Cast
a_2
b_2
Cast
Denominator coefficient data type
Numerator coefficient data type

See Also

Allpole Filter | Digital Filter Design | Discrete FIR Filter | Filter Realization Wizard | `dsp.AllpoleFilter` | `dsp.IIRFilter` | `filterDesigner` | `fvtool`

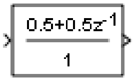
Topics

“Sample- and Frame-Based Concepts” (DSP System Toolbox)
“Working with States”

Introduced before R2006a

Discrete FIR Filter

Model FIR filters
Library: Simulink / 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, and 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 the input dimensions, 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. For more information, 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)	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: • The vector length equal the product of the number of input channels and the number of delay elements in the filter, `#_of_filter_coeffs-1` (or `#_of_reflection_coeffs` for `Lattice MA`). • The matrix must have the same number of rows as the number of delay elements in the filter, `#_of_filter_coeffs-1` (`#_of_reflection_coeffs` for `Lattice MA`), and must have one column for each channel of the input signal.

Ports

Input

In — Input signal

scalar | vector | matrix

Input signal to filter, specified as a scalar, vector, or matrix.

Dependencies

When you set **Coefficient source** to `Dialog parameters`, the port for the input signal is unlabeled. When you set **Coefficient source** to `Input port`, the port for the input signal is labeled **In**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Num — Filter coefficients

scalar | vector | matrix

Specify the filter coefficents as a scalar, vector, or matrix. 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.

Dependencies

To enable this port, set **Coefficient source** to `Input port`.

To implement multiple filters, **Filter structure** must be `Direct form`, and the input must be a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

External reset — External reset signal

scalar

External reset signal, specified as a scalar. When the specified trigger event occurs, the block resets the states to their initial conditions.

Tip The icon for this port changes based on the value of the **External reset** parameter.

Dependencies

To enable this port, set **External reset** to `Rising`, `Falling`, `Either`, `Level`, or `Level hold`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

□ — Enable signal
scalar

Enable signal, specified as a scalar. This port can control execution of the block. The block is 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.

Dependencies

To enable this port, select the **Show enable port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Filtered output signal
scalar | vector | matrix

Filtered output signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Main

`Coefficient source` — Source of coefficients
`Dialog parameters` (default) | `Input port`

Choose to specify the filter coefficients using tunable dialog parameters or through an input port, which is useful for time-varying cofficients.

Programmatic Use
Block Parameter: `CoefSource`
Type: character vector
Values: `'Dialog parameters' | 'Input port'`
Default: `'Dialog parameters'`

`Filter structure` — Filter structure

`Direct form` (default) | `Direct form symmetric` | `Direct form antisymmetric` | `Direct form transposed` | `Lattice MA`

Select the filter structure you want the block to implement.

Dependencies

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`.

Programmatic Use
Block Parameter: `FilterStructure`
Type: character vector
Values: `'Direct form' | 'Direct form symmetric' | 'Direct form antisymmetric' | 'Direct form transposed' | 'Lattice MA'`
Default: `'Direct form'`

`Coefficients` — Filter coefficients

`[0.5 0.5]` (default) | vector | matrix

Specify the coefficient vector for the 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.

Dependencies

To enable this parameter, set **Coefficient source** to `Dialog parameters`.

To implement multiple filters, **Filter structure** must be `Direct form`, and the input must be a scalar.

Programmatic Use
Block Parameter: `Coefficients`
Type: character vector
Values: vector
Default: `'[0.5 0.5]'`

`Input processing` — Sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)' | 'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Initial states` — Initial conditions of filter states

`0` (default) | scalar | vector | matrix

Specify the initial conditions of the filter states. To learn how to specify initial states, see “Specifying Initial States” on page 1-470.

Programmatic Use
Block Parameter: `InitialStates`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`Show enable port` — Create enable port

`off` (default) | `on`

Select to control execution of this block with an enable port. 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.

Programmatic Use
Block Parameter: `ShowEnablePort`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`External reset` — External state reset

`None` (default) | `Rising` | `Falling` | `Either` | `Level` | `Level hold`

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: • 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

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector
Values: `'None'` | `'Rising'` | `'Falling'` | `'Either'` | `'Level'` | `'Level hold'`
Default: `'None'`

`Sample time (-1 for inherited)` — Time interval between samples

`-1` (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Data Types

`Tap sum` — Tap sum data type

`Inherit: Same as input` (default) | `Inherit: Inherit via internal rule` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16,0)` | `<data type expression>`

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)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Dependencies

This parameter is only visible when you set the **Filter structure** to `Direct form symmetric` or `Direct form antisymmetric`.

Programmatic Use
Block Parameter: `TapSumDataTypeStr`

Type: character vector
Values: `'Inherit: Same as input' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Same as input'`

`Coefficients` — Coefficient data type

`Inherit: Same wordlength as input` (default) | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use

Block Parameter: `CoeffDataTypeStr`
Type: character vector
Values: `'Inherit: Same word length as input'| 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Same wordlength as input'`

`Coefficients minimum` — Minimum value of coefficients

`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `CoeffMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Coefficients maximum` — Maximum value of coefficients

`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `CoeffMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Product output` — Product output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use

Block Parameter: `ProductDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Same as input'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Accumulator` — Accumulator data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use

Block Parameter: `AccumDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Same as input'` | `'Inherit: Same as product output'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`State` — State data type

`Inherit: Same as accumulator` (default) | `Inherit: Same as input` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16,0)` | `<data type expression>`

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)`

Dependencies

To enable this parameter, set the **Filter structure** to `Lattice MA`.

Programmatic Use
Block Parameter: `StateDataTypeStr`
Type: character vector
Values: `'Inherit: Same as accumulator'` | `'Inherit: Same as input'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Same as accumulator'`

`Output` — Output data type

`Inherit: Same as accumulator` (default) | `Inherit: Same as input` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Same as accumulator'` | `'Inherit: Same as input'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'<data type expression>'`

Default: `'Inherit: Same as accumulator'`

`Output minimum` — Minimum output value for range checking

`[ ]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[ ]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select to lock 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).

Programmatic Use
Block Parameter: `LockScale`
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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 (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

This block supports custom state attributes to customize and generate code more efficiently. To access or set these attributes, in the Simulink editor, select **View > Model Data Editor** or press **Ctrl+Shift+E**. For an example, see “Custom State Attributes in Discrete FIR Filter block”.

HDL Code Generation

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

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Discrete FIR Filter.

For hardware-friendly valid and reset control signals, and to model exact hardware latency behavior in Simulink, use the Discrete FIR Filter HDL Optimized block instead.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

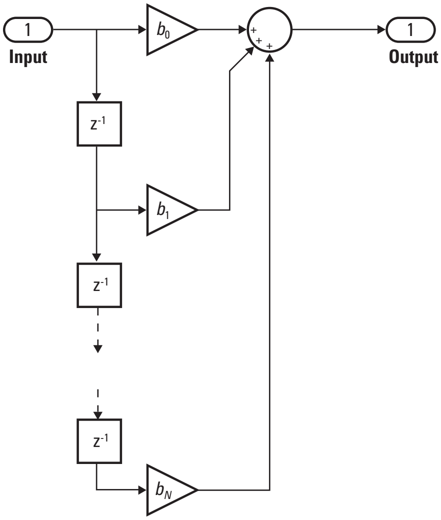
Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

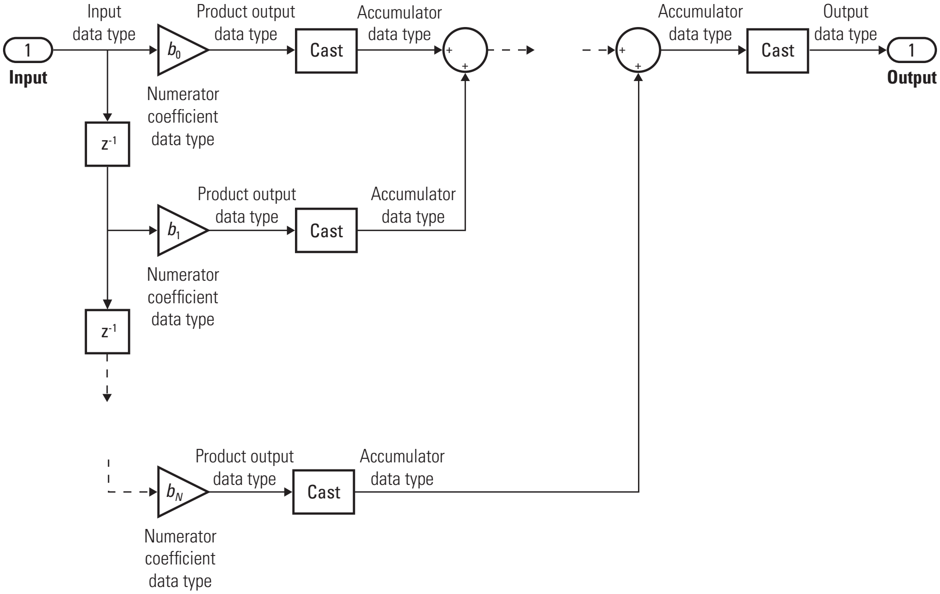
The Discrete FIR Filter block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.

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.

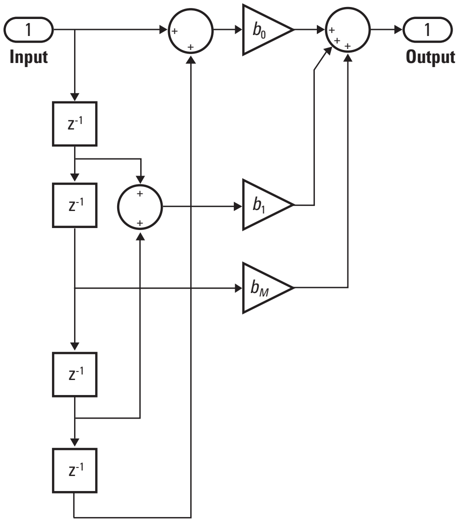




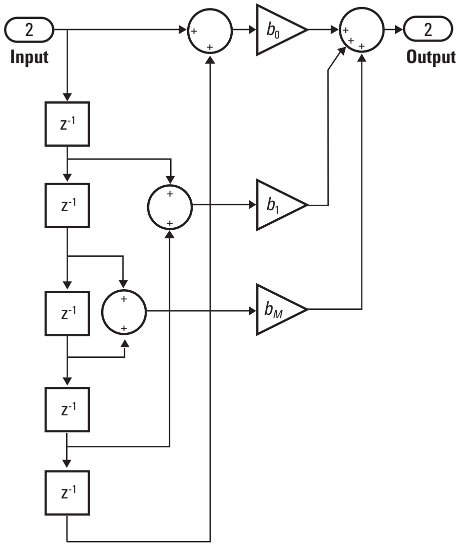
Direct Form Symmetric

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.

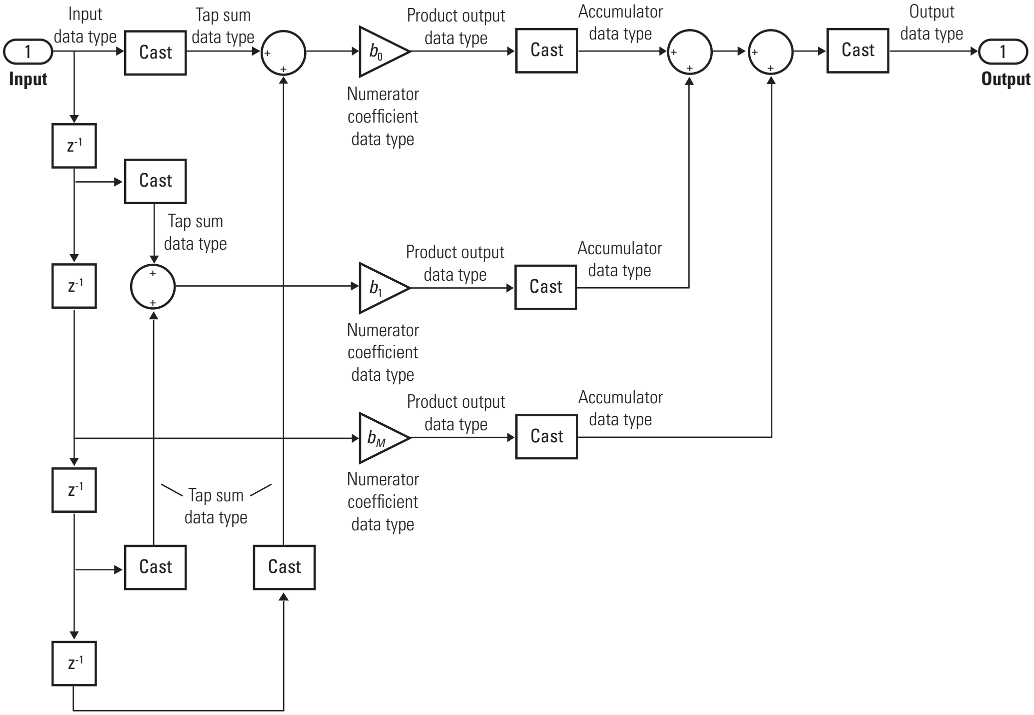
It is assumed that the filter coefficients are symmetric. The block only uses the first half of the coefficients for filtering.



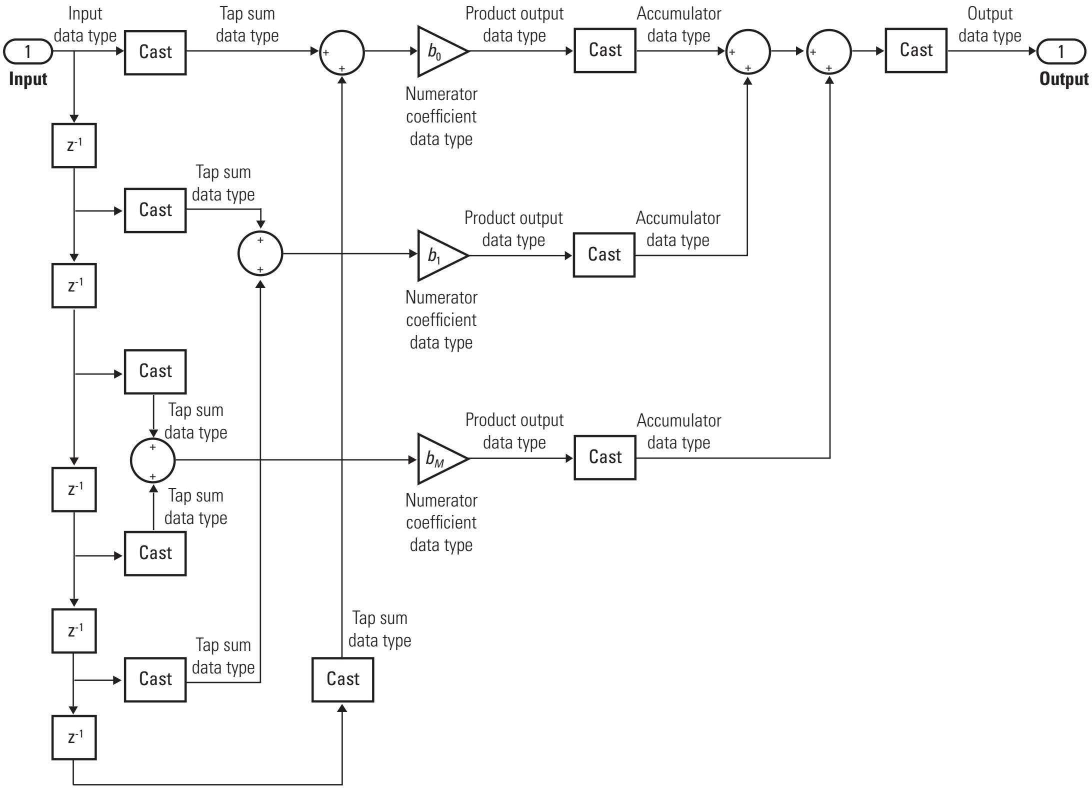
Even Order - Type I



Odd Order - Type II



Even Order - Type I

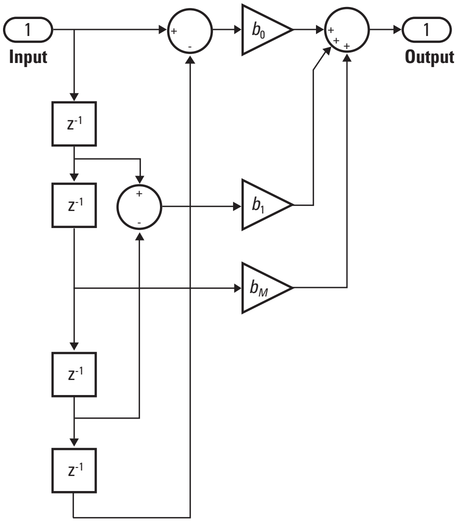


Odd Order - Type II

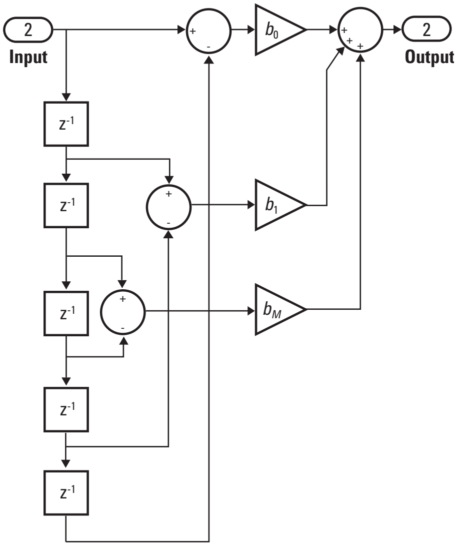
Direct Form Antisymmetric

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.

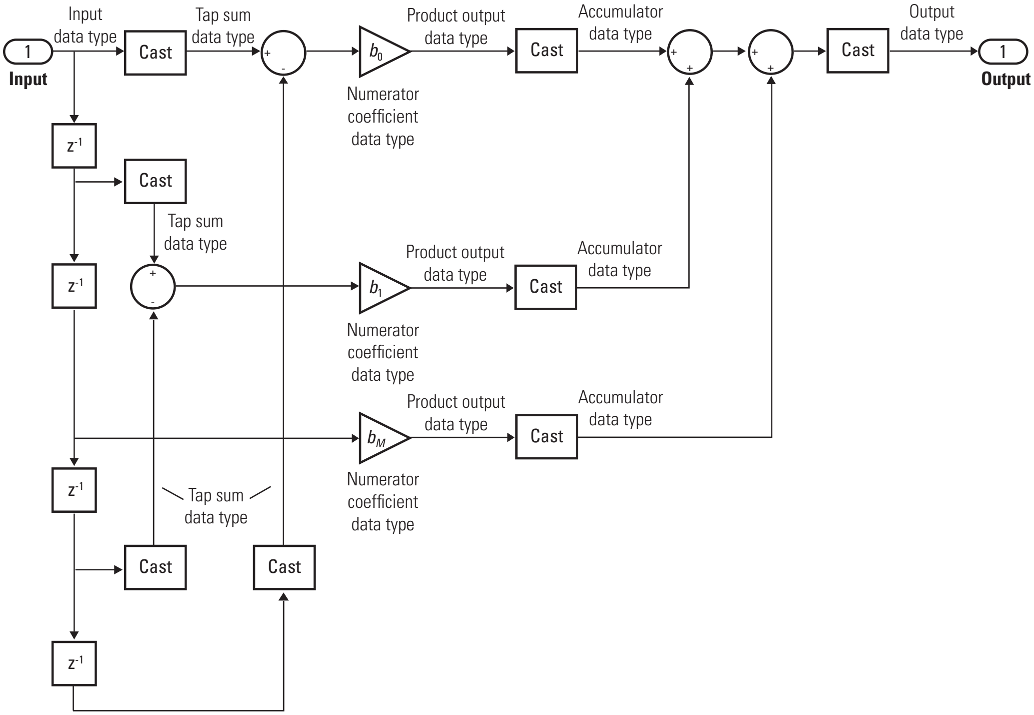
It is assumed that the filter coefficients are antisymmetric. The block only uses the first half of the coefficients for filtering.



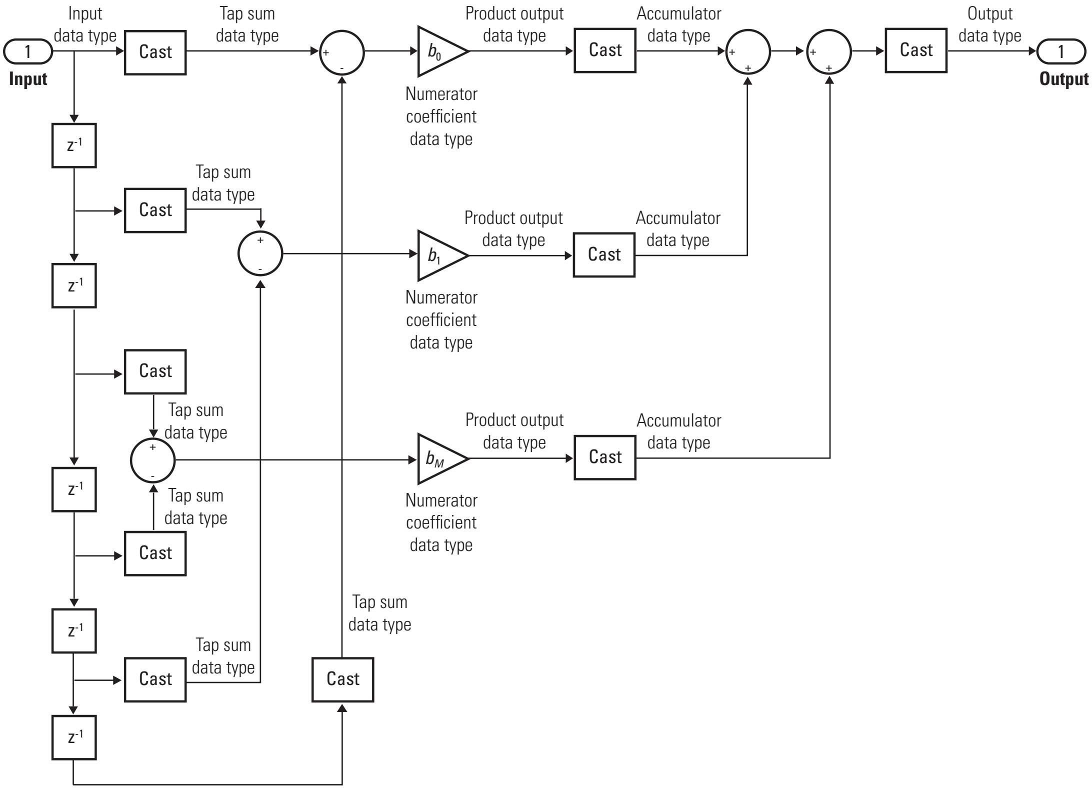
Even Order - Type III



Odd Order - Type IV



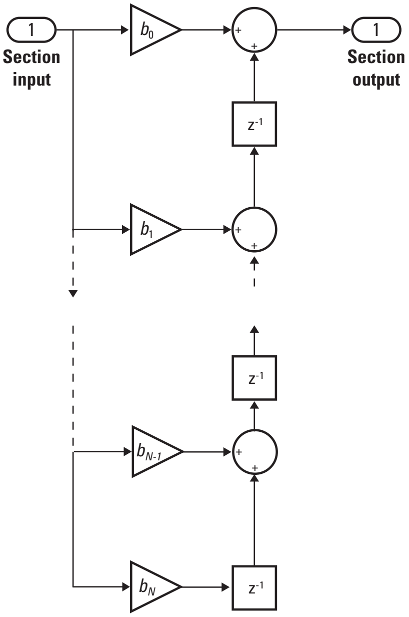
Even Order - Type III

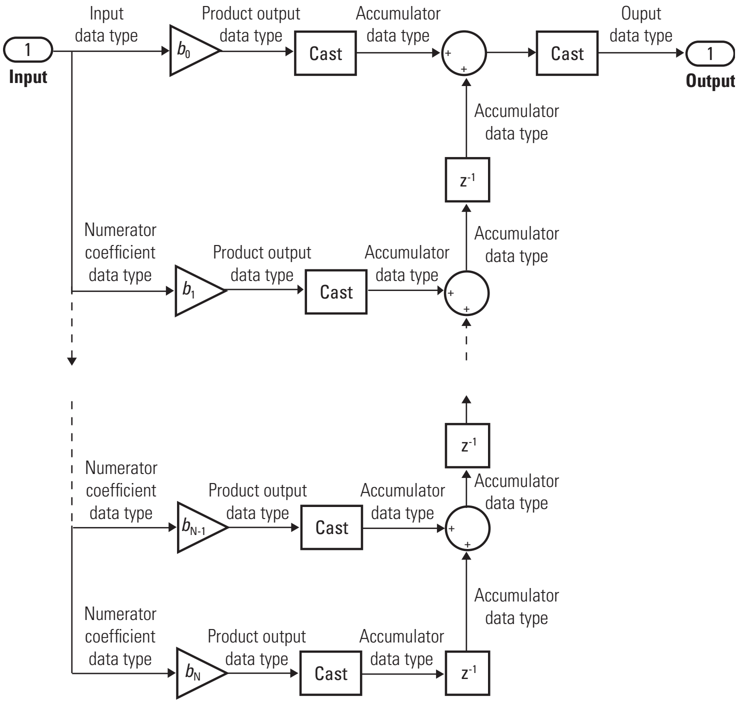


Odd Order - Type IV

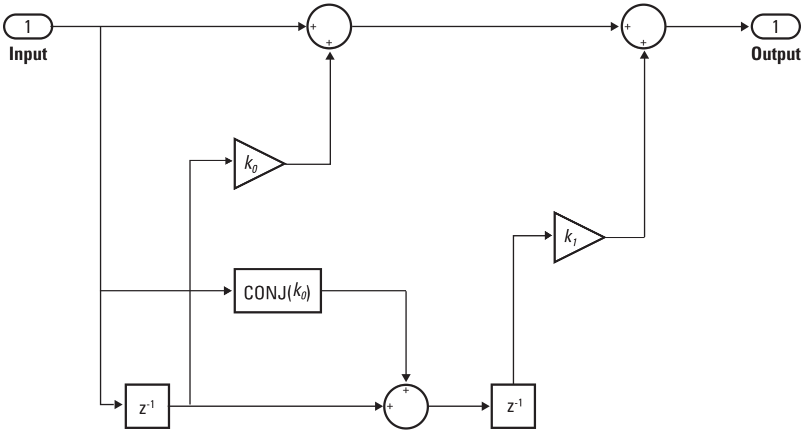
Direct Form Transposed

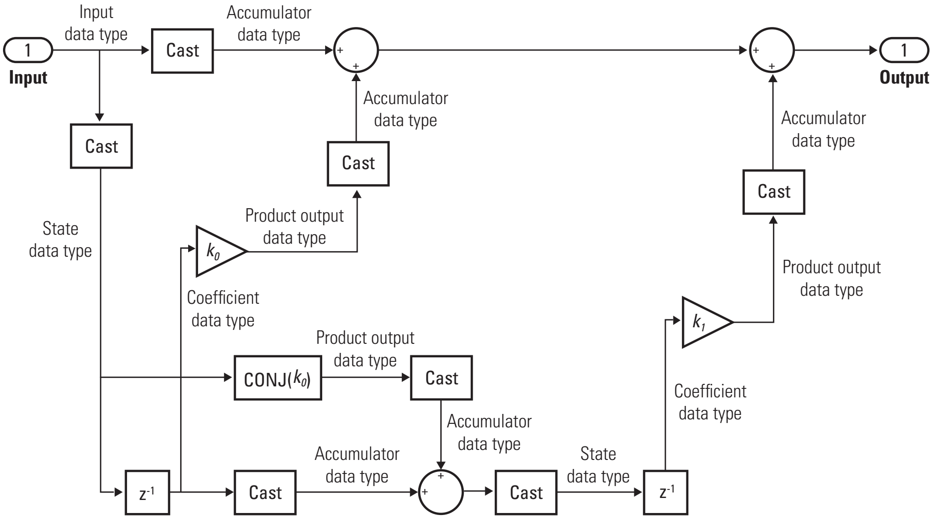
States are complex when either the inputs or the coefficients are complex.


1
Section input
b_0
+
+
1
Section output
z^{-1}
b_1
+
+
z^{-1}
b_{N-1}
+
+
b_N
z^{-1}


Input
data type
Product output
data type
Accumulator
data type
Ouput
data type
1
Input
b_0
Cast
+
+
Cast
1
Output
Accumulator
data type
z^{-1}
Accumulator
data type
Numerator
coefficient
data type
b_1
Product output
data type
Cast
Accumulator
data type
z^{-1}
Numerator
coefficient
data type
b_{N-1}
Product output
data type
Cast
Accumulator
data type
Accumulator
data type
Numerator
coefficient
data type
b_N
Product output
data type
Cast
Accumulator
data type
z^{-1}
Accumulator
data type

Lattice MA


1
Input
+
+
+
+
1
Output
k_0
k_1
CONJ(k_0)
z^{-1}
+
+
z^{-1}



See Also

Digital Filter Design | Discrete Filter

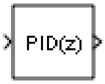
Topics

“Sample- and Frame-Based Concepts” (DSP System Toolbox)
“Working with States”

Introduced in R2008a

Discrete PID Controller

Discrete-time or continuous-time PID controller
Library: Simulink / Discrete



Description

The Discrete PID Controller block implements a PID controller (PID, PI, PD, P only, or I only). The block is identical to the PID Controller block with the **Time domain** parameter set to `Discrete-time`.

The 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.

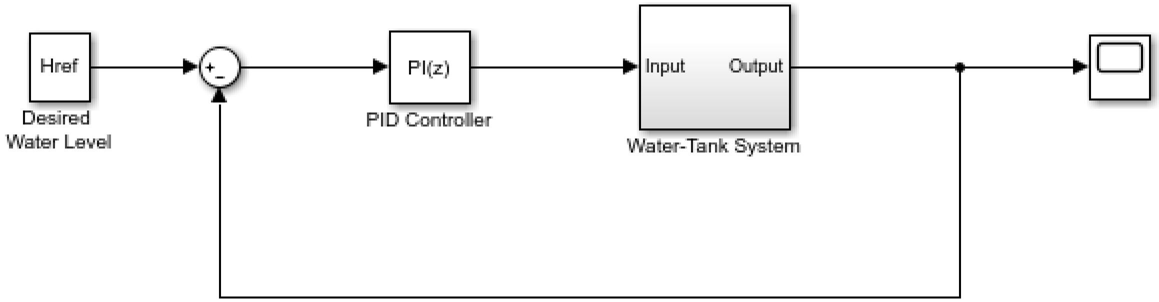
The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, PD, P only, or I only) — See the **Controller** parameter.
- Controller form (Parallel or Ideal) — See the **Form** parameter.
- Time domain (continuous or discrete) — See the **Time domain** parameter.
- Initial conditions and reset trigger — See the **Source** and **External reset** parameters.
- Output saturation limits and built-in anti-windup mechanism — See the **Limit output** parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the **Enable tracking mode** parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (For more information, see “Variant Subsystems”). To examine the internal structure of the block and its variant subsystems, right-click the block and select **Mask > Look Under Mask**.

Control Configuration

In one common implementation, the PID Controller block operates in the feedforward path of a 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 Discrete PID Controller (2DOF).

PID Gain Tuning

The PID controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design™ software. For more information about automatic tuning, see the **Select tuning method** parameter.

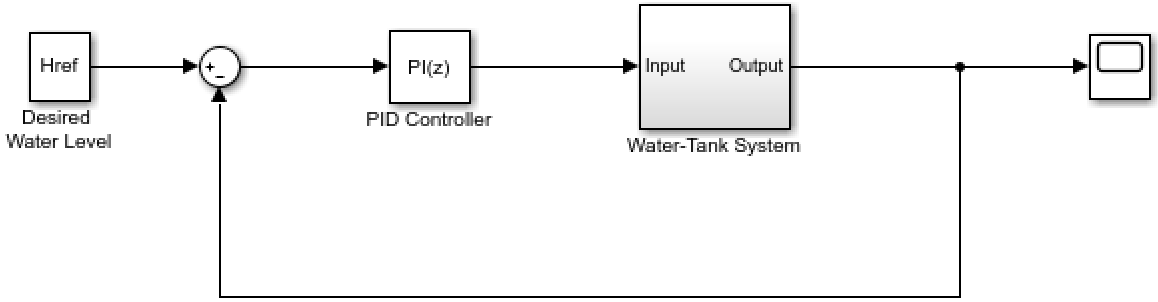
Ports

Input

`Port_1( u )` — Error signal input

scalar | vector

Difference between a reference signal and the output of the system under control, as shown.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

P — Proportional gain

scalar | vector

Proportional gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

I — Integral gain

scalar | vector

Integral gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`D` — Derivative gain

scalar | vector

Derivative gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`N` — Filter coefficient

scalar | vector

Derivative filter coefficient, provided from a source external to the block. External coefficient 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

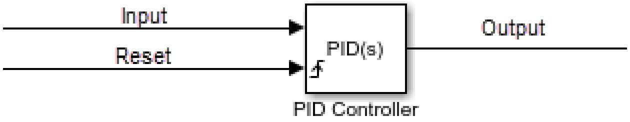
To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has a filtered derivative.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Reset — External reset trigger

scalar

Trigger to reset the integrator and filter to their initial conditions. The value of the **External reset** parameter determines whether reset occurs on a rising signal, a falling signal, or a level signal. The port icon indicates the selected trigger type. For example, the following illustration shows a continuous-time PID block with **External reset** set to `rising`.



When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the **Integrator Initial condition** and **Filter Initial condition** parameters or the $\mathbf{I_0}$ and $\mathbf{D_0}$ ports.

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.

Dependencies

To enable this port, set **External reset** to any value other than `none`.

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

I_0 — Integrator initial condition

scalar | vector

Integrator initial condition, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

D_0 — Filter initial condition

scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

TR — Tracking signal

scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input. Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the **Enable tracking mode** parameter.

Dependencies

To enable this port, select the **Enable tracking mode** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1( y )` — Controller output

scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the **Controller** parameter. The base controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the **Upper Limit** and **Lower Limit** saturation parameters.

The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as *N* independent PID controllers, where *N* is the number of signals in the input vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Controller` — Controller type

`PID` (default) | `PI` | `PD` | `P` | `I`

Specify which of the proportional, integral, and derivative terms are in the controller.

`PID`

Proportional, integral, and derivative action.

`PI`

Proportional and integral action only.

`PD`

Proportional and derivative action only.

`P`

Proportional action only.

`I`

Integral action only.

Tip The controller transfer function for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use
Block Parameter: `Controller`
Type: string, character vector
Values: `"PID"`, `"PI"`, `"PD"`, `"P"`, `"I"`
Default: `"PID"`

`Form` — Controller structure

`Parallel` (default) | `Ideal`

Specify whether the controller structure is parallel or ideal.

`Parallel`

The controller output is the sum of the proportional, integral, and derivative actions, weighted independently by **P**, **I**, and **D**, respectively. For example, for a continuous-time parallel-form PID controller, the transfer function is:

$$C_{par}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right).$$

For a discrete-time parallel-form controller, the transfer function is:

$$C_{par}(z) = P + I\alpha(z) + D\left[\frac{N}{1+N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

`Ideal`

The proportional gain **P** acts on the sum of all actions. For example, for a continuous-time ideal-form 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-form controller, the transfer function is:

$$C_{id}(z) = P\left[1 + I\alpha(z) + D\frac{N}{1+N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $a(z)$ and $b(z)$, respectively.

Tip The controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use
Block Parameter: `Controller`
Type: string, character vector
Values: `"Parallel"`, `"Ideal"`
Default: `"Parallel"`

`Time domain` — Specify discrete-time or continuous-time controller

`Discrete-time` (default) | `Continuous-time`

When you select `Discrete-time`, it is recommended that you specify an explicit sample time for the block. See the **Sample time (-1 for inherited)** parameter. Selecting `Discrete-time` also enables the **Integrator method**, and **Filter method** parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select `Continuous-time`.

Note The PID Controller and Discrete PID Controller blocks are identical except for the default value of this parameter.

Programmatic Use
Block Parameter: `TimeDomain`
Type: string, character vector
Values: `"Continuous-time"`, `"Discrete-time"`
Default: `"Continuous-time"`

`Sample time (-1 for inherited)` — Discrete interval between samples

–1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of –1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.

To implement a continuous-time controller, set **Time domain** to `Continuous-time`.

Tip If you want to run the block with an externally specified or variable sample time, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use
Block Parameter: `SampleTime`
Type: scalar
Values: `-1`, positive scalar
Default: `-1`

`Integrator method` — Method for computing integral in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the integral term of the controller transfer function is $I\alpha(z)$, where $\alpha(z)$ depends on the integrator method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z) = \frac{T_s}{z - 1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z) = \frac{T_s z}{z - 1}.$$

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`

Bilinear approximation,

$$\alpha(z) = \frac{T_s}{2}\frac{z+1}{z-1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

Note For the `BackwardEuler` or `Trapezoidal` methods, you cannot generate HDL code for the block if either:

- **Limit output** is selected and **Anti-Windup Method** is anything other than `none`.
- **Enable tracking mode** is selected.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with integral action.

Programmatic Use

Block Parameter: `IntegratorMethod`
Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

`Filter method` — Method for computing derivative in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the derivative term of the controller transfer function is:

$$D\left[\frac{N}{1+N\alpha(z)}\right],$$

where $\alpha(z)$ depends on the filter method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z)=\frac{T_s}{z-1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z)=\frac{T_s z}{z-1}.$$

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`

Bilinear approximation,

$$\alpha(z)=\frac{T_s}{2}\frac{z+1}{z-1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with derivative action.

Programmatic Use

Block Parameter: `FilterMethod`
Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

Main

`Source` — Source for controller gains and filter coefficient

internal (default) | external

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.

`internal`

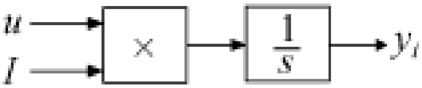
Specify the controller gains and filter coefficient using the block parameters **P**, **I**, **D**, and **N**.

`external`

Specify the PID gains and filter coefficient externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

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 gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. 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 input signal *u* 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.

Programmatic Use
Block Parameter: `ControllerParametersSource`
Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Proportional (P)` — Proportional gain

1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- `Parallel` — Proportional action is independent of the integral and derivative actions. For instance, for a continuous-time parallel PID controller, the transfer function is:

  $$C_{par}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right).$$

  For a discrete-time parallel-form controller, the transfer function is:

  $$C_{par}(z) = P + I\alpha(z) + D\left[\frac{N}{1+N\beta(z)}\right],$$

  where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

- `Ideal` — The proportional gain multiples the integral and derivative terms. For instance, 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-form controller, the transfer function is:

  $$C_{id}(z) = P\left[1 + I\alpha(z) + D\frac{N}{1 + N\beta(z)}\right],$$

  where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to `PID`, `PD`, `PI`, or `P`.

Programmatic Use

Block Parameter: `P`
Type: scalar, vector
Default: 1

`Integral (I)` — Integral gain

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `I`
Type: scalar, vector

Default: 1

`Derivative (D)` — Derivative gain

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to `PID` or `PD`.

Programmatic Use

Block Parameter: `D`
Type: scalar, vector
Default: 0

`Use filtered derivative` — Apply filter to derivative term

`on` (default) | `off`

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller transfer function becomes:

$$D\frac{z-1}{zT_s}.$$

For continuous-time PID controllers, the derivative term is always filtered.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`, and set **Controller** to a type that has derivative action.

Programmatic Use

Block Parameter: `UseFilter`
Type: string, character vector
Values: `"on"`, `"off"`
Default: `"on"`

`Filter coefficient (N)` — Derivative filter coefficient

100 (default) | scalar | vector

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the **Time domain** parameter.

- When **Time domain** is `Continuous-time`, the pole location is `s = -N`.
- When **Time domain** is `Discrete-time`, the pole location depends on the **Filter method** parameter.

Filter Method	Location of Filter Pole
`Forward Euler`	$z_{pole} = 1 - NT_s$
`Backward Euler`	$z_{pole} = \dfrac{1}{1 + NT_s}$
`Trapezoidal`	$z_{pole} = \dfrac{1 - NT_s / 2}{1 + NT_s / 2}$

The block does not support `N = Inf` (ideal unfiltered derivative). When the **Time domain** is `Discrete-time`, you can clear **Use filtered derivative** to remove the derivative filter.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to `PID` or `PD`.

Programmatic Use

Block Parameter: `N`
Type: scalar, vector
Default: 100

`Select tuning method` — Tool for automatic tuning of controller coefficients

`Transfer Function Based (PID Tuner App)` (default) | `Frequency Response Based`

If you have Simulink Control Design software, you can automatically tune the PID coefficients. To do so, use this parameter to select a tuning tool, and click **Tune**.

`Transfer Function Based (PID Tuner App)`

Use **PID Tuner**, which lets you interactively tune PID coefficients 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 coefficients 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`

Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to zero. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

`Enable zero-crossing detection` — Detect zero crossings on reset and on entering or leaving a saturation state

`on` (default) | `off`

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 **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

Programmatic Use

Block Parameter: `ZeroCross`
Type: string, character vector
Values: `"on"`, `"off"`

Default: `"on"`

Initialization

`Source` — Source for integrator and derivative initial conditions

`internal` (default) | `external`

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event. (See the **External reset** parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.

`internal`

> Specify the initial conditions using the **Integrator Initial condition** and **Filter Initial condition** parameters. If **Use filtered derivative** is not selected, use the **Differentiator** parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

`external`

> Specify the initial conditions externally using block inputs. Additional input ports $\mathbf{I_o}$ and $\mathbf{D_o}$ appear on the block. If **Use filtered derivative** is not selected, supply the initial condition for the unfiltered differentiator at $\mathbf{D_o}$ instead of a filter initial condition.

Programmatic Use
Block Parameter: `InitialConditionSource`
Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Integrator` — Integrator initial condition

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `InitialConditionForIntegrator`
Type: scalar, vector
Default: 0

`Filter` — Filter initial condition

0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and use a controller that has a derivative filter.

Programmatic Use

Block Parameter: `InitialConditionForFilter`
Type: scalar, vector
Default: 0

`Differentiator` — Initial condition for unfiltered derivative

0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, set **Time domain** to `Discrete-time`, clear the **Use filtered derivative** check box, and in the **Initialization** tab, set **Source** to `internal`.

Programmatic Use
Block Parameter: `DifferentiatorICPrevScaledInput`
Type: scalar, vector
Default: 0

`Initial condition setting` — Location at which initial condition is applied

`State (most efficient)` (default) | `Output`

Use this parameter to specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. You can change this parameter at the command line only, using `set_param` to set the `InitialConditionSetting` parameter of the block.

`State (most efficient)`

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** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

Programmatic Use
Block Parameter: `InitialConditionSetting`
Type: string, character vector
Values: `"state"`, `"output"`
Default: `"state"`

`External reset` — Trigger for resetting integrator and filter values

`none` (default) | `rising` | `falling` | `either` | `level`

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than `none` enables the **Reset** port on the block for the external reset signal.

`none`

The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

`rising`

Reset the outputs when the reset signal has a rising edge.

`falling`

Reset the outputs when the reset signal has a falling edge.

`either`

Reset the outputs when the reset signal either rises or falls.

`level`

Reset the outputs when the reset signal either:

- Is nonzero at the current time step
- Changes from nonzero at the previous time step to zero at the current time step

This option holds the outputs to the initial conditions while the reset signal is nonzero.

Dependencies

To enable this parameter, set **Controller** to a type that has derivative or integral action.

Programmatic Use
Block Parameter: `ExternalReset`
Type: string, character vector
Values: `"none"`, `"rising"`, `"falling"`, `"either"`,`"level"`
Default: `"none"`

`Ignore reset when linearizing` — Force linearization to ignore reset

`off` (default) | `on`

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the **External reset** parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

Programmatic Use
Block Parameter: `IgnoreLimit`
Type: string, character vector
Values: `"off"`, `"on"`

Default: `"off"`

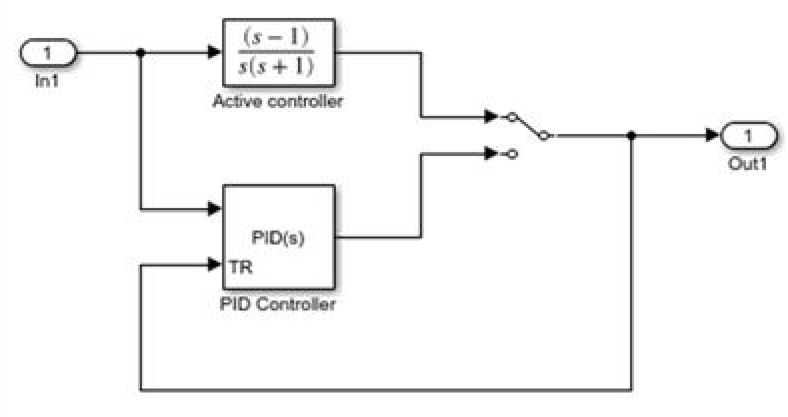
`Enable tracking mode` — Activate signal tracking

`off` (default) | `on`

Signal tracking lets the block output follow a tracking signal that you provide at the **TR** port. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input with a gain `Kt`, specified by the **Tracking gain (Kt)** parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.

Bumpless control transfer

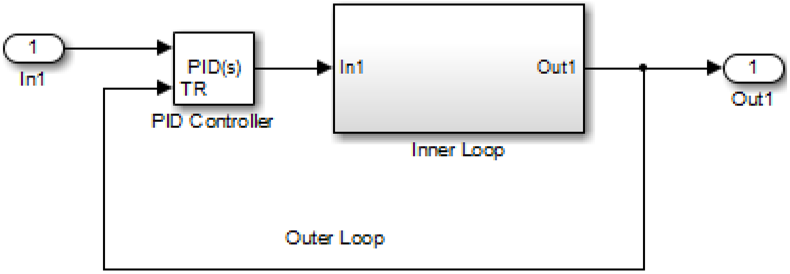
Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and another controller. To do so, connecting the controller output to the **TR** input as shown in the following illustration.



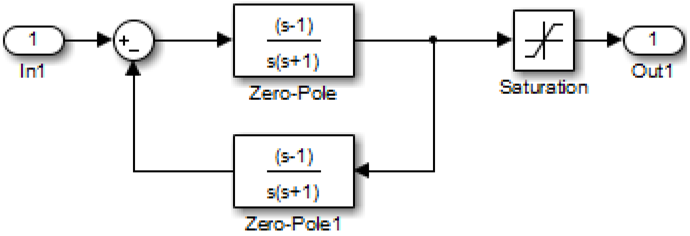
For more information, see “Bumpless Control Transfer” on page 14-111.

Multiloop control

Use signal tracking to prevent block windup in multiloop control approaches, as in the following model.



The Inner Loop subsystem contains the blocks shown in the following diagram.



Because the PID controller tracks the output of the inner loop, its output never exceeds the saturated inner-loop output. For more details, see "Prevent Block Windup in Multiloop Control" on page 14-110.

Dependencies

To enable this parameter, set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `TrackingMode`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Tracking coefficient (Kt)` — Gain of signal-tracking feedback loop

1 (default) | scalar

When you select **Enable tracking mode**, the difference between the signal **TR** and the block output is fed back to the integrator input with a gain `Kt`. Use this parameter to specify the gain in that feedback loop.

Dependencies

To enable this parameter, select **Enable tracking mode**.

Programmatic Use

Block Parameter: `Kt`
Type: scalar
Default: 1

Output saturation

`Limit Output` — Limit block output to specified saturation values

`off` (default) | `on`

Activating this option limits the block output internally to the block, so that you do not need a separate Saturation on page 1-1607 block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the **Anti-windup method** parameter). Specify the saturation limits using the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use

Block Parameter: `LimitOutput`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Upper limit` — Upper saturation limit for block output

`Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use

Block Parameter: `UpperSaturationLimit`
Type: scalar
Default: `Inf`

`Lower limit` — Lower saturation limit for block output

`-Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use

Block Parameter: `LowerSaturationLimit`
Type: scalar
Default: `-Inf`

`Ignore saturation when linearizing` — Force linearization to ignore output limits

`off` (default) | `on`

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the **Upper limit** and **Lower limit** parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use

Block Parameter: `LinearizeAsGain`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Anti-windup method` — Integrator anti-windup method

`none` (default) | `back-calculation` | `clamping`

When you select **Limit output** 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 windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, 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 unwind the integrator and return the weighted sum within the block saturation limit.

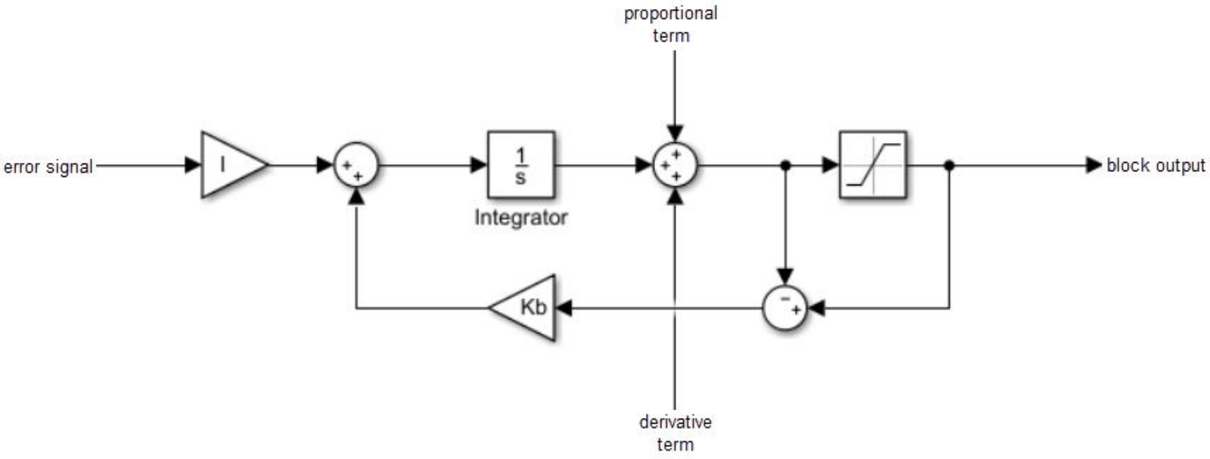
In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.

`none`

Do not use an anti-windup mechanism.

`back-calculation`

Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select **Mask > Look Under Mask**.



Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set `Kb = I`, or for controllers with derivative action, `Kb = sqrt(I*D)`. Back-calculation can be effective for plants with relatively large dead time [1].

`clamping`

Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use
Block Parameter: `AntiWindupMode`
Type: string, character vector
Values: `"none"`, `"back-calculation"`,`"clamping"`
Default: `"none"`

`Back-calculation coefficient (Kb)` — Gain coefficient of anti-windup feedback loop

1 (default) | scalar

The `back-calculation` anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. For more information, see the **Anti-windup method** parameter.

Dependencies

To enable this parameter, select the **Limit output** parameter, and set the **Anti-windup method** parameter to `back-calculation`.

Programmatic Use
Block Parameter: `Kb`
Type: scalar

Default: 1

Data Types

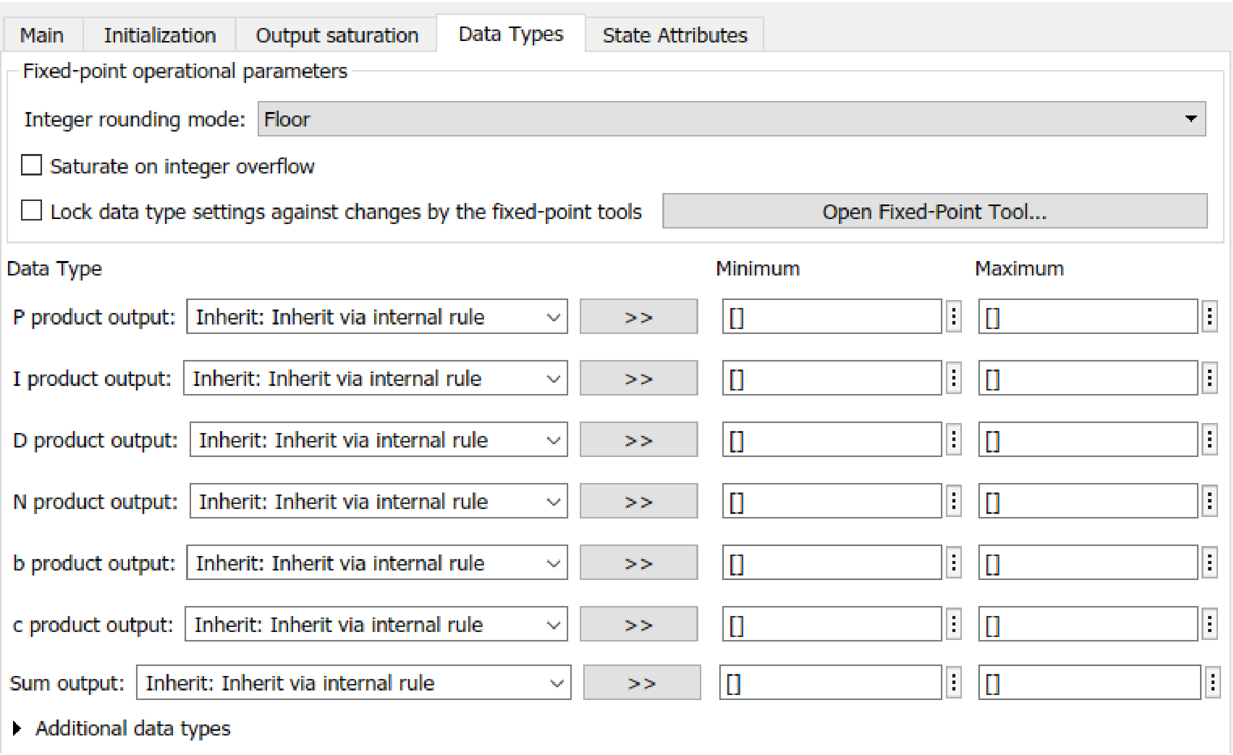
The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click **Open Fixed-Point Tool** and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).

After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.

For assistance in selecting appropriate values, click >> to open the Data Type Assistant for the corresponding quantity. For more information, see “Specify Data Types Using Data Type Assistant”.



Fixed-point operational parameters

Integer rounding mode: Floor

□ Saturate on integer overflow

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

Data Type		Minimum	Maximum
P product output:	Inherit: Inherit via internal rule	[]	[]
I product output:	Inherit: Inherit via internal rule	[]	[]
D product output:	Inherit: Inherit via internal rule	[]	[]
N product output:	Inherit: Inherit via internal rule	[]	[]
b product output:	Inherit: Inherit via internal rule	[]	[]
c product output:	Inherit: Inherit via internal rule	[]	[]
Sum output:	Inherit: Inherit via internal rule	[]	[]

The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- Product output — Stores the result of a multiplication carried out under the block mask. For example, **P product output** stores the output of the gain block that multiplies the block input with the proportional gain **P**.
- Parameter — Stores the value of a numeric block parameter, such as **P**, **I**, or **D**.
- Block output — Stores the output of a block that resides under the PID controller block mask. For example, use **Integrator output** to specify the data type of the output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.
- Accumulator — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block

SumI2. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as `SumI2`. (See “Search and Edit Using Model Explorer”.)

Matching Input and Internal Data Types

By default, all data types in the block are set to `Inherit: Inherit via internal rule`. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type `double`. If the block input signal is a type that cannot be converted to `double`, such as `uint16`, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to `double`, ensuring that the signals reaching the continuous-time integrators are of type `double`.

In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

Fixed-Point Operational Parameters

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector

Values: `'off' | 'on'`
Default: `'off'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

The parameters in this tab are primarily of use in code generation.

`State name (e.g., 'position')` — Name for continuous-time filter and integrator states

`''` (default) | character vector

Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the **State name** parameter.) The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

Dependencies

To enable this parameter, set **Time domain** to `Continuous-time`.

Programmatic Use

Parameter: `IntegratorContinuousStateAttributes`, `FilterContinuousStateAttributes`
Type: character vector
Default: `''`

`State name` — Names for discrete-time filter and integrator states

empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the **State name (e.g., 'position')** parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use

Parameter: `IntegratorStateIdentifier`, `FilterStateIdentifier`
Type: string, character vector
Default: `""`

`State name must resolve to Simulink signal object` — Require that state name resolve to a signal object

`off` (default) | `on`

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class** for the corresponding integrator or filter state.

Programmatic Use

Block Parameter: `IntegratorStateMustResolveToSignalObject`, `FilterStateMustResolveToSignalObject`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Code generation storage class` — Storage class for code generation

`Auto` (default) | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer`

Select state storage class for code generation. If you do not need to interface to external code, select `Auto`.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Programmatic Use

Block Parameter: `IntegratorRTWStateStorageClass`, `FilterRTWStateStorageClass`
Type: string, character vector
Values: `"Auto"`, `"ExportedGlobal"`, `"ImportedExtern"` | `"ImportedExternPointer"`
Default: `"Auto"`

Code generation storage type qualifier — Storage type qualifier

empty string (default) | character vector | "const" | "volatile" | ...

Specify a storage type qualifier such as `const` or `volatile`.

> **Note** This parameter 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.

Dependencies

To enable this parameter, set **Code generation storage class** to any value other than `Auto`.

Programmatic Use

Block Parameter: `IntegratorRTWStateStorageTypeQualifier`,`FilterRTWStateStorageTypeQualifier`
Type: string, character vector
Values:`""`,`"const"`,`"volatile"`
Default: `""`

Block Characteristics

Data Types	double | single | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

References

[1] Visioli, A., "Modified Anti-Windup Scheme for PID Controllers," *IEE Proceedings - Control Theory and Applications*, Vol. 150, Number 1, January 2003

Extended Capabilities

C/C++ Code Generation

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

For continuous-time PID controllers (**Time domain** set to `Continuous-time`):

- Consider using “Model Discretizer” to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, from your model, select **Analysis** > **Control Design** > **Analysis** > **Model Discretizer**.
- Not recommended for production code.

For discrete-time PID controllers (**Time domain** set to `Discrete-time`):

- Depends on absolute time when placed inside a triggered subsystem hierarchy.
- Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

HDL Code Generation

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

HDL code generation is supported for discrete-time PID controllers only (**Time domain** set to `Discrete-time`).

If the **Integrator method** is set to `BackwardEuler` or `Trapezoidal`, you cannot generate HDL code for the block under either of the following conditions:

- **Limit output** is selected and the **Anti-Windup Method** is anything other than `none`.
- **Enable tracking mode** is selected.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (**Time domain** set to `Discrete-time`).

See Also

Discrete Derivative | Discrete PID Controller (2DOF) | Discrete-Time Integrator | Gain | PID Controller

Introduced in R2009b

Discrete PID Controller (2DOF)

Discrete-time or continuous-time two-degree-of-freedom PID controller
Library: Simulink / Discrete

Description

The Discrete PID Controller (2DOF) block implements a two-degree-of-freedom PID controller (PID, PI, or PD). The block is identical to the PID Controller (2DOF) block with the **Time domain** parameter set to `Discrete-time`.

The 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 the proportional and derivative actions according to the setpoint weights (**b** and **c**) that 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 **P**, **I**, and **D**. A first-order pole filters the derivative action.

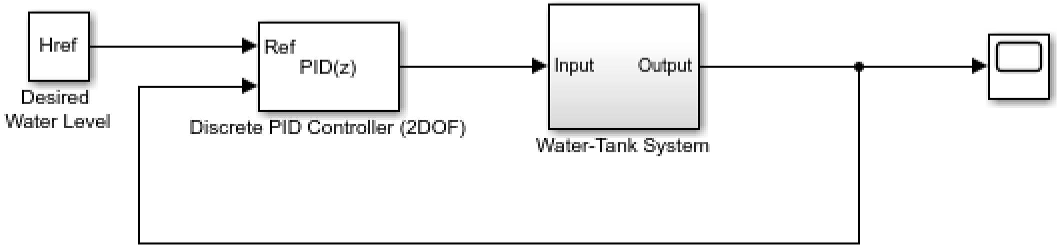
The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, or PD) — See the **Controller** parameter.
- Controller form (Parallel or Ideal) — See the **Form** parameter.
- Time domain (discrete or continuous) — See the **Time domain** parameter.
- Initial conditions and reset trigger — See the **Source** and **External reset** parameters.
- Output saturation limits and built-in anti-windup mechanism — See the **Limit output** parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the **Enable tracking mode** parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (See “Variant Subsystems”.) To examine the internal structure of the block and its variant subsystems, right-click the block and select **Mask > Look Under Mask**.

Control Configuration

In one common implementation, the PID Controller block operates in the feedforward path of a feedback loop.



For a single-input block that accepts an error signal (a difference between a setpoint and a system output), see Discrete PID Controller.

PID Gain Tuning

The PID controller coefficients and the setpoint weights are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software. For more information about automatic tuning, see the **Select tuning method** parameter.

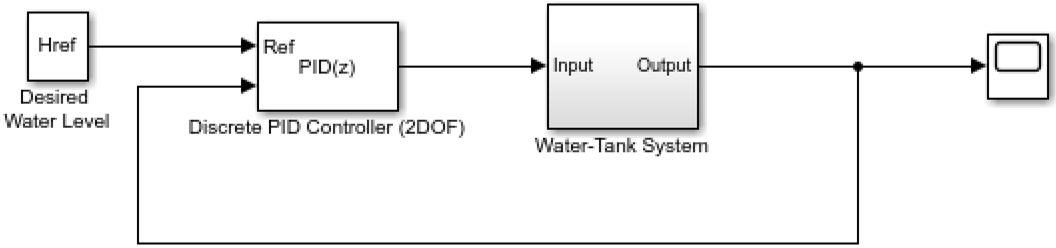
Ports

Input

`Ref` — Reference signal

scalar | vector

Reference signal for plant to follow, as shown.



When the reference signal is a vector, the block acts separately on each signal, vectorizing the PID coefficients and producing a vector output signal of the same dimensions. You can specify the PID coefficients and some other parameters as vectors of the same dimensions as the input signal. Doing so is equivalent to specifying a separate PID controller for each entry in the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_1( y )` — Measured system output

scalar | vector

Feedback signal for the controller, from the plant output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

P — Proportional gain

scalar | vector

Proportional gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`I` — Integral gain

scalar | vector

Integral gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`D` — Derivative gain

scalar | vector

Derivative gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`N` — Filter coefficient

scalar | vector

Derivative filter coefficient, provided from a source external to the block. External coefficient 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has a filtered derivative.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

b — Proportional setpoint weight

scalar | vector

Proportional setpoint weight, provided from a source external to the block. External 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

c — Derivative setpoint weight

scalar | vector

Derivative setpoint weight, provided from a source external to the block. External 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

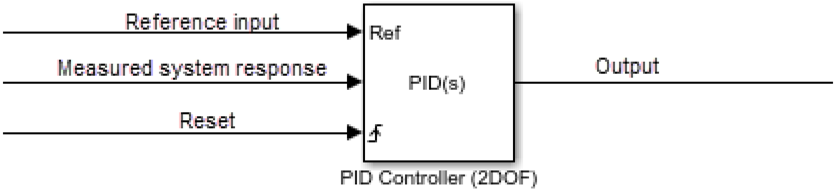
To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Reset — External reset trigger

scalar

Trigger to reset the integrator and filter to their initial conditions. Use the **External reset** parameter to specify what kind of signal triggers a reset. The port icon indicates the trigger type specified in that parameter. For example, the following illustration shows a continuous-time PID Controller (2DOF) block with **External reset** set to `rising`.



When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the **Integrator Initial condition** and **Filter Initial condition** parameters or the $\mathbf{I_0}$ and $\mathbf{D_0}$ ports.

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.

Dependencies

To enable this port, set **External reset** to any value other than `none`.

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

$\mathbf{I_0}$ — Integrator initial condition

scalar | vector

Integrator initial condition, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

D_0 — Filter initial condition

scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

TR — Tracking signal

scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input. Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the **Enable tracking mode** parameter.

Dependencies

To enable this port, select the **Enable tracking mode** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1( u )` — Controller output

scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the setpoint weights and by the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the **Controller** parameter. The base controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the **Upper Limit** and **Lower Limit** saturation parameters.

The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as *N* independent PID controllers, where *N* is the number of signals in the input vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Controller` — Controller type

`PID` (default) | `PI` | `PD`

Specify which of the proportional, integral, and derivative terms are in the controller.

`PID`

Proportional, integral, and derivative action.

`PI`

Proportional and integral action only.

`PD`

Proportional and derivative action only.

Tip The controller output for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use

Block Parameter: `Controller`
Type: string, character vector
Values: `"PID"`, `"PI"`, `"PD"`
Default: `"PID"`

Form — Controller structure

Parallel (default) | Ideal

Specify whether the controller structure is parallel or ideal.

Parallel

The proportional, integral, and derivative gains **P**, **I**, and **D**, are applied independently. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output *u* is:

$$u = P(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y),$$

where *r* is the reference signal, *y* is the measured plant output signal, and *b* and *c* are the setpoint weights.

For a discrete-time 2-DOF controller in parallel form, the controller output is:

$$u = P(br - y) + I\alpha(z)(r - y) + D\frac{N}{1 + N\beta(z)}(cr - y),$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Ideal

The proportional gain **P** acts on the sum of all actions. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$u = P\left[(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y)\right].$$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:

$$u = P\left[(br - y) + I\alpha(z)(r - y) + D\frac{N}{1 + N\beta(z)}(cr - y)\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Tip The controller output for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use
Block Parameter: `Controller`
Type: string, character vector
Values: `"Parallel"`, `"Ideal"`
Default: `"Parallel"`

`Time domain` — Specify discrete-time or continuous-time controller

`Discrete-time` (default) | `Continuous-time`

When you select `Discrete-time`, it is recommended that you specify an explicit sample time for the block. See the **Sample time (-1 for inherited)** parameter. Selecting `Discrete-time` also enables the **Integrator method**, and **Filter method** parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select `Continuous-time`.

Note The PID Controller (2DOF) and Discrete PID Controller (2DOF) blocks are identical except for the default value of this parameter.

Programmatic Use
Block Parameter: `TimeDomain`
Type: string, character vector
Values: `"Continuous-time"`, `"Discrete-time"`
Default: `"Discrete-time"`

`Sample time (-1 for inherited)` — Discrete interval between samples

–1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of –1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the

controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.

To implement a continuous-time controller, set **Time domain** to `Continuous-time`.

Tip If you want to run the block with an externally specified or variable sample time, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use
Block Parameter: `SampleTime`
Type: scalar
Values: `-1`, positive scalar
Default: `-1`

`Integrator method` — Method for computing integral in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the integral term of the controller transfer function is *Ia*(*z*), where *a*(*z*) depends on the integrator method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z) = \frac{T_s}{z-1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z) = \frac{T_s z}{z - 1}.$$

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`

Bilinear approximation,

$$\alpha(z) = \frac{T_s}{2} \frac{z + 1}{z - 1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with integral action.

Programmatic Use
Block Parameter: `IntegratorMethod`
Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

`Filter method` — Method for computing derivative in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the derivative term of the controller transfer function is:

$$D\left[\frac{N}{1+N\alpha(z)}\right],$$

where $\alpha(z)$ depends on the filter method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z)=\frac{T_s}{z-1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z)=\frac{T_s z}{z-1}.$$

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`

Bilinear approximation,

$$\alpha(z)=\frac{T_s}{2}\frac{z+1}{z-1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with derivative action.

Programmatic Use

Block Parameter: `FilterMethod`
Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

Main

`Source` — Source for controller gains and filter coefficient

internal (default) | external

`internal`

Specify the controller gains, filter coefficient, and setpoint weights using the block parameters **P**, **I**, **D**, **N**, **b**, and **c** respectively.

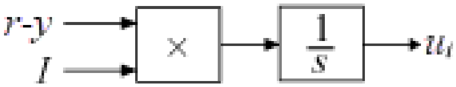
`external`

Specify the PID gains, filter coefficient, and setpoint weights externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

Enabling external inputs for the parameters allows you to compute their values externally to the block and provide them to the block as signal inputs.

External 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 gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. The derivative setpoint weight *c* is also differentiated. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. 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 input signal *u* is multiplied by the externally supplied integrator gain, *I*, before integration. This implementation yields:

$$u_i = \int (r - y) I \, dt.$$

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* and the derivative setpoint weight *c* to be differentiated.

Programmatic Use
Block Parameter: `ControllerParametersSource`
Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Proportional (P)` — Proportional gain

1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- `Parallel` — Proportional action is independent of the integral and derivative actions. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output *u* is:

  $$u = P(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y),$$

  where *r* is the reference signal, *y* is the measured plant output signal, and *b* and *c* are the setpoint weights.

  For a discrete-time 2-DOF controller in parallel form, the controller output is:

  $$u = P(br - y) + I\alpha(z)(r - y) + D\frac{N}{1 + N\beta(z)}(cr - y),$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

- `Ideal` — The proportional gain multiples the integral and derivative terms. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$u = P\left[(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y)\right].$$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:

$$u = P\left[(br - y) + I\alpha(z)(r - y) + D\frac{N}{1 + N\beta(z)}(cr - y)\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Tunable: Yes

Dependencies

To enable this parameter, set the Controller parameters **Source** to `internal`.

Programmatic Use

Block Parameter: `P`
Type: scalar, vector
Default: 1

`Integral (I)` — Integral gain

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use
Block Parameter: `I`
Type: scalar, vector
Default: 1

`Derivative (D)` — Derivative gain

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to `PID` or `PD`.

Programmatic Use
Block Parameter: `D`
Type: scalar, vector
Default: 0

`Use filtered derivative` — Apply filter to derivative term

`on` (default) | `off`

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller output becomes:

$$D\frac{z-1}{zT_s}(cr-y).$$

For continuous-time PID controllers, the derivative term is always filtered.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`, and set **Controller** to a type that has a derivative term.

Programmatic Use
Block Parameter: `UseFilter`
Type: string, character vector
Values: `"on"`, `"off"`

Default: `"on"`

`Filter coefficient (N)` — Derivative filter coefficient

100 (default) | scalar | vector

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the **Time domain** parameter.

- When **Time domain** is `Continuous-time`, the pole location is `s = -N`.
- When **Time domain** is `Discrete-time`, the pole location depends on the **Filter method** parameter.

Filter Method	Location of Filter Pole
`Forward Euler`	$z_{pole} = 1 - NT_s$
`Backward Euler`	$z_{pole} = \frac{1}{1 + NT_s}$
`Trapezoidal`	$z_{pole} = \frac{1 - NT_s / 2}{1 + NT_s / 2}$

The block does not support `N = Inf` (ideal unfiltered derivative). When the **Time domain** is `Discrete-time`, you can clear **Use filtered derivative** to remove the derivative filter.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to `PID` or `PD`.

Programmatic Use

Block Parameter: `N`
Type: scalar, vector
Default: 100

`Setpoint weight (b)` — Proportional setpoint weight

1 (default) | scalar | vector

Setpoint weight on the proportional term of the controller. The proportional term of a 2-DOF controller output is *P*(*br*–*y*), where *r* is the reference signal and *y* is the measured plant output. Setting *b* to 0 eliminates proportional action on the reference signal, which can reduce overshoot in the system response to step changes in the setpoint. Changing the relative values of *b* and *c* changes the balance between disturbance rejection and setpoint tracking.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`.

Programmatic Use
Block Parameter: `b`
Type: scalar, vector
Default: 1

`Setpoint weight (c)` — Derivative setpoint weight

1 (default) | scalar | vector

Setpoint weight on the derivative term of the controller. The derivative term of a 2-DOF controller acts on *cr*–*y*, where *r* is the reference signal and *y* is the measured plant output. Thus, setting *c* to 0 eliminates derivative action on the reference signal, which can reduce transient response to step changes in the setpoint. Setting *c* to 0 can yield a controller that achieves both effective disturbance rejection and smooth setpoint tracking without excessive transient response. Changing the relative values of *b* and *c* changes the balance between disturbance rejection and setpoint tracking.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to a type that has derivative action.

Programmatic Use
Block Parameter: `c`
Type: scalar, vector
Default: 1

`Select tuning method` — Tool for automatic tuning of controller coefficients

`Transfer Function Based (PID Tuner App)` (default) | `Frequency Response Based`

If you have Simulink Control Design software, you can automatically tune the PID coefficients when they are internal to the block. To do so, use this parameter to select a tuning tool, and click **Tune**.

`Transfer Function Based (PID Tuner App)`

> Use **PID Tuner**, which lets you interactively tune PID coefficients while examining relevant system responses to validate performance. **PID Tuner** can tune all the coefficients **P**, **I**, **D**, and **N**, and the setpoint coefficients **b** and **c**. By default, **PID Tuner** works with a linearization of your plant model. For models that cannot be linearized, you can tune PID coefficients against a plant model estimated from simulated or measured response data. For more information, see “Design Two-Degree-of-Freedom PID Controllers” (Simulink Control Design).

`Frequency Response Based`

> Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to zero. **Frequency Response Based PID Tuner** tunes the coefficients **P**, **I**, **D**, and **N**, but does not tune the setpoint coefficients **b** and **c**. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`.

`Enable zero-crossing detection` — Detect zero crossings on reset and on entering or leaving a saturation state

`on` (default) | `off`

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 **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

Programmatic Use
Block Parameter: `ZeroCross`
Type: string, character vector
Values: `"on"`, `"off"`
Default: `"on"`

Initialization

`Source` — Source for integrator and derivative initial conditions

`internal` (default) | `external`

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event. (See the **External reset** parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.

`internal`

Specify the initial conditions using the **Integrator Initial condition** and **Filter Initial condition** parameters. If **Use filtered derivative** is not selected, use the **Differentiator** parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

`external`

Specify the initial conditions externally using block inputs. Additional input ports $\mathbf{I_o}$ and $\mathbf{D_o}$ appear on the block. If **Use filtered derivative** is not selected, supply the initial condition for the unfiltered differentiator at $\mathbf{D_o}$ instead of a filter initial condition.

Programmatic Use
Block Parameter: `InitialConditionSource`

Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Integrator` — Integrator initial condition

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `InitialConditionForIntegrator`
Type: scalar, vector
Default: 0

`Filter` — Filter initial condition

0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and use a controller that has a derivative filter.

Programmatic Use

Block Parameter: `InitialConditionForFilter`
Type: scalar, vector
Default: 0

`Differentiator` — Initial condition for unfiltered derivative

0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, set **Time domain** to `Discrete-time`, clear the **Use filtered derivative** check box, and in the **Initialization** tab, set **Source** to `internal`.

Programmatic Use
Block Parameter: `DifferentiatorICPrevScaledInput`
Type: scalar, vector
Default: 0

`Initial condition setting` — Location at which initial condition is applied

`State (most efficient)` (default) | `Output`

Use this parameter to specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. You can change this parameter at the command line only, using `set_param` to set the `InitialConditionSetting` parameter of the block.

`State (most efficient)`

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** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

Programmatic Use
Block Parameter: `InitialConditionSetting`

Type: string, character vector
Values: "state", "output"
Default: "state"

External reset — Trigger for resetting integrator and filter values

none (default) | rising | falling | either | level

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than none enables the **Reset** port on the block for the external reset signal.

none

The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

rising

Reset the outputs when the reset signal has a rising edge.

falling

Reset the outputs when the reset signal has a falling edge.

either

Reset the outputs when the reset signal either rises or falls.

level

Reset the outputs when the reset signal either:

- Is nonzero at the current time step
- Changes from nonzero at the previous time step to zero at the current time step

This option holds the outputs to the initial conditions while the reset signal is nonzero.

Dependencies

To enable this parameter, set **Controller** to a type that has derivative or integral action.

Programmatic Use

Block Parameter: ExternalReset
Type: string, character vector
Values: "none", "rising", "falling", "either","level"
Default: "none"

`Ignore reset when linearizing` — Force linearization to ignore reset

`off` (default) | `on`

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the **External reset** parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

Programmatic Use

Block Parameter: `IgnoreLimit`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

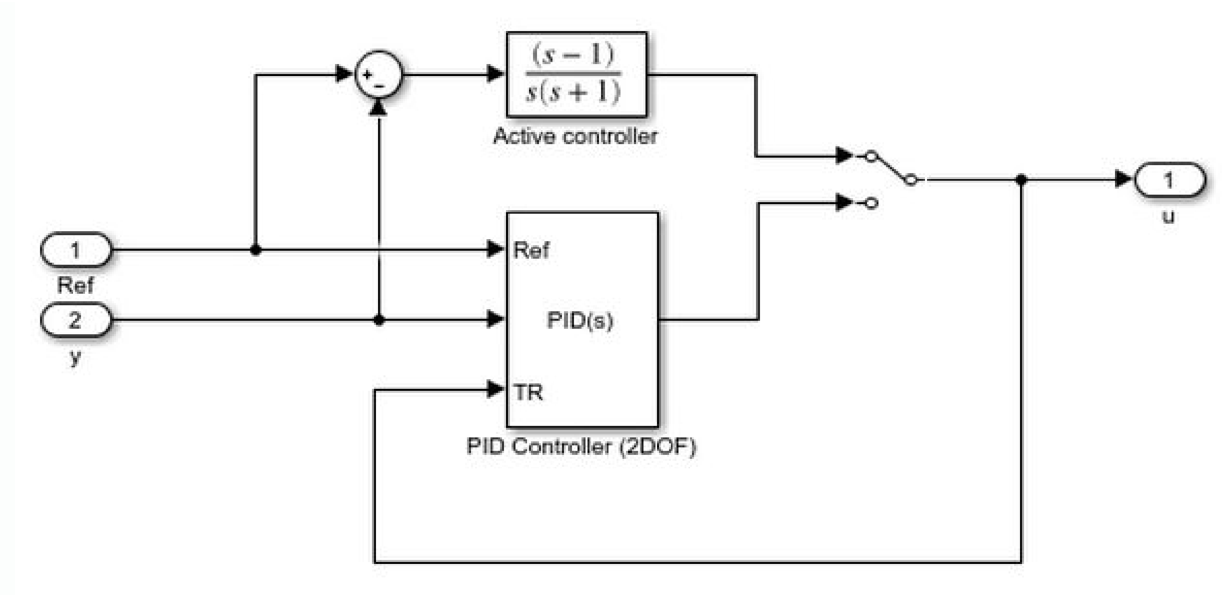
`Enable tracking mode` — Activate signal tracking

`off` (default) | `on`

Signal tracking lets the block output follow a tracking signal that you provide at the **TR** port. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input with a gain `Kt`, specified by the **Tracking gain (Kt)** parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.

Bumpless control transfer

Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and another controller. To do so, connecting the controller output to the **TR** input as shown in the following illustration.



For more information, see “Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller” on page 14-112.

Multiloop control

Use signal tracking to prevent block windup in multiloop control approaches. For an example illustrating this approach with a 1DOF PID controller, see “Prevent Block Windup in Multiloop Control” on page 14-110.

Dependencies

To enable this parameter, set **Controller** to a type that has integral action.

Programmatic Use
Block Parameter: `TrackingMode`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Tracking coefficient (Kt)` — Gain of signal-tracking feedback loop

1 (default) | scalar

When you select **Enable tracking mode**, the difference between the signal **TR** and the block output is fed back to the integrator input with a gain `Kt`. Use this parameter to specify the gain in that feedback loop.

Dependencies

To enable this parameter, select **Enable tracking mode**.

Programmatic Use
Block Parameter: `Kt`
Type: scalar
Default: 1

Output saturation

`Limit Output` — Limit block output to specified saturation values

`off` (default) | `on`

Activating this option limits the block output internally to the block, so that you do not need a separate Saturation on page 1-1607 block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the **Anti-windup method** parameter). Specify the saturation limits using the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use
Block Parameter: `LimitOutput`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Upper limit` — Upper saturation limit for block output

`Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use
Block Parameter: `UpperSaturationLimit`

Type: scalar
Default: `Inf`

`Lower limit` — Lower saturation limit for block output

`-Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use

Block Parameter: `LowerSaturationLimit`
Type: scalar
Default: `-Inf`

`Ignore saturation when linearizing` — Force linearization to ignore output limits

`off` (default) | `on`

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the **Upper limit** and **Lower limit** parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use

Block Parameter: `LinearizeAsGain`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Anti-windup method` — Integrator anti-windup method

`none` (default) | `back-calculation` | `clamping`

When you select **Limit output** 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 windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, two results are possible:

- If the sign of the signal entering the integrator 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 signal entering the integrator changes once the weighted sum has grown beyond the output limits, it can take a long time to unwind the integrator and return the weighted sum within the block saturation limit.

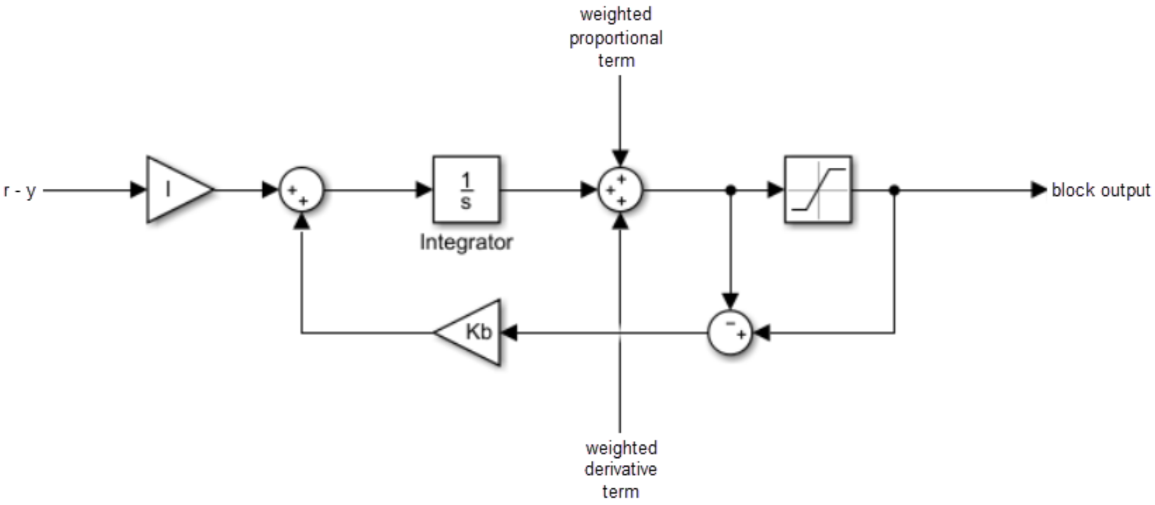
In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.

`none`

Do not use an anti-windup mechanism.

`back-calculation`

Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select **Mask > Look Under Mask**.



Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set `Kb = I`, or for controllers with derivative action, `Kb = sqrt(I*D)`. Back-calculation can be effective for plants with relatively large dead time [1].

`clamping`

Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use
Block Parameter: `AntiWindupMode`
Type: string, character vector
Values: `"none"`, `"back-calculation"`,`"clamping"`

Default: `"none"`

`Back-calculation coefficient (Kb)` — Gain coefficient of anti-windup feedback loop

1 (default) | scalar

The `back-calculation` anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. For more information, see the **Anti-windup method** parameter.

Dependencies

To enable this parameter, select the **Limit output** parameter, and set the **Anti-windup method** parameter to `back-calculation`.

Programmatic Use

Block Parameter: `Kb`
Type: scalar
Default: 1

Data Types

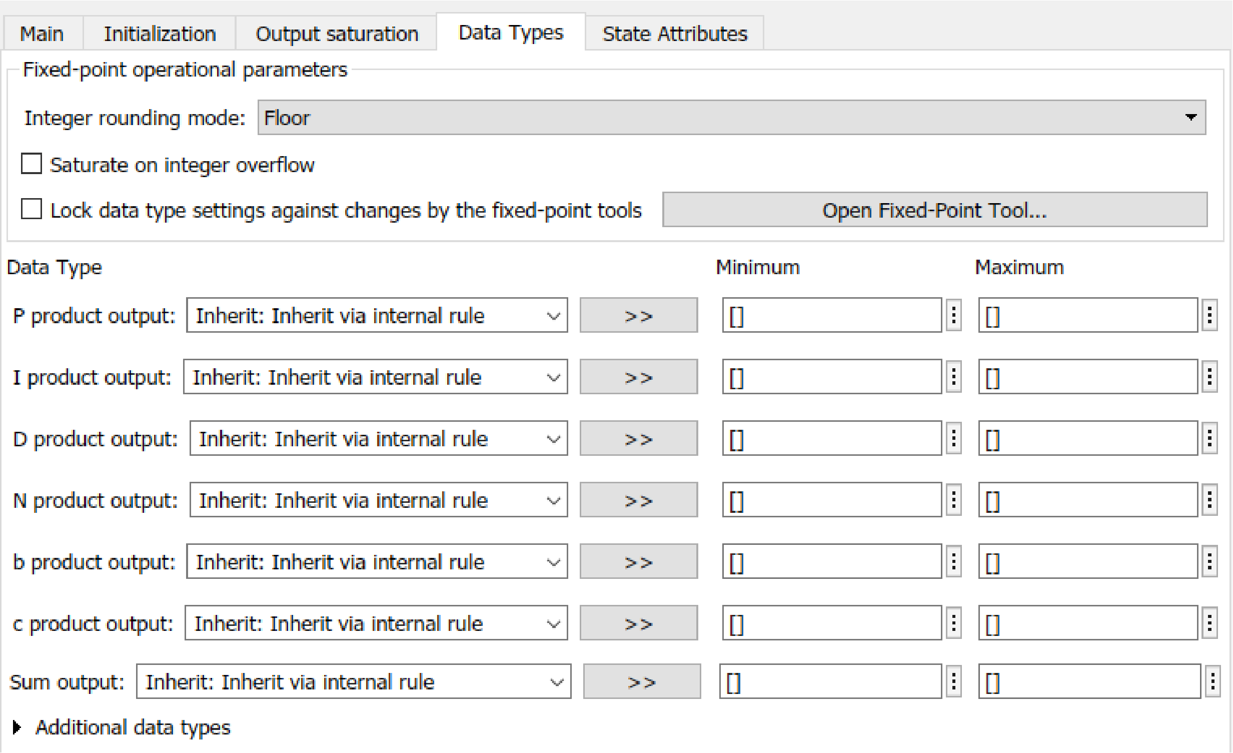
The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click **Open Fixed-Point Tool** and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).

After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.

For assistance in selecting appropriate values, click >> to open the Data Type Assistant for the corresponding quantity. For more information, see “Specify Data Types Using Data Type Assistant”.



Fixed-point operational parameters

Integer rounding mode: Floor

□ Saturate on integer overflow

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

Data Type			Minimum	Maximum
P product output:	Inherit: Inherit via internal rule	>>	[]	[]
I product output:	Inherit: Inherit via internal rule	>>	[]	[]
D product output:	Inherit: Inherit via internal rule	>>	[]	[]
N product output:	Inherit: Inherit via internal rule	>>	[]	[]
b product output:	Inherit: Inherit via internal rule	>>	[]	[]
c product output:	Inherit: Inherit via internal rule	>>	[]	[]
Sum output:	Inherit: Inherit via internal rule	>>	[]	[]

The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- Product output — Stores the result of a multiplication carried out under the block mask. For example, **P product output** stores the output of the gain block that multiplies the block input with the proportional gain **P**.
- Parameter — Stores the value of a numeric block parameter, such as **P**, **I**, or **D**.
- Block output — Stores the output of a block that resides under the PID controller block mask. For example, use **Integrator output** to specify the data type of the

output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.

- Accumulator — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block SumI2. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as `SumI2`. (See “Search and Edit Using Model Explorer”.)

Matching Input and Internal Data Types

By default, all data types in the block are set to `Inherit: Inherit via internal rule`. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type `double`. If the block input signal is a type that cannot be converted to `double`, such as `uint16`, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to `double`, ensuring that the signals reaching the continuous-time integrators are of type `double`.

In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

Fixed-Point Operational Parameters

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations. For more information, see "Rounding" (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see "Lock the Output Data Type Setting" (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

The parameters in this tab are primarily of use in code generation.

`State name (e.g., 'position')` — Name for continuous-time filter and integrator states

`''` (default) | character vector

Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the **State name** parameter.) The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

Dependencies

To enable this parameter, set **Time domain** to `Continuous-time`.

Programmatic Use
Parameter: `IntegratorContinuousStateAttributes`, `FilterContinuousStateAttributes`
Type: character vector
Default: `''`

`State name` — Names for discrete-time filter and integrator states

empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the **State name (e.g., 'position')** parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use
Parameter: `IntegratorStateIdentifier`, `FilterStateIdentifier`
Type: string, character vector
Default: `""`

`State name must resolve to Simulink signal object` — Require that state name resolve to a signal object

`off` (default) | `on`

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class** for the corresponding integrator or filter state.

Programmatic Use
Block Parameter: `IntegratorStateMustResolveToSignalObject`, `FilterStateMustResolveToSignalObject`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Code generation storage class` — Storage class for code generation

`Auto` (default) | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer`

Select state storage class for code generation. If you do not need to interface to external code, select `Auto`.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Programmatic Use
Block Parameter: `IntegratorRTWStateStorageClass`, `FilterRTWStateStorageClass`
Type: string, character vector
Values: `"Auto"`, `"ExportedGlobal"`, `"ImportedExtern"` | `"ImportedExternPointer"`
Default: `"Auto"`

Code generation storage type qualifier — Storage type qualifier

empty string (default) | character vector | "const" | "volatile" | ...

Specify a storage type qualifier such as `const` or `volatile`.

Note This parameter 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.

Dependencies

To enable this parameter, set **Code generation storage class** to any value other than `Auto`.

Programmatic Use
Block Parameter: `IntegratorRTWStateStorageTypeQualifier`,`FilterRTWStateStorageTypeQualifier`
Type: string, character vector
Values:`""`,`"const"`,`"volatile"`
Default: `""`

Block Characteristics

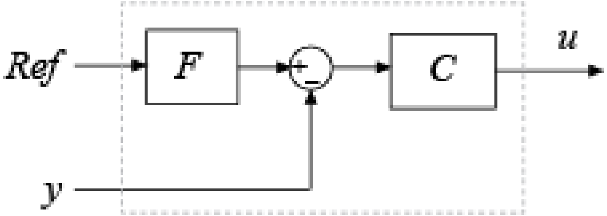
Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Definitions

Decomposition of 2-DOF PID Controllers

A 2-DOF PID controller can be interpreted as a PID controller with a prefilter, or a PID controller with a feedforward element.

In parallel form, a two-degree-of-freedom PID controller can be equivalently modeled by the following block diagram, where *C* is a single degree-of-freedom PID controller and *F* is a prefilter on the reference signal.



Ref is the reference signal, *y* is the feedback from the measured system output, and *u* is the controller output. For a continuous-time 2-DOF PID controller in parallel form, the transfer functions for *F* and *C* 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 weights.

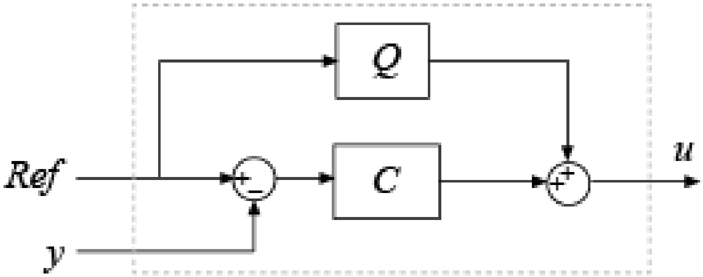
For a 2-DOF PID controller in ideal form, the transfer functions are

$$F_{id}(s) = \frac{(b + cDN)s^2 + (bN + I)s + IN}{(1 + DN)s^2 + (N + I)s + IN},$$

$$C_{id}(s) = P\,\frac{(1 + DN)s^2 + (N + I)s + IN}{s(s + N)}.$$

A similar decomposition applies for a discrete-time 2-DOF controller.

Alternatively, the parallel two-degree-of-freedom PID controller can be modeled by the following block diagram.



In this realization, *Q* acts as feed-forward conditioning on the reference signal. For a continuous-time 2-DOF PID controller in parallel form, the transfer function for *Q* is

$$Q_{par}(s) = \frac{\big((b-1)P + (c-1)DN\big)s + (b-1)PN}{s + N}.$$

For a 2-DOF PID controller in ideal form, the transfer function is

$$Q_{id}(s) = P\,\frac{\big((b-1) + (c-1)DN\big)s + (b-1)N}{s + N}.$$

The transfer functions for *C* are the same as in the filter decomposition.

A similar decomposition applies for a discrete-time 2-DOF controller.

References

[1] Visioli, A., "Modified Anti-Windup Scheme for PID Controllers," *IEE Proceedings - Control Theory and Applications*, Vol. 150, Number 1, January 2003

Extended Capabilities

C/C++ Code Generation

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

For continuous-time PID controllers (**Time domain** set to `Continuous-time`):

- Consider using “Model Discretizer” to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, from your model, select **Analysis** > **Control Design** > **Analysis** > **Model Discretizer**.
- Not recommended for production code.

For discrete-time PID controllers (**Time domain** set to `Discrete-time`):

- Depends on absolute time when placed inside a triggered subsystem hierarchy.
- Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (**Time domain** set to `Discrete-time`).

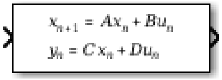
See Also

Discrete Derivative | Discrete PID Controller | Discrete-Time Integrator | Gain | PID Controller (2DOF)

Introduced in R2009b

Discrete State-Space

Implement discrete state-space system
Library: Simulink / Discrete



Description

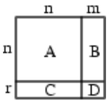
Block Behavior for Non-Empty Matrices

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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input vector, where the width equals the number of columns in the **B** and **D** matrices. For more information, see "Description" on page 1-576.

Tip For integer and fixed-point input signals, use the Fixed-Point State-Space block.

Data Types: `single` | `double`

Output

`Port_1` — Output vector
scalar | vector | matrix

Output vector, with width equal to the number of rows in the **C** and **D** matrices. For more information, see "Description" on page 1-576.

Data Types: `single` | `double`

Parameters

Main

`A` — Matrix coefficient A
`1` (default) | scalar | vector | matrix

Specify the matrix coefficient A, as a real-valued *n*-by-*n* matrix, where *n* is the number of states. For more information on the matrix coefficients, see “Description” on page 1-576.

Programmatic Use
Block Parameter: A
Type: character vector
Values: scalar | vector | matrix
Default: '1'

B — Matrix coefficient B

1 (default) | scalar | vector | matrix

Specify the matrix coefficient B, as a real-valued *n*-by-*m* matrix, where *n* is the number of states, and *m* is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-576.

Programmatic Use
Block Parameter: B
Type: character vector
Values: scalar | vector | matrix
Default: '1'

C — Matrix coefficient, C

1 (default) | scalar | vector | matrix

Specify the matrix coefficient **C**, as a real-valued *r*-by-*n* matrix, where *r* is the number of outputs, and *n* is the number of states. For more information on the matrix coefficients, see “Description” on page 1-576.

Programmatic Use
Block Parameter: C
Type: character vector
Values: scalar | vector | matrix
Default: '1'

D — Matrix coefficient, D

1 (default) | scalar | vector | matrix

Specify the matrix coefficient **D**, as a real-valued *r*-by-*m* matrix, where *r* is the number of outputs, and *m* is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-576.

Programmatic Use

Block Parameter: `D`
Type: character vector
Values: scalar | vector | matrix
Default: `'1'`

`Initial conditions` — Initial state vector

0 (default) | scalar | vector | matrix

Specify the initial state vector as a scalar, vector, or matrix. Simulink does not allow the initial states of this block to be `inf` or `NaN`.

Programmatic Use

Block Parameter: `InitialCondition`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`Sample time (–1 for inherited)` — Interval between samples

`-1` (default) | scalar | vector

Specify the time interval between samples. See “Specify Sample Time”.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: `StateName`
Type: character vector
Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, specify a value for **State name**. 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**.

Programmatic Use
Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default) | `<StorageClass.PackageName>`

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use
Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`
Default: `'Simulink.Signal'`

`Code generation storage class` — State storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `BitField (Custom)` | `Model default` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `AutoScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation.

- `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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

To enable this parameter, specify a value for **State name**.

Programmatic Use
Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto' | 'SimulinkGlobal' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

`TypeQualifier` — Storage type qualifier

`''` (default) | `const` | `volatile` | ...

Specify a storage type qualifier such as `const` or `volatile`.

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.

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 “Data Objects”.

Dependencies

To enable this parameter, set **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `Model default`. This parameter is hidden unless you previously set its value.

Programmatic Use

Block Parameter: `RTWStateStorageTypeQualifier`
Type: character vector
Values: `'' | 'const' | 'volatile' | ...`
Default: `''`

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No

Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Fixed-Point State-Space | State-Space

Topics

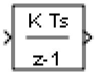
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder)
“Data Objects”

Introduced before R2006a

Discrete-Time Integrator

Perform discrete-time integration or accumulation of signal

Library: Simulink / Commonly Used Blocks
Simulink / Discrete



Description

Use the Discrete-Time Integrator block in place of the Integrator block to create a purely discrete model. 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 with an additional reset input

Output Equations

With the first time step, block state `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`.

Integration and Accumulation Methods

This block can integrate or accumulate a signal using a forward Euler, backward Euler, or trapezoidal method. 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

The software approximates `1/s` as `T/(z-1)`. The expressions for the output of the block at step `n` are:

```
x(n+1) = x(n) + K*T*u(n)
y(n)   = x(n)
```

The block uses these steps to compute the output:

```
Step 0:         y(0)   = IC (clip if necessary)
                x(1)   = y(0) + K*T*u(0)

Step 1:         y(1)   = x(1)
                x(2)   = x(1) + K*T*u(1)

Step n:         y(n)   = x(n)
                x(n+1) = x(n) + K*T*u(n) (clip if necessary)
```

Using this method, input port 1 does not have direct feedthrough.

Backward Euler method, also known as backward rectangular or right-hand approximation

The software approximates `1/s` as `T*z/(z-1)`. The resulting expression for the output of the block at step `n` is

```
y(n) = y(n-1) + K*T*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` or `Auto` for triggered and function-call subsystems:

  ```
  Step 0:          y(0) = IC (clipped if necessary)
                   x(1) = y(0)
  ```
- If the parameter **Initial condition setting** is set to `Auto` for non-triggered subsystems:

  ```
  Step 0:          x(0)   = IC (clipped if necessary)
                   x(1)   = y(0) = x(0) + K*T*u(0)

  Step 1:          y(1)   = x(1) + K*T*u(1)
                   x(2)   = y(1)

  Step n:          y(n)   = x(n) + K*T*u(n)
                   x(n+1) = y(n)
  ```

Using this method, input port 1 has direct feedthrough.

For this method, the software approximates `1/s` as `T/2*(z+1)/(z-1)`.

When `T` is fixed (equal to the sampling period), the expressions to compute the output are:

```
x(n) = y(n-1) + K*T/2*u(n-1)
y(n) = x(n) + K*T/2*u(n)
```

- If the parameter **Initial condition setting** is set to `Output` or `Auto` for triggered and function-call subsystems:

  ```
  Step 0:          y(0)   = IC (clipped if necessary)
                   x(1)   = y(0) + K*T/2*u(0)
  ```
- If the parameter **Initial condition setting** is set to `Auto` for non-triggered subsystems:

  ```
  Step 0:          x(0)   = IC (clipped if necessary)
                   y(0)   = x(0) + K*T/2*u(0)
  ```

```
                 x(1)   = y(0) + K*T/2*u(0)

Step 1:          y(1)   = x(1) + K*T/2*u(1)
                 x(2)   = y(1) + K*T/2*u(1)

Step n:          y(n)   = x(n) + K*T/2*u(n)
                 x(n+1) = y(n) + K*T/2*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)`.

Using this method, input port 1 has direct feedthrough.

When`T` is a variable (for example, obtained from the triggering times), the block uses these steps to compute the output.

- If the parameter **Initial condition setting** is set to `Output` or `Auto` for triggered and function-call subsystems:

```
Step 0:          y(0)   = IC (clipped if necessary)
                 x(1)   = y(0)
```

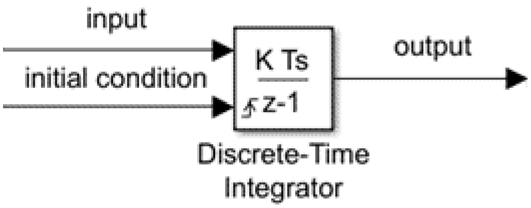
- If the parameter **Initial condition setting** is set to `Auto` for non-triggered subsystems:

```
Step 0:          x(0)   = IC (clipped if necessary)
                 x(1)   = y(0) = x(0) + K*T/2*u(0)

Step 1:          y(1)   = x(1) + T/2*(u(1) + u(0))
                 x(2)   = y(1)

Step n:          y(n)   = x(n) + T/2*(u(n) + u(n-1))
                 x(n+1) = y(n)
```

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.

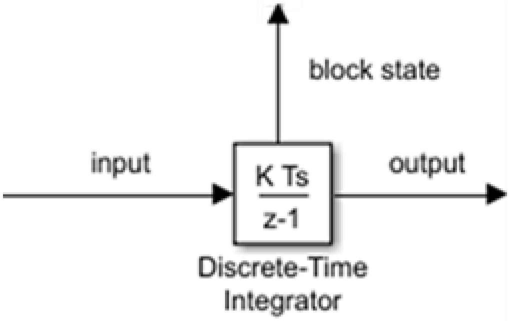


When to Use the State Port

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.

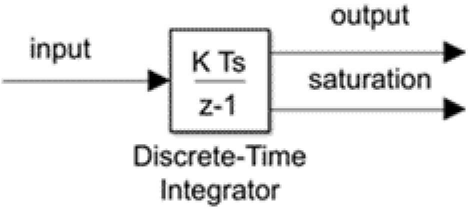


Limit the Integral

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.

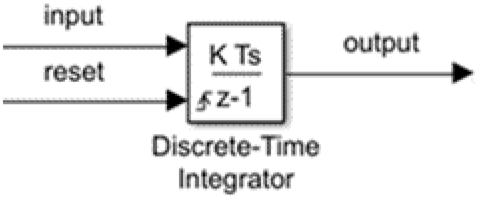


The saturation signal has one of three values:

- 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 resets 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** parameter options. A reset port appears that indicates the reset trigger type.

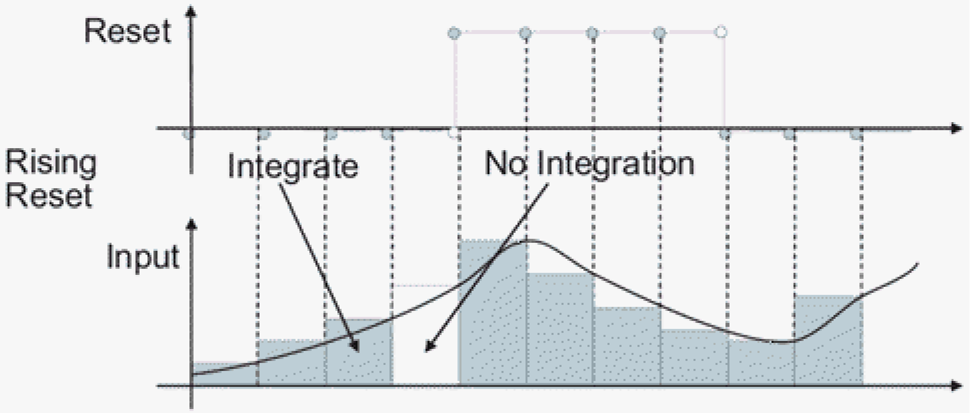


The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results. To resolve this loop, feed the output of the block state port into the reset port instead. To access the block state, select the **Show state port** check box.

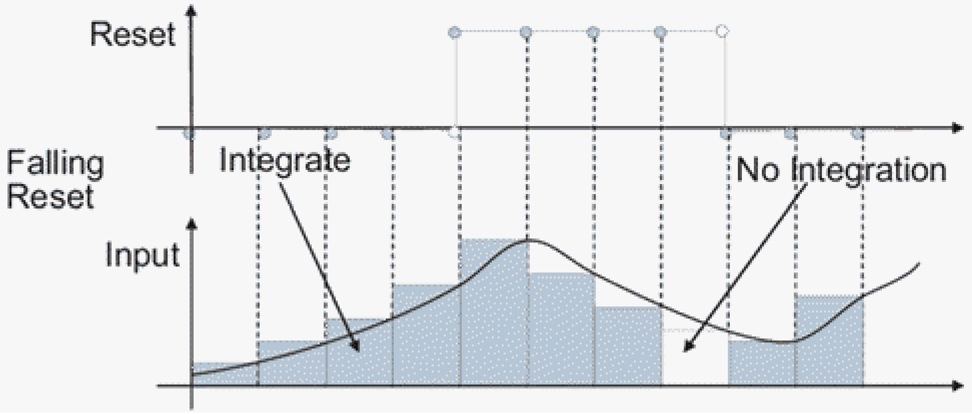
Reset Trigger Types

The **External reset** parameter lets you determine the attribute of the reset signal that triggers the reset. The trigger options include:

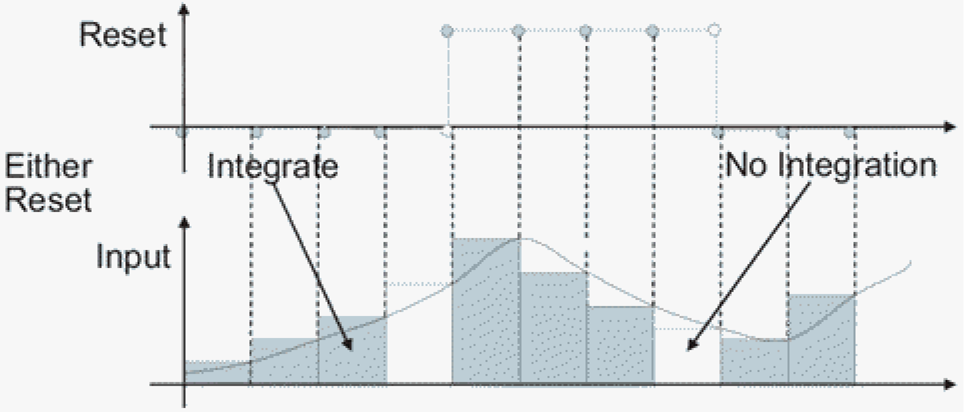
- `rising` – Resets the state when the reset signal has a rising edge. For example, this figure shows the effect that a rising reset trigger has on backward Euler integration.



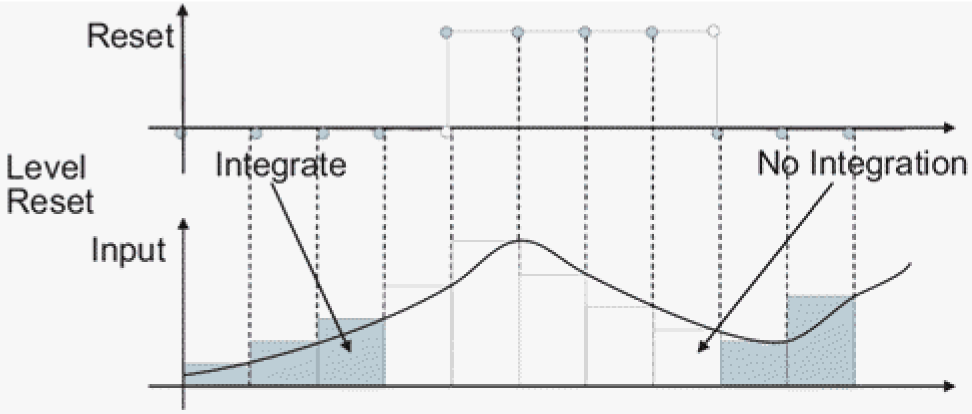
- `falling` — Resets the state when the reset signal has a falling edge. For example, this figure shows the effect that a falling reset trigger has on backward Euler integration.



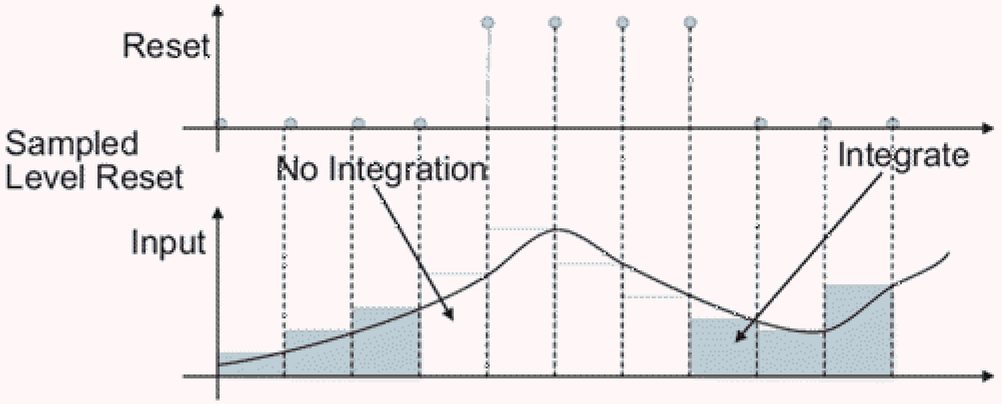
- `either` — Resets the state when the reset signal rises or falls. For example, the following figure shows the effect that an either reset trigger has on backward Euler integration.



- `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.

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` in 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` for triggered and function-call subsystems, the enable and disable behavior of the block is simplified as follows.

At disable time t_d:

$y(t_d) = y(t_d-1)$

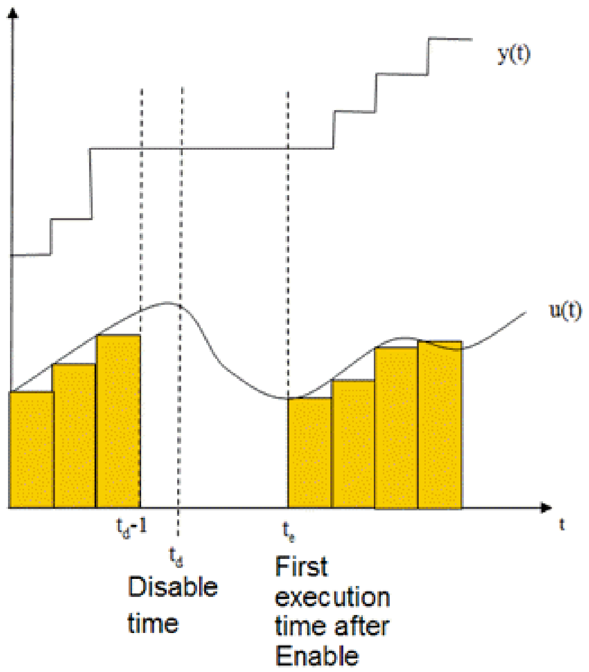
At enable time t_e:

- 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 block.

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.

Behavior in an Enabled Subsystem Inside a Function-Call Subsystem

Suppose you have a function-call subsystem that includes 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 `delta T` When Function-Call Subsystem Executes for the First Time After Enabled	Reason for Behavior
Forward Euler	Triggered	`t – tstart`	When the function-call subsystem executes for the first time, the integrator algorithm uses `tstart` as the previous simulation time.
Backward Euler and Trapezoidal	Triggered	`t – tprevious`	When the function-call subsystem executes for the first time, the integrator algorithm uses `tprevious` 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 `delta T`.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

IC — Initial conditions of the states

scalar | vector | matrix

Initial conditions of the states, specified as a finite scalar, vector, or matrix.

Dependencies

To enable this port, set **Initial condition source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

Port_1 — Discrete-time integration or accumulation of input

scalar | vector | matrix

Discrete-time integration or accumulation of the input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Port_2 — Saturation output

scalar | vector | matrix

Signal indicating when the state is being limited, specified as a scalar, vector, or matrix. 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.

Dependencies

To enable this port, select the **Show saturation port** check box.

Data Types: `single` | `double` | `int8`

Port_3 — State output

scalar | vector | matrix

Block states, output as a scalar, vector, or matrix. By default, the block adds this port to the top of the block icon. Use the state 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.
- 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.

For more information, see “When to Use the State Port” on page 1-588.

Dependencies

To enable this port, select the **Show state port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Main

`Integrator method` — Accumulation method

`Integration: Forward Euler` (default) | `Integration: Backward Euler` | `Integration: Trapezoidal` | `Accumulation: Forward Euler` | `Accumulation: Backward Euler` | `Accumulation: Trapezoidal`

Specify the integration or accumulation method. See “Output Equations” on page 1-584 and “Integration and Accumulation Methods” on page 1-585 for more information.

Programmatic Use
Block Parameter: `IntegratorMethod`
Type: character vector
Values: `'Integration: Forward Euler'` | `'Integration: Backward Euler'` | `'Integration: Trapezoidal'` | `'Accumulation: Forward Euler'` | `'Accumulation: Backward Euler'` | `'Accumulation: Trapezoidal'`
Default: `'Integration: Forward Euler'`

`Gain value` — Value to multiply with integrator input

`1.0` (default) | scalar | vector

Specify a scalar, vector, or matrix by which to multiply the integrator input. Each element of the gain must be a positive real number.

- 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]`

Tip 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.

Programmatic Use
Block Parameter: `gainval`
Type: character vector
Values: scalar | vector
Default: `'1.0'`

`External reset` — Select when to reset states to initial conditions

`none` (default) | `rising` | `falling` | `either` | `level` | `sampled level`

Select the type of trigger event that resets the states to their initial conditions:

- `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 hold 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.

For more information, see “Reset the State” on page 1-589 and “Reset Trigger Types” on page 1-590.

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector
Values: `'none'` | `'rising'` | `'falling'` | `'either'` | `'level'` | `'sampled level'`
Default: `'none'`

`Initial condition source` — Select source of initial condition

`internal` (default) | `external`

Select source of initial condition:

- `internal` — Get the initial conditions of the states from the **Initial condition** block parameter.
- `external` — Get the initial conditions of the states from an external block, via the **IC** input port.

Dependencies

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter and enables the **IC** input port.

Programmatic Use
Block Parameter: `InitialConditionSource`
Type: character vector, string
Values: `'internal'` | `'external'`
Default: `'internal'`

`Initial condition` — Initial condition of states

`0` (default) | scalar | vector | matrix

Specify initial condition of the block states. The minimum and maximum values are bound by the **Output minimum** and **Output maximum** block parameters.

Tip Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

To enable this parameter, set the **Initial condition source** to `internal`.

Programmatic Use
Block Parameter: `InitialCondition`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'0'`

`Initial condition setting` — Select where to apply the initial condition

`Auto` (default) | `Output` | `Compatibility`

Select whether to apply the value of the **Initial condition** parameter to the block state or block output. The initial condition is also the reset value.

- `Auto` — Block chooses where to apply the **Initial condition** parameter.

  - If the block is in a non-triggered subsystem and **Integrator method** is set to an integration method, set initial conditions:

    ```
    x(0) = IC
    ```

    At reset:

    ```
    x(n) = IC
    ```
  - If the block is in a triggered or function-call subsystem and **Integrator method** is set to an integration method, set initial conditions as if `output` was selected.
- `Output` — Use this option when the block is in a triggered or a function-call subsystem and **Integrator method** is set to an integration method.

  Set 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 `Auto` 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 `Auto`.
- If **Underspecified initialization detection** is set to `Simplified`, the **Initial condition setting** parameter behaves as `Output`.

Note This parameter was named **Use initial condition as initial and reset value for** in Simulink before R2014a.

Programmatic Use
Block Parameter: `InitialConditionSetting`
Type: character vector
Value: `'Auto'` | `'Output'` | `'Compatibility'`
Default: `'Auto'`

`Sample time (-1 for inherited)` — Interval between samples

`-1` (default) | scalar | vector

Enter the discrete time interval between steps.

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 “Specify Sample Time” for more information.

Tips

- 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.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

`Limit output` — Limit block output values to specified range

`off` (default) | `on`

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

- Selecting this check box limits the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.
- Clearing this check box does not limit the block's output values.

Dependencies

Selecting this parameter enables the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use
Block Parameter: `LimitOutput`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Upper saturation limit` — Upper limit for the integral

`inf` (default) | scalar | vector | matrix

Specify the upper limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

Dependencies

To enable this parameter, select the **Limit output** check box.

Programmatic Use
Block Parameter: `UpperSaturationLimit`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'inf'`

`Lower saturation limit` — Lower limit for the integral

`-inf` (default) | scalar | vector | matrix

Specify the lower limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

Dependencies

To enable this parameter, select the **Limit output** check box.

Programmatic Use

Block Parameter: `LowerSaturationLimit`
Type: character vector , string
Values: scalar | vector | matrix
Default: `'-inf'`

`Show saturation port` — Enable saturation output port

`off` (default) | `on`

Select this check box to add a saturation output port to the block. When you clear this check box, the block does not have a saturation output port.

Dependencies

Selecting this parameter enables a saturation output port.

Programmatic Use

Block Parameter: `ShowSaturationPort`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Show state port` — Enable state output port

`off` (default) | `on`

Select this check box to add a state output port to the block. When you clear this check box, the block does not have a state output port.

Dependencies

Selecting this parameter enables a state output port.

Programmatic Use

Block Parameter: `ShowStatePort`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Ignore limit and reset when linearizing` — Treat block as not resettable

`off` (default) | `on`

Select this check box to have Simulink linearization commands 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.

Tip 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.

Programmatic Use
Block Parameter: `IgnoreLimit`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Signal Attributes

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Data type` — Output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- `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.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'`, `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type
`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Specify the rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Choose one of these rounding modes.

`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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

See Also

For more information, see “Rounding” (Fixed-Point Designer).

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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.

For more information, see "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder).

Dependencies

When you specify a value for **State name** and click **Apply**, you enable the **State name must resolve to Simulink signal object** parameter.

Programmatic Use
Parameter: `StateName`
Type: character vector
Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state names resolve to signal object

`Off` (default) | Boolean

Specify if requiring that state name resolve to Simulink signal objects or not. If selected, the software generates an error at run time if you specify a state name that does not match the name of a Simulink signal object.

Dependency

Enabled when you give the parameter **State name** a value and set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Programmatic Use

Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default)

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use

Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`
Default: `'Simulink.Signal'`

`Code generation storage class` — Storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Bitfield (Custom)` | `Volatile (Custom)` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation. If you do not need to interface to external code, select `Auto`.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use
Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto'` | `'Model default'` | `'ExportedGlobal''` | `'ImportedExtern'` | `'ImportedExternPointer'` | `'Custom'`
Default: `'Auto'`

`TypeQualifier` — Storage type qualifier

`''` (default) | `const` | `volatile` | ...

Specify a storage type qualifier such as `const` or `volatile`.

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.

Dependencies

To enable this parameter, set **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `Model default`. This parameter is hidden unless you previously set its value.

Programmatic Use
Block Parameter: `RTWStateStorageTypeQualifier`
Type: character vector
Values:`'' | 'const' | 'volatile' | ...`
Default: `''`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Depends on absolute time when used inside a triggered subsystem hierarchy.

HDL Code Generation

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

For more information about HDL code generation, see Discrete-Time Integrator.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Integrator

Topics

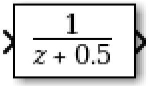
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder)

Introduced before R2006a

Discrete Transfer Fcn

Implement discrete transfer function

Library: Simulink / Discrete



Description

The Discrete Transfer Fcn block implements the *z*-transform transfer function:

$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^m + num_1 z^{m-1} + ... + num_m}{den_0 z^n + den_1 z^{n-1} + ... + den_n}$$

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. 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)
• Column vector (*K*-by-1) • Unoriented vector (*K*)	1	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)	• Scalar • Column vector (*M*-by-1)
• Row vector (1-by-*N*) • Matrix (*K*-by-*N*)	*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-*N*)	• Scalar • Matrix (*M*-by-*N*)

Sample-Based Processing

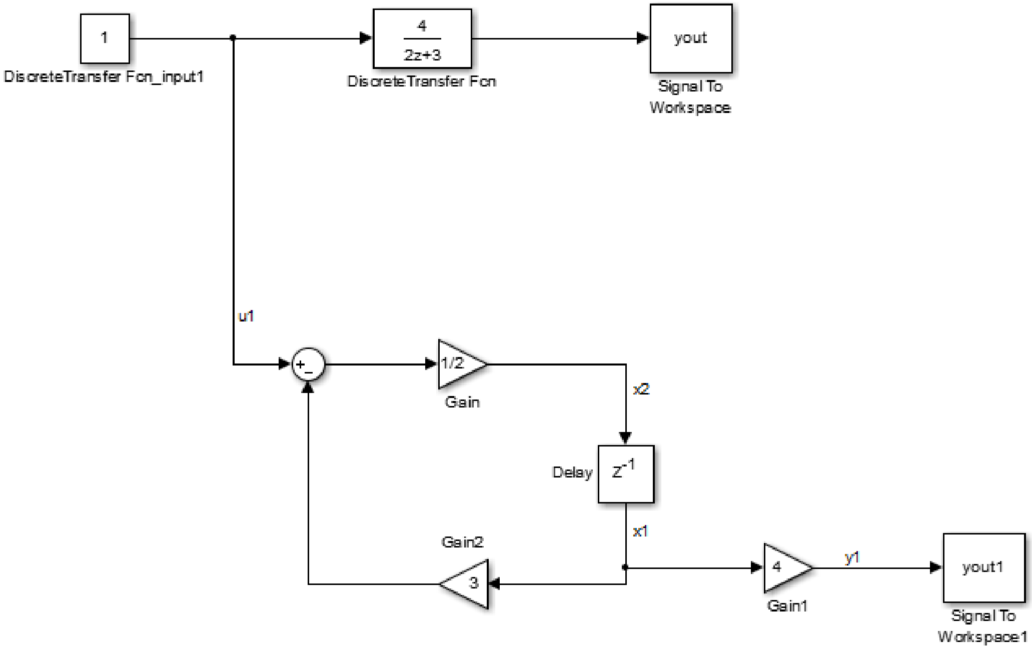
Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
• Scalar	1	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)
• Row vector (1-by-*N*) • Column vector (*N*-by-1) • Unoriented vector (*N*)	*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-*N*)	• Scalar
• Matrix (*K*-by-*N*)	*K* × *N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-(*K*×*N*))	• Scalar

When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. To initialize all states to zero, enter `0`. 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(number of zeros, number of poles)`.
- The matrix must have the same number of rows as the number of delay elements in the filter, `M = max(number of zeros, 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:

$$y_1 = 4x_1$$
$$x_2 = 1/2(u_1 - 3x_1)$$



Ports

Input

u — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

`Num` — Numerator coefficients

scalar | vector | matrix

Coefficients of the numerator polynomial specified as a vector or matrix in 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. The order of the denominator must be greater than or equal to the order of the numerator.

Dependencies

To enable this port, set **Numerator Source** to `Input port`.

Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

`Den` — Denominator coefficients

scalar | vector | matrix

Coefficients of the denominator polynomial specified as a vector in 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. The order of the denominator must be greater than or equal to the order of the numerator.

Dependencies

To enable this port, set **Denominator Source** to `Input port`.

Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

`External reset` — External reset signal
scalar

External reset signal, specified as a scalar. When the specified trigger event occurs, the block resets the states to their initial conditions.

Tip The icon for this port changes based on the value of the **External reset** parameter.

Dependencies

To enable this port, set **External reset** to `Rising`, `Falling`, `Either`, `Level`, or `Level hold`.

Limitations

The reset signal must be a scalar of type single, double, boolean, or integer. Fixed-point data types, except for `ufix1`, are not supported.

Data Types: `single` | `double` | `Boolean` | `int8` | `int16` | `int32` | `fixed point`

`x0` — Initial states
scalar | vector | matrix

Initial states, specified as a scalar, vector, or matrix. For more information about specifying states, see “Specifying Initial States” on page 1-614. States are complex when either the input or the coefficients are complex.

Dependencies

To enable this port, set **Initial states Source** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

Parameters

Main

`Numerator Source` — Source of numerator coefficients

`Dialog` (default) | `Input port`

Specify the source of the numerator coefficients as `Dialog` or `Input port`.

Programmatic Use
Block Parameter: `NumeratorSource`
Type: character vector
Values: `'Dialog' | 'Input port'`
Default: `'Dialog'`

`Numerator Value` — Numerator coefficients

`[1]` (default) | scalar | vector | matrix

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.

Dependencies

To enable this parameter, set the **Numerator Source** to `Dialog`.

Programmatic Use
Block Parameter: `Numerator`
Type: character vector
Values: scalar | vector | matrix
Default: `'[1]'`

`Denominator Source` — Source of denominator coefficients

`Dialog` (default) | `Input port`

Specify the source of the denominator coefficients as `Dialog` or `Input port`.

Programmatic Use

Block Parameter: `DenominatorSource`
Type: character vector
Values: `'Dialog' | 'Input port'`
Default: `'Dialog'`

`Denominator Value` — Denominator coefficients

`[1 0.5]` (default) | scalar | vector | matrix

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.

Dependencies

To enable this parameter, set the **Denominator Source** to `Dialog`.

Programmatic Use

Block Parameter: `Denominator`
Type: character vector
Values: scalar | vector | matrix
Default: `'[1 0.5]'`

`Initial states Source` — Source of initial states

`Dialog` (default) | `Input port`

Specify the source of the initial states as `Dialog` or `Input port`.

Programmatic Use

Block Parameter: `InitialStatesSource`
Type: character vector
Values: `'Dialog' | 'Input port'`
Default: `'Dialog'`

`Initial states Value` — Initial filter states

`0` (default) | scalar | vector | matrix

Specify the initial filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-614.

Dependencies

To enable this parameter, set **Initial states Source** to `Dialog`.

Programmatic Use
Block Parameter: `InitialStates`
Type: character vector
Values: scalar | vector | matrix
Default: `'0'`

`External reset` — External state reset

`None` (default) | `Rising` | `Falling` | `Either` | `Level` | `Level hold`

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: • 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

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector
Values: `'None'` | `'Rising'` | `'Falling'` | `'Either'` | `'Level'` | `'Level hold'`
Default: `'None'`

`Input processing` — Sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Elements as channels (sample based)' | 'Columns as channels (frame based)'`
Default: `'Elements as channels (sample based)'`

Optimize by skipping divide by leading denominator coefficient (a0) — Skip divide by a0

`off` (default) | `on`

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.

Programmatic Use
Block Parameter: `a0EqualsOne`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Sample time (-1 for inherited) — Interval between samples

`-1` (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Data Types

`State` — State data type

`Inherit: Same as input` (default) | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

`Numerator coefficients` — Numerator coefficient data type

`Inherit: Inherit via internal rule` (default) | `int8` | `int16` | `int32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

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 data type attributes. For more information, see "Specify Data Types Using Data Type Assistant".

Programmatic Use
Block Parameter: `NumCoeffDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Numerator coefficient minimum` — Minimum value of numerator coefficients

[ ] (default) | scalar

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

Programmatic Use
Block Parameter: `NumCoeffMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Numerator coefficient maximum` — Maximum value of numerator coefficients

[ ] (default) | scalar

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

Programmatic Use
Block Parameter: `NumCoeffMax`
Type: character vector

Values: scalar
Default: `'[]'`

`Numerator product output` — Numerator product output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use

Block Parameter: `NumProductDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via interal rule'`

`Numerator accumulator` — Numerator accumulator data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `NumAccumDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Same as input'` | `'Inherit: Same as product output'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via interal rule'`

`Denominator coefficients` — Denominator coefficient data type

`Inherit: Inherit via internal rule` (default) | `int8` | `int16` | `int32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `DenCoeffDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Same wordlength as input'`

`Denominator coefficient minimum` — Minimum value of denominator coefficients

`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `DenCoeffMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Denominator coefficient maximum` — Maximum value of denominator coefficients

`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `DenCoeffMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Denominator product output` — Denominator product output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `DenProductDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Same as input'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via interal rule'`

`Denominator accumulator` — Denominator accumulator data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `int8` | `int16` | `int32` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `DenAccumDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Same as input'` | `'Inherit: Same as product output'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via interal rule'`

`Output` — Output data type
`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `int8` | `int16` | `int32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'`
Default: `'Inherit: Inherit via interal rule'`

`Output minimum` — Minimum value of output
`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Output maximum` — Maximum value of output
`[]` (default) | scalar

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

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

Saturate on integer overflow — Method of overflow action

off (default) | on

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (on).	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 int8 (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 (off).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the int8 (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 int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: `StateName`
Type: character vector
Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, specify a value for **State name**. 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**.

Programmatic Use

Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default) | `<StorageClass.PackageName>`

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use

Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`
Default: `'Simulink.Signal'`

`Code generation storage class` — State storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `BitField (Custom)` | `Model default` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `AutoScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation.

- `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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

To enable this parameter, specify a value for **State name**.

Programmatic Use

Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto' | 'SimulinkGlobal' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

`TypeQualifier` — Storage type qualifier

`''` (default) | `const` | `volatile` | ...

Specify a storage type qualifier such as `const` or `volatile`.

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.

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 “Data Objects”.

Dependencies

To enable this parameter, set **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `Model default`. This parameter is hidden unless you previously set its value.

Programmatic Use
Block Parameter: `RTWStateStorageTypeQualifier`
Type: character vector
Values: `'' | 'const' | 'volatile' | ...`
Default: `''`

Block Characteristics

Data Types	`double` | `single` | `base integer`[a] | `fixed point`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[a] This block only supports signed fixed-point data types.

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™.

For more information about HDL code generation, see Discrete Transfer Fcn.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

This block only supports signed fixed-point data types.

See Also

Discrete Filter | Transfer Fcn

Topics

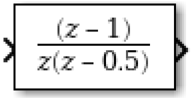
“Working with States”
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)

Introduced before R2006a

Discrete Zero-Pole

Model system defined by zeros and poles of discrete transfer function

Library: Simulink / 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:

$$H(z) = K\frac{Z(z)}{P(z)} = K\frac{(z-Z_1)(z-Z_2)\ldots(z-Z_m)}{(z-P_1)(z-P_2)\ldots(z-P_n)},$$

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.

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**.

Ports

Input

`Port_1` — Input signal

scalar

Input signal specified as a real-valued scalar.

Data Types: `single` | `double`

Output

`Port_1` — Model of discrete system

scalar | vector

Model of system as defined by zeros, poles, and gain of discrete transfer function. The width of the output is equal to the number of columns in the **Zeros** matrix, or one if **Zeros** is a vector.

Data Types: `single` | `double`

Parameters

Main

`Zeros` — Matrix of zeros

`[1]` (default) | vector | matrix

Specify the vector or matrix of zeros. The number of zeros must be less than or equal to the number of poles. If the poles and zeros are complex, they must be complex conjugate pairs.

- 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 the matrix contains the zeros of a transfer function that relates the system input to one of the outputs.

Programmatic Use
Block Parameter: `Zeros`
Type: character vector
Values: vector
Default: `'[1]'`

`Poles` — Vector of poles

`[0 0.5]` (default) | vector

Specify the vector of poles. 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 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.

Programmatic Use
Block Parameter: `Poles`
Type: character vector
Values: vector
Default: `'[0 0.5]'`

`Gain` — Gain value

`1` (default) | scalar | vector

Specify vector of gain values.

- For a single-output system, enter a scalar or 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**.

Programmatic Use
Block Parameter: `Gain`
Type: character vector
Values: scalar | vector
Default: `'1'`

`Sample time (-1 for inherited)` — Interval between samples

`-1` | scalar | vector

Specify the time interval between samples. For more information, see Specifying Sample Time.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: `StateName`
Type: character vector
Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, specify a value for **State name**. 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**.

Programmatic Use
Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default) | `<StorageClass.PackageName>`

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use
Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`
Default: `'Simulink.Signal'`

`Code generation storage class` — State storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `BitField (Custom)` | `Model default` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `AutoScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation.

- `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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

To enable this parameter, specify a value for **State name**.

Programmatic Use
Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto' | 'SimulinkGlobal' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

`TypeQualifier` — Storage type qualifier

`''` (default) | `const` | `volatile` | ...

Specify a storage type qualifier such as `const` or `volatile`.

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.

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 “Data Objects”.

Dependencies

To enable this parameter, set **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `Model default`. This parameter is hidden unless you previously set its value.

Programmatic Use

Block Parameter: `RTWStateStorageTypeQualifier`
Type: character vector
Values: `'' | 'const' | 'volatile' | ...`
Default: `''`

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

See Also

Discrete Transfer Fcn | Zero-Pole

Topics

“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder)
“Data Objects”

Introduced before R2006a

Display

Display signal value during simulation

Library: Simulink / Dashboard

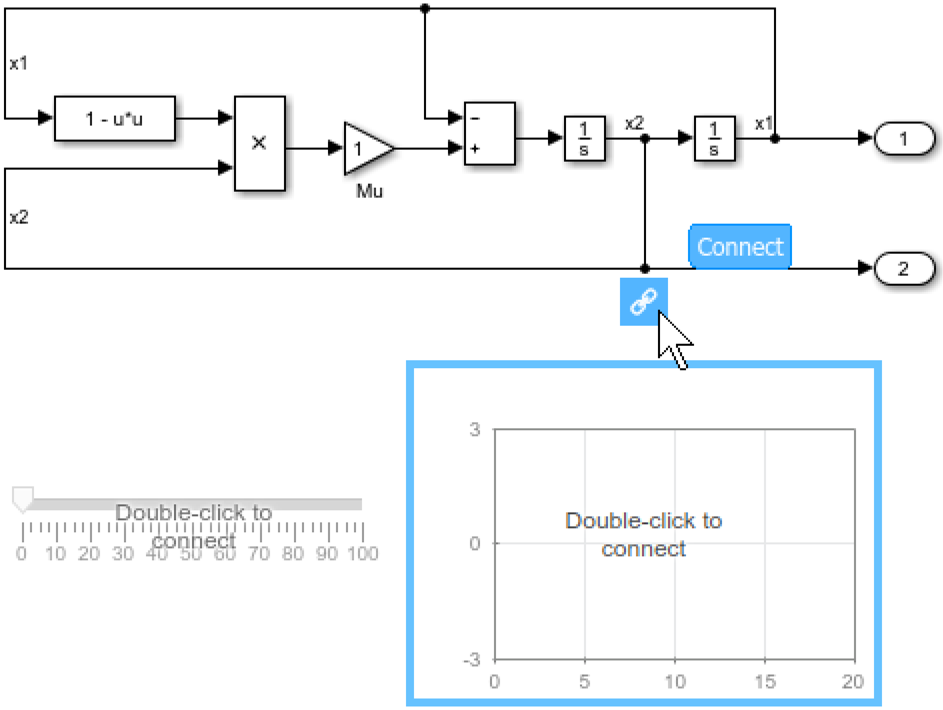
Description

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.

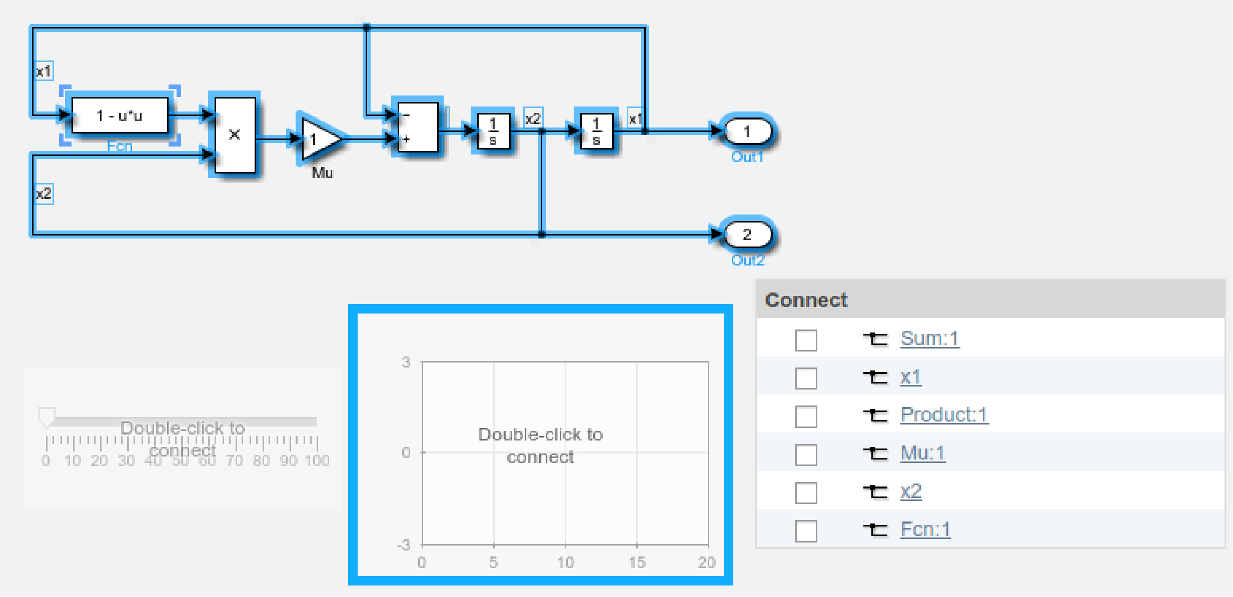
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

□ Sum:1
□ x1
□ Product:1
□ Mu:1
□ x2
□ Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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.

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: Simulink / Sinks

Description

The Display block shows the value of the input data. You can specify the format and the frequency of display.

If the block input is an array, you can resize the block vertically or horizontally to show more than just the first element. 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.

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

When You See...	The Value That Appears Is...
`oct`	In octal format

Displaying Strings

When working with strings, the Display block displays:

- Strings with double quotes.
- Special characters such as newline are shown as escaped sequences, for example `'\n'`.
- Non-displayable characters as escaped octal number, for example `'\201'`.

If the incoming signal is of type string, the **Format** parameter selection does not affect the display of the string.

Ports

Input

`Port_1` — Input data
scalar | vector

Input data to display.

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

Parameters

`Format` — Format to display input data
`short` (default) | `long` | `short_e` | `long_e` | `bank` | `hex (Stored Integer)` | `binary (Stored Integer)` | `decimal (Stored Integer)` | `octal (Stored Integer)`

Specify the format of the data that appears.

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
`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.

If the incoming signal is of type string, the selection of the **Format** parameter does not affect the display of the string.

Programmatic Use
Block Parameter: `Format`
Type: character vector
Values: `'short'` | `'long'` | `'short_e'` | `'long_e'` | `'bank'` | `'hex (Stored Integer)'` | `'binary (Stored Integer)'` | `'decimal (Stored Integer)'` | `'octal (Stored Integer)'`
Default: `'short'`

Decimation — Display rate

`1` (default) | `integer`

Specify how often to display data.

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.

Programmatic Use
Block Parameter: `Decimation`
Type: character vector
Values: `'1'` | integer
Default: `'1'`

`Floating display` — Floating display

`off` (default) | `on`

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.

Programmatic Use
Block Parameter: `Floating`

Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

HDL Code Generation

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

For more information about HDL code generation, see Display.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

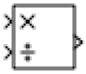
Scope | To File | To Workspace

Introduced before R2006a

Divide

Divide one input by another

Library: Simulink / Math Operations



Description

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 nondefault 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.

Ports

Input

X — Input signal to multiply

scalar | vector | matrix | N-D array

Input signal to be multiplied with other inputs.

Dependencies

To enable one or more **X** ports, specify one or more * characters for the **Number of inputs** parameter.

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

÷ — Input signal to divide or invert

scalar | vector | matrix | N-D array

Input signal for division or inversion operations.

Dependencies

To enable one or more ÷ ports, specify one or more `/` characters for the **Number of inputs** parameter.

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

`Port_1` — First input to multiply or divide

scalar | vector | matrix | N-D array

First input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

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

`Port_N` — Nth input to multiply or divide

scalar | vector | matrix | N-D array

Nth input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

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

Output

`Port_1` — Output computed by multiplying, dividing, or inverting inputs

scalar | vector | matrix | N-D array

Output computed by multiplying, dividing, or inverting inputs.

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

Parameters

Main

`Number of inputs` — Control number of inputs and type of operation

`*/` (default) | positive integer scalar | `*` or `/` for each input port

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

When you specify:

- **1 or * or /**

  The block has one input port. In element-wise mode, the block 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 1-1448 and "Matrix Mode" on page 1-1449 for more information.
- **Integer value > 1**

  The block 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 1-1448 and "Matrix Mode" on page 1-1449 for more information.
- **Unquoted string of two or more * and / characters**

  The block 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 1-1448 and "Matrix Mode" on page 1-1449 for more information.

Programmatic Use
Block Parameter: `Inputs`

Type: character vector
Values: `'2' | '*' | '**' | '*/' | '*/*' | ...`
Default: `'*/'`

`Multiplication` — Element-wise (.*) or Matrix (*) multiplication

`Element-wise(.*)` (default) | `Matrix(*)`

Specify whether the block performs `Element-wise(.*)` or `Matrix(*)` multiplication.

Programmatic Use

Block Parameter: `Multiplication`
Type: character vector
Values: `'Element-wise(.*)' | 'Matrix(*)'`
Default: `'Element-wise(.*)'`

`Multiply over` — All dimensions or specified dimension

`All dimensions` (default) | `Specified dimension`

Specify the dimension to multiply over as `All dimensions`, or `Specified dimension`. When you select `Specified dimension`, you can specify the **Dimension** as `1` or `2`.

Dependencies

To enable this parameter, set **Number of inputs** to `*` and **Multiplication** to `Element-wise (.*)`.

Programmatic Use

Block Parameter: `CollapseMode`
Type: character vector
Values: `'All dimensions' | 'Specified dimension'`
Default: `'All dimensions'`

`Dimension` — Dimension to multiply over

`1` (default) | `2` | `...` | `N`

Specify the dimension to multiply over as an integer less than or equal to the number of dimensions of the input signal.

Dependencies

To enable this parameter, set:

- **Number of inputs** to `*`

- **Multiplication** to `Element-wise (.*)`
- **Multiply over** to `Specified dimension`

Programmatic Use
Block Parameter: `CollapseDim`
Type: character vector
Values: `'1' | '2' | ...`
Default: `'1'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

`Require all inputs to have the same data type` — Require that all inputs have the same data type

`off` (default) | `on`

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `Inherit: Same as first input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- `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)`, in other words, 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 first input` — Use data type of first input signal.

Programmatic Use

Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule`|`'Inherit: Same as first input'`|`'Inherit: Inherit via back propagation'`|`'double'`|`'single'`|`'int8'`|`'uint8'`|`'int16'`|`'uint16'`|`'int32'`|`'uint32'`|`'fixdt(1,16)'`|`'fixdt(1,16,0)'`|`'fixdt(1,16,2^0,0)'`|`'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations. You can select:

`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.

For more information, see "Rounding" (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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 (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Check for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

These conditions may yield different results between simulation and the generated code:

- The Divide block inputs contain a `NaN` or `inf` value
- The Divide block generates `NaN` or `inf` during execution

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`.

The Simulink Coder build process provides efficient code for matrix inverse and division operations. This table 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.

HDL Code Generation

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

For more information about HDL code generation, see Divide.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Dot Product | Product | Product of Elements

Introduced before R2006a

DocBlock

Create text that documents model and save text with model
Library: Simulink / 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.

Tip To edit the block parameters of the DocBlock, right-click the block icon and select **Mask > Mask Parameters...**.

Parameters

`Code generation template symbol` — Template symbol for generated code

`' '` | `Abstract` | `Description` | `History` | `Modified History` | `Notes`

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).

Dependencies

For comments to appear in the generated code, you must also set the **Document type** to `Text`.

Programmatic Use
Block Parameter: `ECoderFlag`
Type: character vector
Values: `Abstract | Description | History | Modified History | Notes`
Default: `'0'`

`Document type` — Type of document

`Text` (default) | `RTF` | `HTML`

Select the type of document associated with the DocBlock. The options are:

- `Text`
- `RTF`
- `HTML`

Dependencies

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.

Programmatic Use
Block Parameter: `DocumentType`
Type: character vector
Values: `Text | RTF | HTML`
Default: `'Text'`

Block Characteristics

Data Types	
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Uses the template symbol you specify for the Embedded Coder Flag block parameter to add comments to generated code. Requires an Embedded Coder license. For more information, see “Use a Simulink DocBlock to Add a Comment” (Embedded Coder).

HDL Code Generation

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

For more information about HDL code generation, see DocBlock.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Model Info | `docblock`

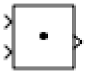
Topics

“Add Global Comments” (Embedded Coder)

Introduced before R2006a

Dot Product

Generate dot product of two vectors
Library: Simulink / Math Operations



Description

The Dot Product block generates the dot product of the input vectors. The scalar output, `y`, is equal to the MATLAB operation

```
y = sum(conj(u1) .* u2 )
```

where `u1` and `u2` represent the input vectors. 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
complex	complex	complex

Ports

Input

`Port_1` — First operand input signal

scalar | vector

Signal representing the first operand to the dot product calculation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_2` — Second operand input signal

scalar | vector

Signal representing the second operand to the dot product calculation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Dot product output signal

scalar | vector

Output signal resulting from the dot product calculation of the two input signals.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Require all inputs to have the same data type` — Require all inputs to have the same data type

`on` (default) | `off`

Clear this check box for all the inputs to have different data types.

Programmatic Use

Block Parameter: `InputSameDT`

Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Output minimum` — Minimum output value for range checking

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Specify the upper value of the output range that Simulink checks 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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `Inherit: Same as first input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- `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)`, in other words, 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 first input` — Use data type of first input signal.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Same as first input'` | `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types
`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations. For more information, see "Rounding" (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
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™.

For information about HDL Code Generation, see Dot Product.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

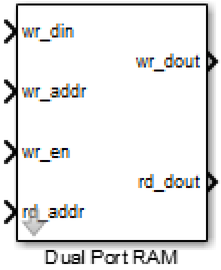
See Also

Product | Product of Elements

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`.

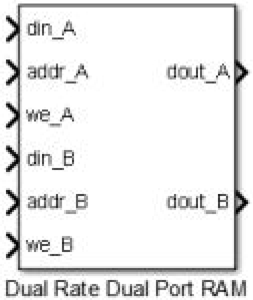
See Also

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

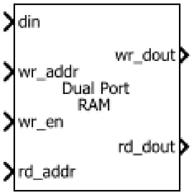
Dual Port RAM | HDL FIFO | Simple Dual Port RAM | Single Port RAM

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

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.

Limitations

- The block does not support `boolean` inputs. Cast any `boolean` types to `ufix1` for input to the block.

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.

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

Convert floating-point algorithms to fixed point 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.

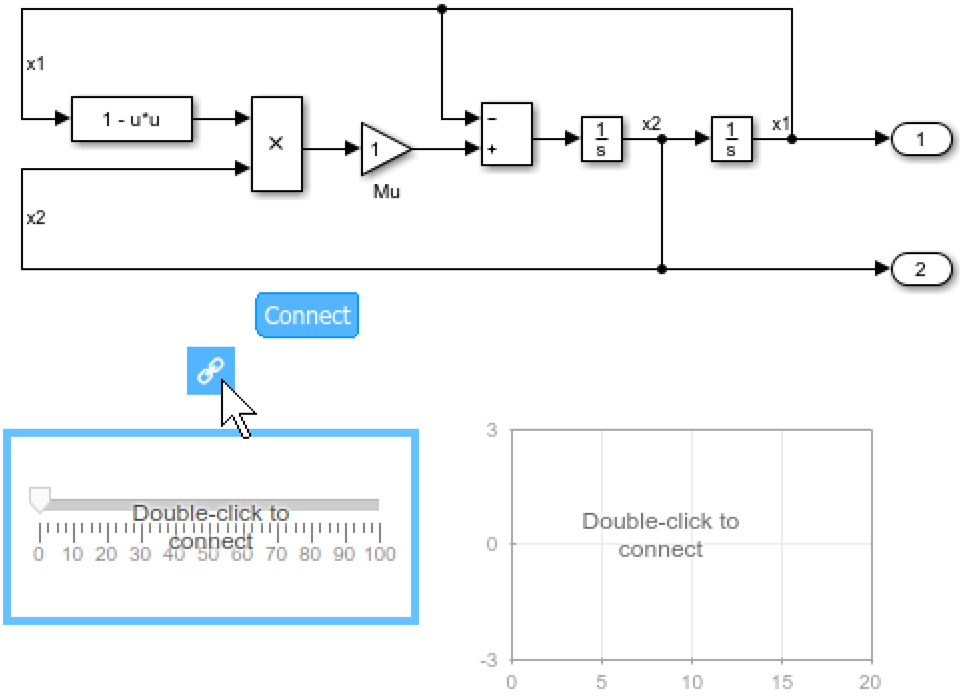
Double-clicking the Edit block does not open its dialog box during simulation and when the block is selected. To edit the block parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

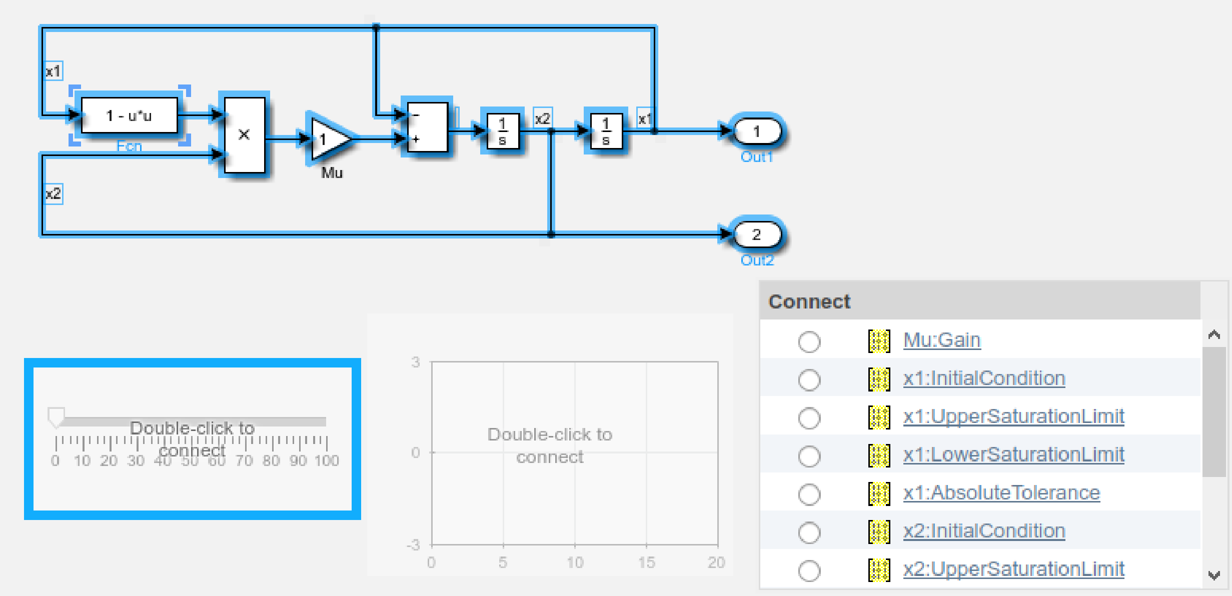
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○ Mu:Gain

○ x1:InitialCondition

○ x1:UpperSaturationLimit

○ x1:LowerSaturationLimit

○ x1:AbsoluteTolerance

○ x2:InitialCondition

○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

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: Simulink / 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
Values: `'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
Values: `'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
Values: `'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
Values: `'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
Values: `'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
Values: `'-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” (Simulink Coder).

`[]`

Unspecified minimum value.

real scalar

Real double scalar value.

Programmatic Use
Block parameter: `OutMin`
Type: character vector
Values: `'[]'|'<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” (Simulink Coder).

`[]`

Unspecified maximum value.

real scalar

Real double scalar value.

Programmatic Use
Block parameter: `OutMax`

Type: character vector
Values: `'[]'` | `'<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
Values: `'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
Values: `'on'` | `'off'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

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™.

For more information about HDL code generation, see Enable.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Enabled Subsystem | Enabled and Triggered Subsystem | Subsystem

Topics

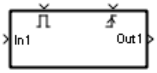
"Conditionally Executed Subsystems Overview"
"Using Enabled Subsystems"
"Using Enabled and Triggered Subsystems"

Introduced before R2006a

Enabled and Triggered Subsystem

Subsystem whose execution is enabled and triggered by external input

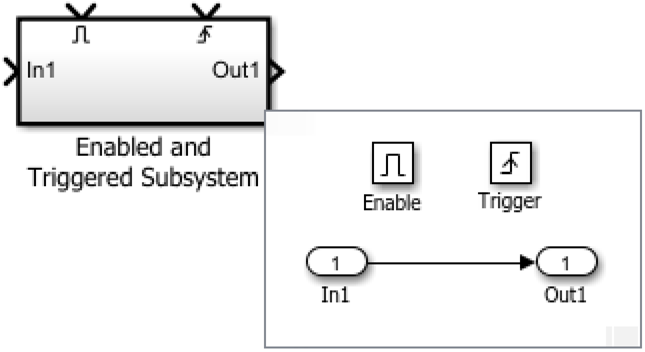
Library: Simulink / Ports & Subsystems



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`

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[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 code generation support depends on block implementation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks

Enable | Enabled Subsystem | Function-Call Subsystem | Subsystem | Trigger | Triggered Subsystem

Topics

"Conditionally Executed Subsystems Overview"
"Using Enabled Subsystems"
"Using Triggered Subsystems"
"Using Enabled and Triggered Subsystems"
"Using Function-Call Subsystems"

Introduced before R2006a

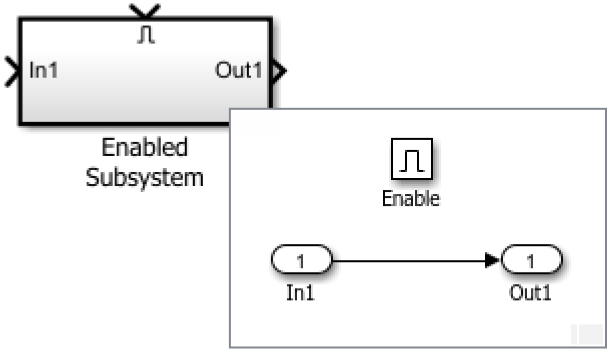
Enabled Subsystem

Subsystem whose execution is enabled by external input

Library: Simulink / 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`

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[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 code generation support depends on block implementation.

HDL Code Generation

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

For more information about HDL code generation, see Enabled Subsystem.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks
Enable | Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Triggered Subsystem

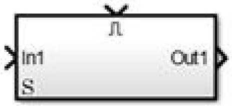
Topics

"Conditionally Executed Subsystems Overview"
"Using Enabled Subsystems"
"Using Triggered Subsystems"
"Using 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 “Using 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: Simulink / 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, use Enumerated Constant rather than Constant. For more information, see “Simulink Enumerations”.

Ports

Output

`Port_1` — Enumerated constant
scalar | vector | matrix

Enumerated constant value, specified as a scalar, vector, or matrix.

Data Types: `enumerated`

Parameters

`Output data type` — Output data type
`Enum: SlDemoSign` (default) | `Enum:<ClassName>`

Specify 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 drop-down list or enter `Enum:`*`ClassName`* in the **Output data type** field, where *`ClassName`* is the name of the MATLAB class that defines the type.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Enum:<ClassName>'`
Default: `'Enum: SlDemoSign'`

`Mode` — Category of data to specify

`Enumerated` (default)

Select the category of data to specify.

`Enumerated`

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

`Value` — Enumerated value

`SlDemoSign.Positive` (default) | `Enum:<ClassName.Value>`

Specify the value or values 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 values, select from the drop-down list or enter any MATLAB expression that evaluates to the desired result, including an expression that uses tunable parameters. All specified values 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.

Programmatic Use
Block Parameter: `Value`
Type: character vector
Values: `'Enum:<ClassName.Value>'`
Default: `'SlDemoSign.Positive'`

`Sample time` — Sample time

`inf` (default) | scalar | vector

Specify the interval between times that the block output can change during simulation (for example, due to tuning the **Value** parameter). The default value of `inf` indicates that the block output can never change. A sample time of `inf` speeds the simulation and generated code by avoiding the need to recompute the block output. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'inf'`

Block Characteristics

Data Types	`enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
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™.

For more information about HDL code generation, see Enumerated Constant.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Constant | `enumeration`

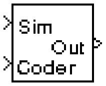
Topics

"Use Enumerated Data in Simulink Models"
"Simulink Enumerations"
"Code Generation for Enumerations"
"Specify Sample Time"

Introduced in R2009b

Environment Controller

Create branches of block diagram that apply only to simulation or only to code generation
Library: Simulink / Signal Routing



Description

The Environment Controller 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. This table 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 in normal or accelerator modes	**Sim**
Simulation of a referenced model in 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 **Code Generation > Optimization** 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 more information, see “Block reduction”.

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.

Ports

Input

`Sim` — Simulation input

scalar | vector | matrix

Simulation input values, specified as a scalar, vector, or matrix. Input signal must have the same width as the input to the **Coder** port.

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

`Coder` — Code generation input

scalar | vector | matrix

Code generation input values, specified as a scalar, vector, or matrix. Input signal must have the same width as the input to the **Sim** port.

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

Output

`Out` — Values from Sim or Coder input port
scalar | vector | matrix

Values from the **Sim** or **Coder** input port, depending on the current environment. For more information on what the block outputs in various simulation and code generation modes, see “Description” on page 1-724.

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

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

“Default parameter behavior” (Simulink Coder)

Topics

“Spawn and Synchronize Execution of RTOS Task” (Simulink Coder)
“Dual-Model Approach: Code Generation” (Simulink Coder)
“Block reduction”

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
Block 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 tying 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

Block 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

Block Parameter: `Variant`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Note

- The Reset event port of a Model block can be conditional and propagates the net Variant conditions defined on the corresponding Reset Function block(s) in the referenced model.
- Initialize and Terminate event ports are always unconditional because they control both the model default and block-specific initialize and terminate events of the referenced model. If you define an Initialize function block in the referenced model, it corresponds to an explicit initialize event.

`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

Block 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

Block Parameter: `GeneratePreprocessorConditionals`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

See Also

Initialize Function | Reset Function | State Reader | State Writer | Terminate Function

Topics

“Customize Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”

Extract Bits

Output selection of contiguous bits from input signal
Library: Simulink / 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

  number of output bits = ceil(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

  number of output bits = ceil(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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. 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.

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

Output

`Port_1` — Extracted bits

scalar | vector | matrix | N-D array

Contiguous selection of extracted bits, specified as a scalar, vector, matrix, or N-D array. Floating-point inputs are passed through the block unchanged.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Bits to extract` — Method for extracting bits

`Upper half` (default) | `Lower half` | `Range starting with most significant bit` | `Range ending with least significant big` | `Range of bits`

Select the method for extracting bits from the input signal.

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.

Programmatic Use

Block Parameter: `bitsToExtract`
Type: character vector
Values: `'Upper half' | 'Lower half' | 'Range starting with most significant bit' | 'Range ending with least significant bit' | 'Range of bits'`
Default: `'Upper half'`

`Number of bits` — Number of bits to output

8 (default) | positive integer

Select the number of bits to output from the input signal. Signed integer data types must have at least two bits. Unsigned data integer types can be as short as a single bit.

Dependencies

To enable this parameter, set **Bits to extract** to `Range starting with most significant bit` or `Range ending with least significant bit`.

Programmatic Use

Block Parameter: `numBits`
Type: character vector
Values: positive integer
Default: `'8'`

`Bit indices` — Contiguous range of bits to output

[0 7] (default) | contiguous range

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.

Dependencies

To enable this parameter, set **Bits to extract** to `Range of bits`.

Programmatic Use
Block Parameter: `bitIdxRange`
Type: character vector
Values: contiguous range
Default: `'[0 7]'`

`Output scaling mode` — Output scaling mode

`Preserve fixed-point scaling` (default) | `Treat bit field as an integer`

Select the scaling mode to use on the output bit 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.

Programmatic Use
Block Parameter: `outScalingMode`
Type: character vector
Values: `'Preserve fixed-point scaling' | 'Treat bit field as an integer'`
Default: `'Preserve fixed-point scaling'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`

Multidimensional Signals	No
Variable-Size Signals	No
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™.

For more information about HDL code generation, see Extract Bits.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Bit Clear | Bit Set | Bitwise Operator

Introduced before R2006a

Fcn

Apply specified expression to input

Library: Simulink / 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

- 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 “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Ports

Input

In — Input to a Fcn block
scalar | matrix | vector

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.

Data Types: `single` | `double`

Output

`Out` — Output from a Fcn block

scalar | matrix | vector

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.

Data Types: `single` | `double`

Parameters

`Expression` — Specify the mathematical expression

mathematical 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 1-738).

Programmatic Use
Block Parameter: `Expr`
Type: character vector
Value: mathematical expression
Default: `'sin(u(1)*exp(2.3*(-u(2))))'`

`Sample time` — Specify sample time in the block

scalar

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”.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Interpreted MATLAB Function | MATLAB Function | MATLAB System

Introduced before R2006a

Find Nonzero Elements

Find nonzero elements in array
Library: Simulink / Math Operations

Description

The Find Nonzero Elements 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 Nonzero Elements 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.

The Find Nonzero Elements block outputs a variable-size signal. The sample time for any variable-size signal must be discrete. If your model does not already use a fixed-step solver, you may need to select a fixed-step solver in the Configuration Parameters dialog. For more information, see “Solvers” and “Choose a Solver”.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix | N-D array

Input signal from which the block finds all nonzero elements.

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

Output

`Port_1` — Indices of nonzero elements
variable-size signal

The Find Nonzero Elements block outputs the indices of nonzero elements as a variable-size signal. You control the data type of the output using the **Output data type** block parameter.

Dependencies

By default, the block outputs linear indices from the first output port. When you change the **Index output format** to `Subscripts`, the block instead provides the element indices of a two-dimension or larger signal in a subscript form. In this mode, you must specify the **Number of input dimensions**, and the block creates a separate output port for each dimension.

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

`Port_2` — Values of nonzero elements

variable-size signal

The Find block can optionally output the values of all nonzero elements as a variable-size signal.

Dependencies

To enable this port, select **Show output port for nonzero input values**.

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

Parameters

Main

`Index output format` — Format for indices of nonzero elements

`Linear indices` (default) | `Subscripts`

Select the output format for the indices of the nonzero input values.

- Selecting `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.

- Selecting `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 specify the **Number of input dimensions**.

Programmatic Use
Block Parameter: `IndexOutputFormat`
Type: character vector
Values: `'Linear indices'` | `'Subscripts'`
Default: `'Linear indices'`

`Number of input dimensions` — Number of dimensions for the input signal

`1` (default) | scalar

Specify the number of dimensions for the input signal as a positive integer value from 1 to 32.

Dependencies

To enable this parameter, set **Index output format** to `Subscripts`.

Programmatic Use
Block Parameter: `NumberOfInputDimensions`
Type: character vector
Values: scalar
Default: `'1'`

`Index mode` — Specify zero- or one-based indexing

`Zero-based` (default) | `One-based`

Specify the indexing mode as `Zero-based` or `One-based`.

- For `Zero-based` indexing, an index of `0` specifies the first element of the input vector. An index of `1` specifies the second element, and so on.
- For `One-based` indexing, an index of `1` specifies the first element of the input vector. An index of `2`, specifies the second element, and so on.

Programmatic Use
Block Parameter: `IndexMode`
Type: character vector
Values: `'Zero-based'` | `'One-based'`
Default: `'Zero-based'`

`Show output port for nonzero input values` — Enable output port for nonzero values

`off` (default) | `on`

Show or hide the output port for nonzero input values.

- When you clear this check box (`off`), the block hides the output port for nonzero input values.
- When you select this check box (`on`), the block displays the output port for nonzero input values. The additional output port provides values of the nonzero input elements.

Programmatic Use
Block Parameter: `ShowOutputPortForNonzeroInputValues`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Data Types

`Output data type` — Output data type

`Inherit: Inherit via internal rule` (default) | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `<data type expression>`

Specify the output data type.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

`find`

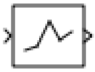
Topics

“Array Indexing” (MATLAB)
“Variable-Size Signal Basics”
“Simulink Models Using Variable-Size Signals”
“Control Signal Data Types”

Introduced in R2010a

First-Order Hold

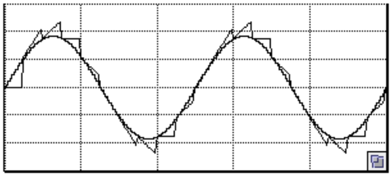
Implement first-order sample-and-hold
Library: Simulink / Discrete



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.



Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `double`

Output

`Port_1` — First-order sample-and-hold
scalar | vector | matrix

First-order sample and hold applied to the input signal.

Data Types: `double`

Parameters

`Sample time` — Time interval between samples
1 (default) | scalar | vector

The time interval between samples. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: `Ts`
Type: character vector
Values: scalar | vector
Default: `'1'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

See Also

Memory | Zero-Order Hold

Topics

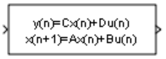
“What Is Sample Time?”

Introduced before R2006a

Fixed-Point State-Space

Implement discrete-time state space

Library: Simulink / Additional Math & Discrete / Additional Discrete



Description

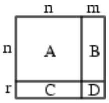
The Fixed-Point State-Space block implements the system described by

$$y(n) = \mathrm{C}x(n) + \mathrm{D}u(n)$$

$$x(n+1) = \mathrm{A}x(n) + \mathrm{B}u(n)$$

where *u* is the input, *x* is the state, and *y* is the output. Both equations have the same data type.

- **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 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input vector, where the width equals the number of columns in the **B** and **D** matrices. For more information, see “Description” on page 1-750.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal, with width equal to the number of rows in the **C** and **D** matrices. For more information, see “Description” on page 1-750.

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

Parameters

Main

`State Matrix A` — Matrix of states

`[2.6020 -2.2793 0.6708; 1 0 0; 0 1 0]` (default) | scalar | vector | matrix

Specify the matrix of states as a real-valued *n*-by-*n* matrix, where *n* is the number of states. For more information on the matrix coefficients, see “Description” on page 1-750.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: scalar | vector | matrix
Default: `'[2.6020 -2.2793 0.6708; 1 0 0; 0 1 0]'`

`Input Matrix B` — Column vector of inputs

`[ 1; 0; 0]` (default) | scalar | vector | matrix

Specify the column vector of inputs as a real-valued *n*-by-*m* matrix, where *n* is the number of states, and *m* is the number of inputs. For more information on the matrix coefficients, see "Description" on page 1-750.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: scalar | vector | matrix
Default: `'[ 1; 0; 0]'`

`Output Matrix C` — Column vector of outputs

`[0.0184 0.0024 0.0055]` (default) | scalar | vector | matrix

Specify the column vector of outputs as a real-valued *r*-by-*n* matrix, where *r* is the number of outputs, and *n* is the number of states. For more information on the matrix coefficients, see "Description" on page 1-750.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: scalar | vector | matrix
Default: `'[0.0184 0.0024 0.0055]'`

`Direct Feedthrough Matrix D` — Matrix for direct feedthrough

`[0.0033]` (default) | scalar | vector | matrix

Specify the matrix for direct feedthrough as a real-valued *r*-by-*m* matrix, where *r* is the number of outputs, and *m* is the number of inputs. For more information on the matrix coefficients, see "Description" on page 1-750.

Programmatic Use
Block Parameter: `D`
Type: character vector

Values: scalar | vector | matrix
Default: `'[0.0033]'`

`Initial condition for state` — Initial state vector

`0.0` (default) | scalar | vector | matrix

Specify the initial condition for the state.

Limitations

The initial state cannot be `inf` or `NaN`.

Programmatic Use
Block Parameter: `X0`
Type: character vector
Values: scalar | vector | matrix
Default: `'0.0'`

Signal Attributes

`Data type for internal calculations` — Data type for internal calculations

`fixdt('double')` (default) | data type string

Specify the data type the block uses for internal calculations.

Programmatic Use
Block Parameter: `InternalDataType`
Type: character vector
Values: data type string
Default: `'fixdt('double')'`

`Scaling for State Equation AX+BU` — Scaling for state equations

`2^0` (default) | scalar

Specify the scaling for the state equation **AX+BU**.

Programmatic Use
Block Parameter: `StateEqScaling`
Type: character vector
Values: scalar
Default: `'2^0'`

`Scaling for Output Equation CX+DU` — Scaling for output equations

`2^0` (default) | scalar

Specify the scaling for the output equation **CX+DU**.

Programmatic Use
Block Parameter: `InternalDataType`
Type: character vector
Values: scalar
Default: `'2^0'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean`[a] | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

a. This block is not recommended for use with Boolean signals.

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Discrete State-Space | State-Space

Topics

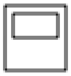
“States”
“Fixed-Point Numbers”

Introduced before R2006a

Floating Scope and Scope Viewer

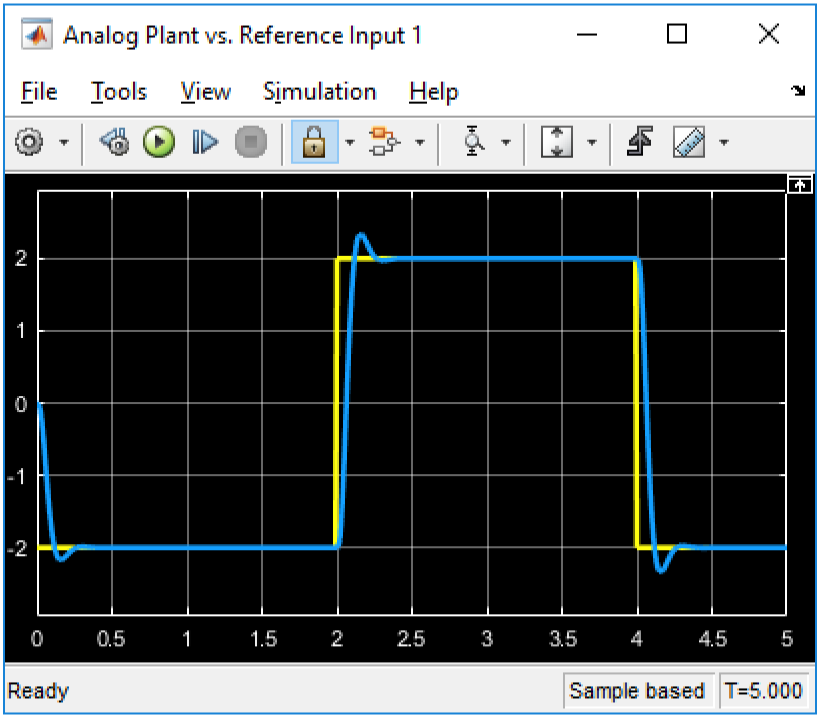
Display signals generated during simulation without signal lines

Library: Simulink / Sinks



Description

The Simulink Scope Viewer and Floating Scope block display time domain signals with respect to simulation time. The Scope Viewer and Floating Scope block have the same functionality as the Scope block, but they are not connected to signal lines. Use the button to add and display signals on a Floating Scope or Scope Viewer.



- Multiple *y*-axes (displays) — Display multiple *y*-axes with multiple input ports. All 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 scope is closed at the start of a simulation, scope data is still written to the scope during the simulation. If you open the scope after a simulation, the scope displays simulation results for input signals.

Oscilloscope features:

- Triggers — Set triggers on 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.

For information on controlling a Floating Scope block from the command line, see “Control Scope Blocks Programmatically” in the Simulink documentation.

Limitations

When you use model configuration parameters that optimize the simulation, such as **Signal storage reuse** and **Block reduction**, 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”.

- 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.
- When connected to a constant signal, the Scope Viewer plots a single point.
- Simulink messages are not supported for Floating Scope block and Scope Viewer.
- You cannot connect signals from ForEach subsystems.
- You cannot connect testpointed Stateflow signals to a Floating Scope. To connect these signals to a Scope Viewer, use the “Signal Selector”.

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 (array). 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) | on

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

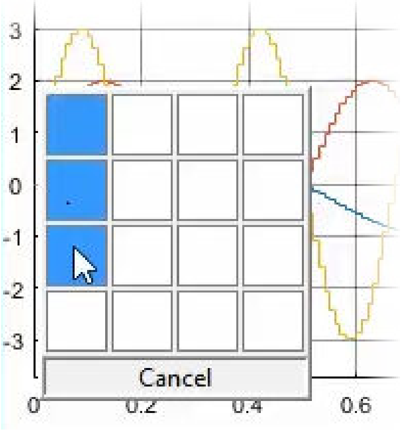
off (default) | on

Select this check box to display the block path in addition to the block name.

Layout — Number and arrangement of displays

1-by-1 display (default) | an arrangement of *m*-by-*n* displays

Specify number and arrangement of displays. To expand the layout grid beyond 4 by 4, click within the dialog box and drag. The maximum layout is 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 and then across.

Programmatic Use

See LayoutDimensions.

`Input processing` — Channel or element signal processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

- `Elements as channels (sample based)` — Process each element as a unique sample.
- `Columns as channels (frame based)` — Process signal values in a column 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) | `Auto` | `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` — Enter any 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 the data from the left.

- `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 in 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

To activate this property, set:

- "Show time-axis label" on page 1-0 to on.
- "Maximize axes" on page 1-0 to off.

The "Active display" on page 1-0 property determines which display is affected.

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

To activate this property, set "Time-axis labels" on page 1-0 to `All` or `Bottom Displays Only`.

The "Active display" on page 1-0 property determines which display is affected.

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. For layouts with multiple columns and rows, display numbers are mapped down and then across.

Programmatic Use

See “Active display” on page 1-0 .

`Title` — Display name

`%<SignalLabel>` (default) | character vector | string

Title for a 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

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]`.

The “Active display” on page 1-0 property determines which display is affected.

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

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]`.

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See YLimits.

`Y-label` — Y-axis label

none (default for Scope) | `'Amplitude'` (default for Time Scope) | character vector | string

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

If you select **Plot signals as magnitude and phase**, this property does not apply. The *y*-axes are labeled `Magnitude` and `Phase`.

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See YLabel.

Logging

`Limit data points to last` — Limit buffered data values

off (default) | on

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 — Specify the number of data values saved for each signal (5000 by default). 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**.

  Sometimes, selecting this parameter cause signals to be plotted for less than the entire time range of a simulation. For example, where the sample time is small. If a scope plots a portion of your signals, consider increasing the number of data values the simulation saves.
- 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 (default) | on

- On — Plot and log (save) scope data every N^{th} data point, where N is the decimation factor entered in the text box. The default decimation factor is 2. 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/Unlog Viewed Signals to Workspace` — Toggle logging

on | off

For signals selected with the Signal Selector, clicking this button toggles the state of the **Log signal data** check boxes in the Signals Properties dialog boxes.

Axes Scaling Properties

The Axes Scaling Properties 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 the *y*-axis range with the **Scale Y-axis Limits** toolbar button.
- `Auto` — Scale the *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 use for plotting

`80` (default) | integer between [1, 100]

Specify the percentage of the *y*-axis range used for plotting data. If you set this property to `100`, the 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 the maximum values of the *y*-axis range.
- `Center` — Center signals between the minimum and maximum values.
- `Bottom` — Align signals with the minimum values of the *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`, the data currently within the axes is scaled, not the entire signal in the data buffer.

`X-axis Data range (%)` — Percent of *x*-axis to use for plotting
`100` (default) | integer in the range [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 the maximum values of the *x*-axis range.
- `Center` — Center signals between the minimum and maximum values.
- `Bottom` — Align signals with the minimum values of the *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

solid line (default style) | 0.75 (default width) | yellow (default 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 shape.

Dependency

The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

HDL Code Generation

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

For more information about HDL code generation, see Floating Scope.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Blocks
Scope

Topics

“Scope Blocks and Scope Viewer Overview”
“Common Scope Block Interactions”
“Simulate a Model Interactively”
“Step Through a Simulation”
“Monitor Test Points in Stateflow Charts” (Stateflow)

Introduced in R2015b

Float Typecast

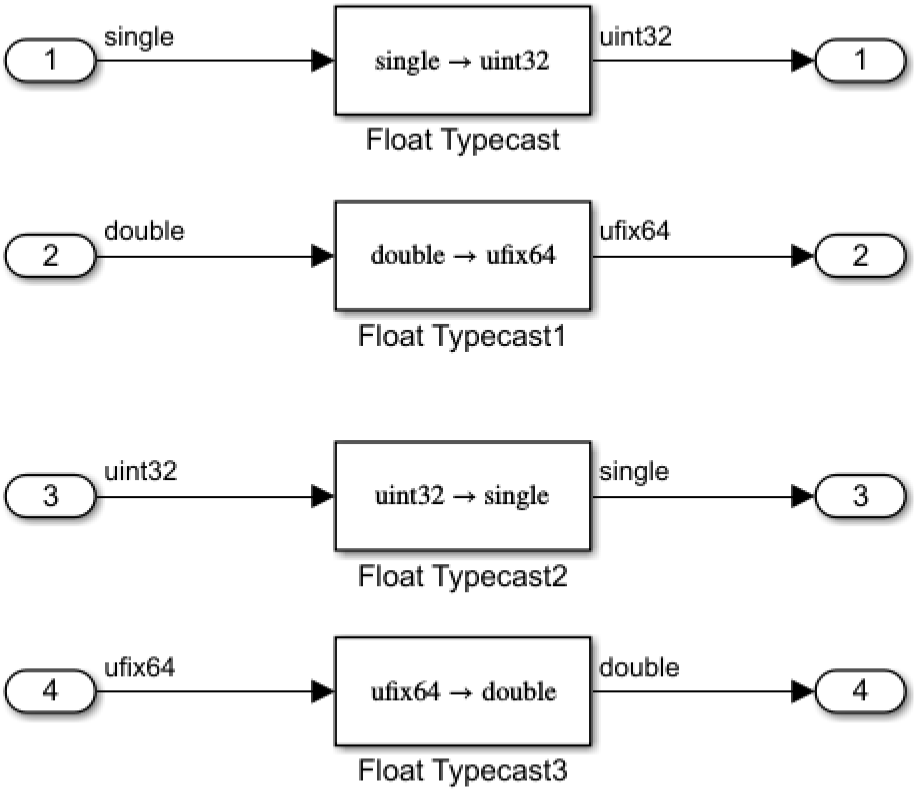
Typecast a floating-point type to an unsigned integer or vice versa

Library: HDL Coder / HDL Floating Point Operations / Float Typecast

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`

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 block in the `Native Floating Point` mode. For more information, see Float Typecast.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point 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 1-780

Concatenate Output

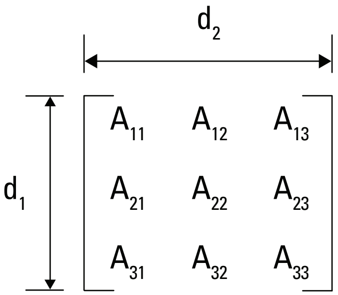
You define the dimension along which to concatenate the results by specifying the **Concatenation Dimension** in the Output Concatenation tab.

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.

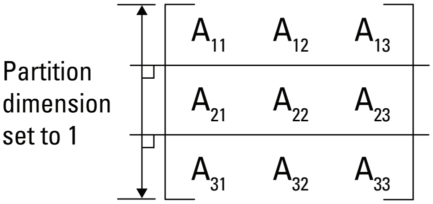
Select Partition Parameters

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.



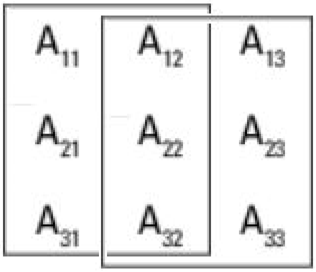
Matrix A decomposes into these three row vectors.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \end{bmatrix}$$
$$\begin{bmatrix} A_{21} & A_{22} & A_{23} \end{bmatrix}$$
$$\begin{bmatrix} A_{31} & A_{32} & A_{33} \end{bmatrix}$$

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.

$$\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix} \quad \begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix} \quad \begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix}$$

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).

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
Block Parameter: `InputPartition`
Type: character vector
Values: `'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
Block Parameter: `InputPartitionDimension`
Type: character vector
Values: `'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
Block Parameter: `InputPartitionWidth`
Type: character vector
Values: `'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
Block Parameter: `SubsysMaskParameterOffset`
Type: character vector
Values: `'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
Block Parameter: `OutputConcatenationDimension`
Type: character vector
Values: `'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

Block Parameter: `SubsysMaskParameterPartition`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Partition Dimension` — Specify dimension

1 (default) | integer

Specify the dimension through which to slice the input signal array as an integer greater than or equal to one. The resulting slices are perpendicular to the dimension that you specify. The slices also partition the array into subarrays or elements, as appropriate.

Programmatic Use

Block Parameter: `SubsysMaskParameterPartitionDimension`
Type: character vector
Values: `'1'` | `'<integer>'`
Default: `'1'`

`Partition Width` — Specify partition width

1 (default) | integer

Specify the width of each partition slice of the input signal as an integer greater than or equal to one.

Programmatic Use
Block Parameter: `SubsysMaskParameterPartitionWidth`
Type: character vector
Values: `'1'` | `'<integer>'`
Default: `'1'`

See Also

Blocks
For Each Subsystem | Subsystem

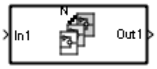
Topics
"Repeat an Algorithm Using a For Each Subsystem"
"Log Signals in For Each Subsystems"

Introduced in R2010a

For Each Subsystem

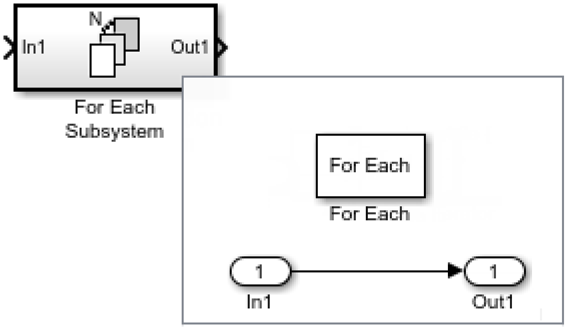
Subsystem that repeats execution on each element or subarray of input signal and concatenates results

Library: Simulink / 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 user data 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
You cannot log bus or an array of bus signals directly in the For Each subsystem.	Use one of these approaches: • 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.
You cannot log a signal inside a referenced model that is inside a For Each subsystem if either of these conditions exists: • 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.	For the first condition, use Accelerator mode. For the second condition, use Normal or Rapid Accelerator mode.
You cannot log the states of the blocks in a For Each subsystem .	Save and restore the simulation state.
You cannot use Normal mode to simulate a Model block inside a For Each subsystem.	Use Accelerator or Rapid Accelerator mode.
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.	Permute the signal dimensions to transform the partition dimension and the concatenation dimension to the highest nonsingleton dimension for n-D signals.

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

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a]
Direct Feedthrough	No

Multidimensional Signals	Yes[a]
Variable-Size Signals	No
Zero-Crossing Detection	No

[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 code generation support depends on block implementation.

HDL Code Generation

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

For more information about HDL code generation, see For Each Subsystem.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

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

Repeat execution of a subsystem during a time step for a specified number of iterations
Library: Ports & Subsystems

Description

The For Iterator block, when placed in a Subsystem block, repeats the execution of a subsystem during the current time step until an iteration variable exceeds the specified iteration limit. You can use this block to implement the block diagram equivalent of a `for` loop in a programming language.

The output of a For Iterator Subsystem block cannot be a function-call signal. Simulink displays an error message when the model updates.

Ports

Input

`Number of Iterations` — External value for iterator variable
scalar | vector, size 1 | matrix, size 1x1

- The input port accepts data of mixed numeric types.
- If the input port value is non-integer, 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 type, the overflow wraps around.

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

Output

`Iterator value` — Value of iterator variable during time step
scalar | vector, size 1 | matrix, size 1x1

Selecting the **Show iteration variable** parameter check box adds an output port to this block

Data Types: `double` | `int8` | `int16` | `int32`

Parameters

`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 and before the first iteration loop.

Programmatic Use
Block Parameter: `ResetStates`
Type: character vector
Values: `'held'` | `'reset'`
Default: `'held'`

`Iteration limit source` — Select source for number of iterations
`internal` (default) | `external`

Select source for number of iterations.

`internal`

Value of the **Iteration limit** parameter determines the number of iterations.

`external`

Value of the signal at the **N** port determines the number of iterations. The signal source must reside outside the For Iterator Subsystem block.

Dependencies

Selecting `internal` displays and enables the **Iteration limit** parameter. Selecting `external` adds an input port labeled **N**.

Programmatic Use

Block Parameter: `IterationSource`
Type: character vector
Values: `'internal'` | `'external'`
Default: `'internal'`

`Iteration limit` — Specify number of iterations

5 (default) | integer

Specify the number of iterations. 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.

`5`

Iterate blocks in the For Iterator Subsystem block `5` times.

integer

Specify an integer or a named constant variable.

Dependencies

To enable this parameter, select `internal` from the **Iteration limit source** drop-down list.

Programmatic Use

Block Parameter: `IterationLimit`
Type: character vector
Values: `'5'` | `'<integer>'`
Default: `'5'`

`Set Next i (iteration variable) externally` — Control display of input port

off (default) | on

Control display of an input port.

off

Remove input port.

on

Add input port labeled **Next_i** for connecting to 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.

Dependencies

To enable this parameter, select the **Show iteration variable** parameter which also displays an output port labeled **1:N**.

Programmatic Use

Block Parameter: `ExternalIncrement`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Show iteration variable` — Control display of output port

on (default) | off

Control the display of an output port with the current iterator value for a loop.

on

Add output port labeled **1:N** to the For Iterator block.

off

Remove output port.

Dependencies

Selecting this parameter enables the **Set next i (iteration variable) externally** parameter.

Programmatic Use

Block Parameter: `ShowIterationPort`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Index mode` — Select initial iteration number

`One-based` (default) | `Zero-based`

Select an initial iteration number of `0` or `1`.

`One-based`

Iteration number starts at one.

`Zero-based`

Iteration number starts at zero.

Programmatic Use
Block Parameter: `IndexMode`
Type: character vector
Values: `'One-based'` | `'Zero-based'`
Default: `'One-based'`

`Iteration variable data type` — Select data type

`int32` (default) | `int16` | `int8` | `double`

Set the data type for the iteration value output from the iteration number port.

`int32`

Set data type to `int32`.

`int16`

Set data type to `int16`.

`int8`

Set data type to `int8`.

`double`

Set data type to `double`.

Programmatic Use
Block Parameter: `IterationVariableDataType`
Type: character vector
Value: `'int32'` | `'int16'` | `'int8'` | `'double'`
Default: `'int32'`

See Also

Blocks
For Iterator Subsystem | Subsystem

Topics

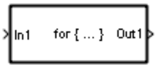
Iterator Subsystem Execution

Introduced before R2006a

For Iterator Subsystem

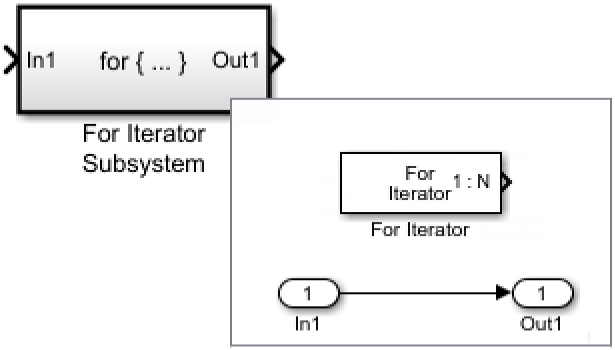
Subsystem that repeats execution during a simulation time step

Library: Simulink / Ports & Subsystems



Description

The For Iterator Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats the execution during a simulation time step for a specified number of 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.

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`

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

See Also

Blocks

For Iterator | Subsystem

Topics

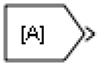
Iterator Subsystem Execution

Introduced before R2006a

From

Accept input from Goto block

Library: Simulink / Signal Routing

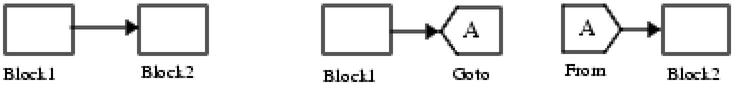


Description

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.

Ports

Output

`Port_1` — Signal from connected Goto block

scalar | vector | matrix | N-D array

Signal from connected Goto block, output with the same dimensions and data type as the input to the Goto block.

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

Parameters

`Goto Tag` — Tag of the Goto block that forwards its signal to this block

A (default) | `<More Tags...>` | ...

Specify the tag of the Goto block that forwards its signal to this From block. To change the tag, select a new tag from the 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 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.

Tip 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. To do so, use the **Rename All** button in the Goto block dialog box.

To find the relevant Goto block, use the **Goto Source** hyperlink in the From block dialog box.

Programmatic Use
Block Parameter: `GotoTag`
Type: character vector
Values: `'A'` | `...`
Default: `'A'`

`Update Tags` — Update list of visible tags

Updates the list of tags visible to this From block, including tags residing in libraries referenced by the model containing this From block. 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.

`Goto Source` — Path to connected Goto block

block path

Path of the Goto block connected to this From block. Clicking the path displays and highlights the Goto block in your model.

`Icon Display` — Text to display on block icon

`Tag` | `Tag and signal name` | `Signal name`

Specifies the text to display on the From block icon. The options are the block tag, the name of the signal that the block represents, or both the tag and the signal name.

Programmatic Use
Block Parameter: `IconDisplay`
Type: character vector
Values: `'Signal name'` | `'Tag'` | `'Tag and signal name'`
Default: `'Tag'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`

Multidimensional Signals	Yes
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™.

For more information on HDL code generation, see From.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Goto | Goto Tag Visibility

Topics

“Signal Label Propagation”

Introduced before R2006a

From File

Load data from MAT-file
Library: Simulink / Sources

untitled.mat

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”.

Ports

Output

`Port_1` — File data
scalar | vector | matrix | N-D array

MAT-file data, specified as 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.

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

Parameters

`File name` — Path or file name

`untitled.mat` (default) | path, or MAT-file name

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`.)

After you specify the **File name**, you can use the view button ( ) to preview the signal from the MAT-file. For more information, see “Preview Signal Data”.

Dependencies

Code generation for RSim target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, the MAT-file must contain a nonempty, finite, real matrix with at least two rows.

For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-821.

Programmatic Use
Block Parameter: `FileName`
Type: character vector
Values: MAT-file name
Default: `'untitled.mat'`

`Output data type` — Output data type

`Inherit: auto` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class_name>` | `Bus: <bus_object>` | `<data type expression>`

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”.

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, dimensions, and complexity are the same for the data and for the bus object.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

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>' | '<data type expression>'`
Default: `'Inherit: auto'`

`Sample time` — Sampling period and offset

`0` | scalar | vector

Specify 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”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'0'`

`Data extrapolation before first data point` — Extrapolation method for simulation times before initial time stamp in MAT-file

`Linear extrapolation` (default) | `Hold first value` | `Ground value`

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
`Linear extrapolation`	(Default) If the MAT-file contains only one sample, then the From File block outputs the corresponding data value. If the MAT-file contains more than one sample, then the From File block linearly extrapolates using the first two samples: • 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: • Upcasts the data to `double` • Performs linear extrapolation (as described for `double` data) • Downcasts the extrapolated data value to the original data type 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.
`Hold first value`	Uses the first data value in the file
`Ground value`	Uses a value that depends on the data type of MAT-file sample data values: • 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

Dependencies

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, you must set this parameter to `Linear extrapolation`. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-821.

Programmatic Use
Block Parameter: `ExtrapolationBeforeFirstDataPoint`
Type: character vector
Values: `'Linear extrapolation' | 'Hold first value' | 'Ground value'`
Default: `'Linear extrapolation'`

`Data interpolation within time range` — Interpolation method for simulation times that fall between two time stamps in the MAT-file
`Linear interpolation` (default) | `Zero order hold`

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the MAT-file. Choose one of these interpolation methods.

Method	Description
`Linear interpolation`	(Default) The From File block interpolates using the two corresponding MAT-file samples: • For `double` data, linearly interpolates the value using the two corresponding samples • For `Boolean` data, uses `false` for the first half of the sample and `true` for the second half. • For a built-in data type other than `double` or `Boolean`, the From File block: • Upcasts the data to `double` • Performs linear interpolation, as described for `double` data • Downcasts the interpolated value to the original data type
`Zero order hold`	Uses the data from the first of the two samples

Limitations

You cannot use the `Linear interpolation` option with enumerated (`enum`) data. All signals in a bus use the same interpolation setting. If any signal in a bus uses `enum` data, then you cannot use the `Linear interpolation` option.

Dependencies

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, you must set this parameter to `Linear interpolation`. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-821.

Programmatic Use

Block Parameter: `InterpolationWithinTimeRange`
Type: character vector
Values: `'Linear interpolation' | 'Zero order hold'`
Default: `'Linear interpolation'`

`Data extrapolation after last data point` — Extrapolation method for simulation times after last time stamp in MAT-file

`Linear extrapolation` (default) | `Hold last value` | `Ground value`

The extrapolation method for a simulation time hit that occurs after the last time stamp in the MAT-file. Choose one of these extrapolation methods.

Method	Description
`Linear extrapolation`	(Default) If the MAT-file contains only one sample, then the From File block outputs the corresponding data value. If the MAT-file contains more than one sample, then the From File block linearly extrapolates using data values of the last two samples: • For `double` data, extrapolates the value using the last two samples. • For `Boolean` data, outputs the first data value. • For built-in data types other than double or `Boolean`: • Upcasts the data to `double` • Performs linear extrapolation, as described for `double` data • Downcasts the extrapolated value to the original data type
`Hold last value`	Uses the last data value in the file
`Ground value`	Uses a value that depends on the data type of MAT-file sample data values: • 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

Limitations

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.

Dependencies

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, you must set this parameter to `Linear extrapolation`. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-821.

Programmatic Use

Block Parameter: `ExtrapolationAfterLastDataPoint`
Type: character vector
Values: `'Linear extrapolation' | 'Hold last value' | 'Ground value'`
Default: `'Linear extrapolation'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`off` (default) | `on`

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 these parameters:

- **Data extrapolation before first data point**

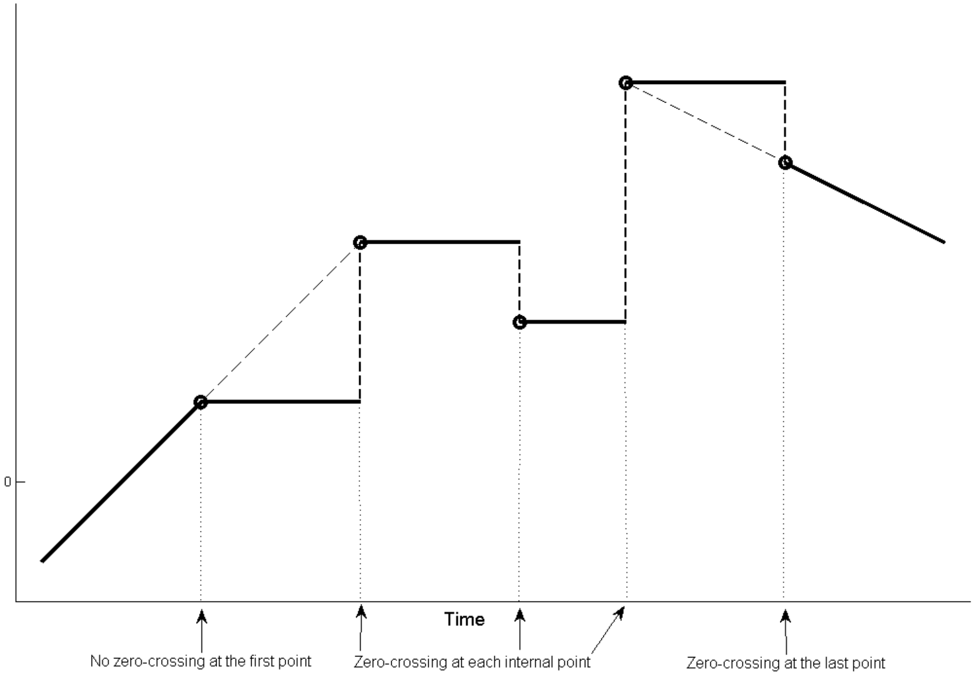
- **Data interpolation within time range**
- **Data extrapolation after last data point**

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 apply: • **Data extrapolation after last data point** is set to `Ground value`. • **Data interpolation within time range** is set to `Zero-order hold`.

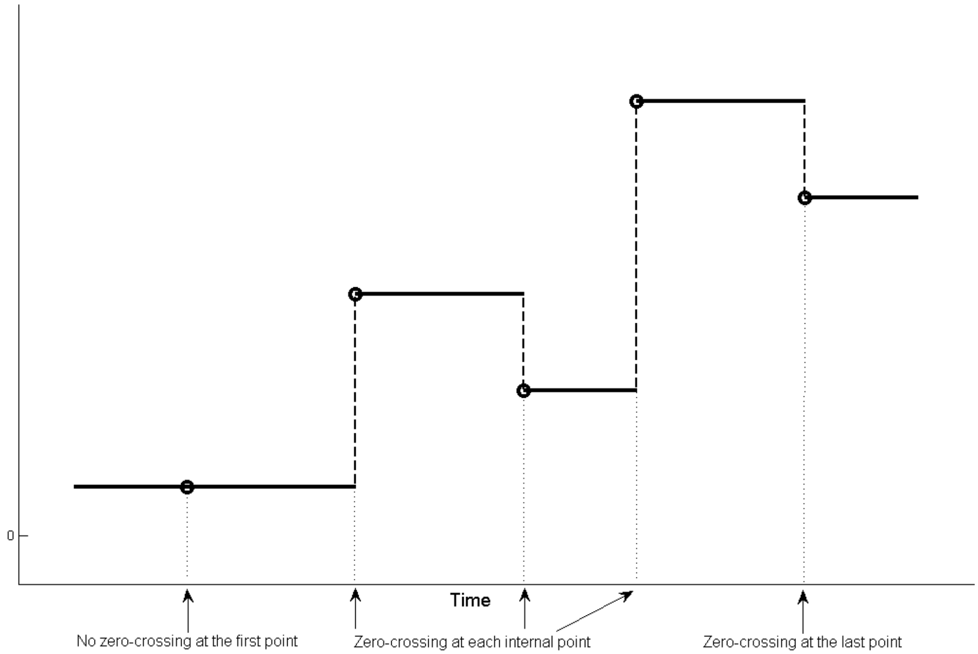
This figure illustrates zero-crossing detection for data accessed by a From File block that has these settings:

- **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`



Dependencies

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, clear this check box. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-821.

Programmatic Use

Block Parameter: `ZeroCross`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`[a] | `enumerated` | `bus`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

[a] Supports up to 32-bit fixed-point data types.

Extended Capabilities

C/C++ Code Generation

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

- Not recommended for production code.
- Code generation for RSim target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.
- 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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Supports up to 32-bit fixed-point data types.

See Also

From Spreadsheet | From Workspace | To File | To Workspace

Topics

"Overview of Signal Loading Techniques"
"Comparison of Signal Loading Techniques"
"Create Data for a From File Block"
"Load Data Using the From File Block"
"Load Signal Data That Uses Units"
"Specify Source for Data in Model Workspace"

Introduced before R2006a

From Spreadsheet

Read data from spreadsheet
Library: Simulink / Sources

untitled.xlsx
Sheet1

Description

The From Spreadsheet block reads data from Microsoft Excel® (all platforms) or CSV (MicrosoftWindows® platform with Microsoft Office installed only) spreadsheets and outputs the data as a signal. The From Spreadsheet block does not support Microsoft Excel spreadsheet charts.

The From Spreadsheet icon displays the spreadsheet file name and sheet name specified in the block **File name** and **Sheet name** parameters.

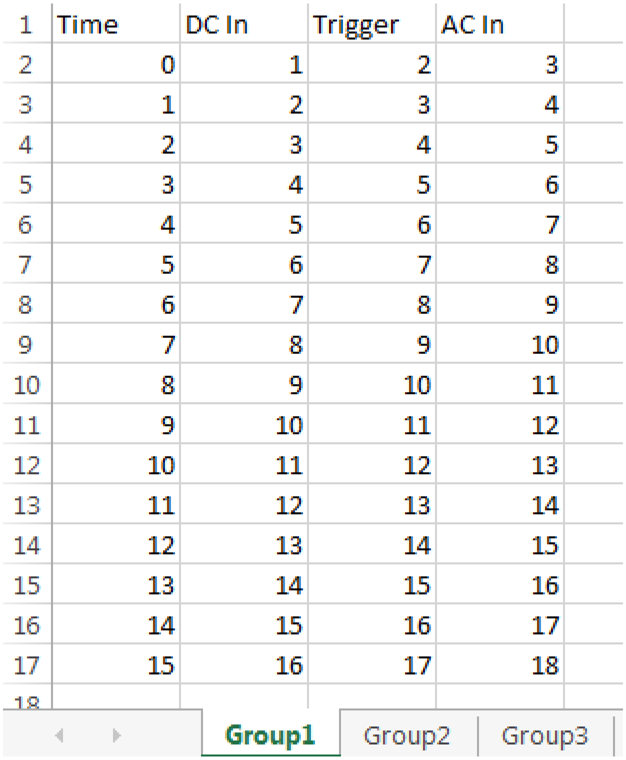
Storage Formats

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				

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 1-827. 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 1-825.

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 these parameters:

- “Data extrapolation before first data point” on page 1-0
- “Data interpolation within time range” on page 1-0
- “Data extrapolation after last data point” on page 1-0

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.

Ports

Output

`Port_1` — Data from spreadsheet

scalar | vector | matrix

Incremental data from the specified spreadsheet.

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 1-827. 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 1-825.

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.

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

Parameters

`File name` — Full path and file name

`untitled.xlsx` (default) | full path and 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.

Programmatic Use
Block Parameter: `FileName`

Type: character vector
Value: full path and file name
Default: `'untitled.xlsx'`

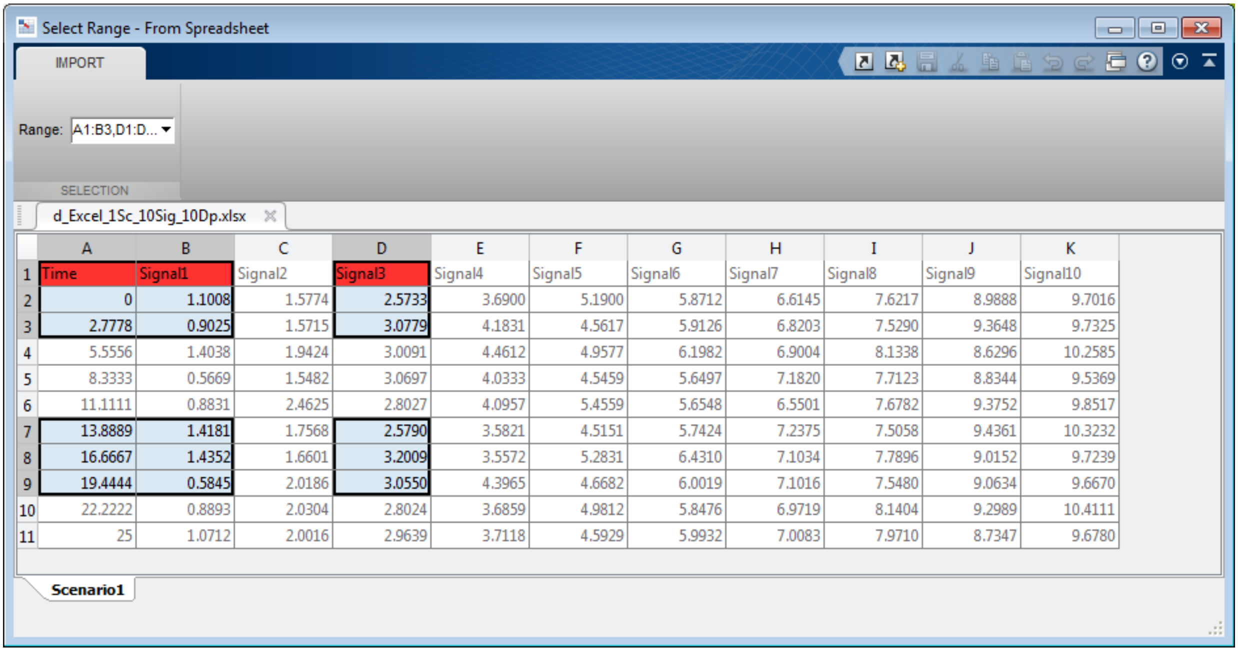
Sheet name — Name of sheet in spreadsheet

`Sheet1` (default) | 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.

If your spreadsheet is the CSV format, the block populates this parameter with the name of the CSV file without the extension. Do not change this value.

To open the sheet, click . In the sheet, you can select the range of data by dragging over the desired range of values.



Range: A1:B3,D1:D...

	A	B	C	D	E	F	G	H	I	J	K
1	Time	Signal1	Signal2	Signal3	Signal4	Signal5	Signal6	Signal7	Signal8	Signal9	Signal10
2	0	1.1008	1.5774	2.5733	3.6900	5.1900	5.8712	6.6145	7.6217	8.9888	9.7016
3	2.7778	0.9025	1.5715	3.0779	4.1831	4.5617	5.9126	6.8203	7.5290	9.3648	9.7325
4	5.5556	1.4038	1.9424	3.0091	4.4612	4.9577	6.1982	6.9004	8.1338	8.6296	10.2585
5	8.3333	0.5669	1.5482	3.0697	4.0333	4.5459	5.6497	7.1820	7.7123	8.8344	9.5369
6	11.1111	0.8831	2.4625	2.8027	4.0957	5.4559	5.6548	6.5501	7.6782	9.3752	9.8517
7	13.8889	1.4181	1.7568	2.5790	3.5821	4.5151	5.7424	7.2375	7.5058	9.4361	10.3232
8	16.6667	1.4352	1.6601	3.2009	3.5572	5.2831	6.4310	7.1034	7.7896	9.0152	9.7239
9	19.4444	0.5845	2.0186	3.0550	4.3965	4.6682	6.0019	7.1016	7.5480	9.0634	9.6670
10	22.2222	0.8893	2.0304	2.8024	3.6859	4.9812	5.8476	6.9719	8.1404	9.2989	10.4111
11	25	1.0712	2.0016	2.9639	3.7118	4.5929	5.9932	7.0083	7.9710	8.7347	9.6780

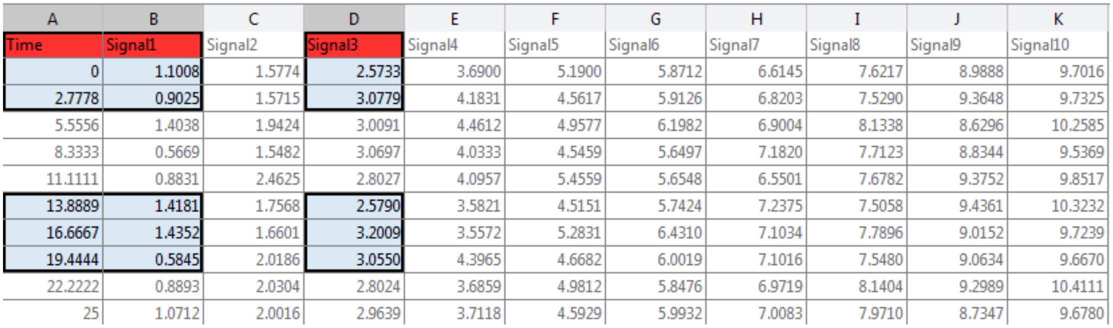
Alternatively, you can select the range of data by specifying the range of values in the **Range** parameter.

Programmatic Use
Block Parameter: `SheetName`
Type: character vector
Value: Sheet name
Default: `'Sheet1'`

Range — Cell range

entire range of used cells in sheet (default) | `A1:B3,D1:D3,A7:B9,D7:D9` | comma-separated list of *`column`*`:`*`row`*

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.

An alternate to using the **Range** parameter is to open the sheet, by clicking . In the sheet, you can select the range of data by dragging over the desired range of values.

Programmatic Use
Block Parameter: `Range`
Type: character vector
Value: Cell range
Default: `''`

`Output data type` — Output data type

`Inherit: auto` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class_name>` | `Bus: <bus_object>` | `<data type expression>`

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**.

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.

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”.

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>'` | `'<data type expression>'`
Default: `'Inherit: auto'`

`Treat first column as` — Time or data

`Time` (default) | `Data`

Select how the block should treat the first column of the spreadsheet:

- `Time` — Treat first column as time.
- `Data` — Treat first column as data.

Dependencies

When you select `Data`, the block disables:

- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

And enables:

- **Output after last data point**

Programmatic Use
Block Parameter: `TreatFirstColumnAs`
Type: character vector
Value: `'Time' | 'Data'`
Default: `'Time'`

`Sample time` — Sampling period and offset

`0` (default) | scalar | vector

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".

Programmatic Use
Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'0'`

`Data extrapolation before first data point` — Extrapolation method for simulation times before initial time stamp in MAT-file

`Linear extrapolation` (default) | `Hold first value` | `Ground value`

Extrapolation method that Simulink uses for a simulation time hit that is before the first time stamp in the spreadsheet. Choose one of these extrapolation methods.

Method	Description
`Linear extrapolation`	(Default) If the spreadsheet contains only one sample, the From Spreadsheet block outputs the corresponding data value. If the spreadsheet contains more than one sample, the From Spreadsheet block linearly extrapolates using the first two samples: • 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`: • Upcasts the data to `double` • Performs linear extrapolation (as described above for `double` data) • Downcasts the extrapolated data value to the original data type You cannot use the `Linear extrapolation` option with enumerated (`enum`) data.
`Hold first value`	Uses the first data value in the file
`Ground value`	Uses a value that depends on the data type of spreadsheet sample data values: • 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

Dependencies

To enable this parameter, set **Treat first column as** to `Time`.

Programmatic Use
Parameter: `ExtrapolationBeforeFirstDataPoint`

Type: character vector
Values: `'Linear extrapolation' | 'Hold first value' | 'Ground value'`
Default: `'Linear extrapolation'`

`Data interpolation within time range` — Interpolation method for simulation times that fall between two time stamps in the MAT-file

`Linear interpolation` (default) | `Zero order hold`

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`	(Default) The From Spreadsheet block interpolates using the two corresponding spreadsheet samples: • For `double` data, linearly interpolates the value using the two corresponding samples • For `Boolean` data, uses `false` for the first half of the sample 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 above for `double` data) • Downcasts the interpolated value to the original data type You cannot use the `Linear interpolation` option with enumerated (`enum`) data.
`Zero order hold`	Uses the data from the first of the two samples

Dependencies

To enable this parameter, set **Treat first column as** to `Time`.

Programmatic Use

Parameter: `InterpolationWithinTimeRange`
Type: character vector

Values: `'Linear interpolation' | 'Zero order hold'`
Default: `'Linear interpolation'`

`Data extrapolation after last data point` — Extrapolation method for simulation times after last time stamp in MAT-file
`Linear extrapolation` (default) | `Hold last value` | `Ground value`

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`	(Default) If the spreadsheet contains only one sample, the From Spreadsheet block outputs the corresponding data value. If the spreadsheet contains more than one sample, the From Spreadsheet block linearly extrapolates using data values of the last two samples: • 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`: • Upcasts the data to `double`. • Performs linear extrapolation (as described above for `double` data). • Downcasts the extrapolated value to the original data type. You cannot use the `Linear extrapolation` option with enumerated (`enum`) data.
`Hold last value`	Uses the last data value in the file

Method	Description
`Ground value`	Uses a value that depends on the data type of spreadsheet sample data values: • 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

Dependencies

To enable this parameter, set **Treat first column as** to `Time`.

Programmatic Use
Parameter: `ExtrapolationAfterLastDataPoint`
Type: character vector
Values: `'Linear extrapolation' | 'Hold last value' | 'Ground value'`
Default: `'Linear extrapolation'`

`Output after last data point` — Action after last data point

`Repeating sequence` (default) | `Hold final value` | `Ground value`

Select action after last data point:

- `Repeating sequence` — Repeat the sequence by reading the data from the first row of the range specified in **Range**
- `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**.

Dependencies

To enable this parameter, set **Treat first column as** to `Data`.

Programmatic Use
Parameter: `OutputAfterLastPoint`
Type: character vector
Values: `'Repeating sequence' | 'Hold final value' | 'Ground value'`
Default: `'Repeating sequence'`

Enable zero-crossing detection — Enable zero-crossing detection

off (default) | on

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. In this context, zero-crossing is used 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:

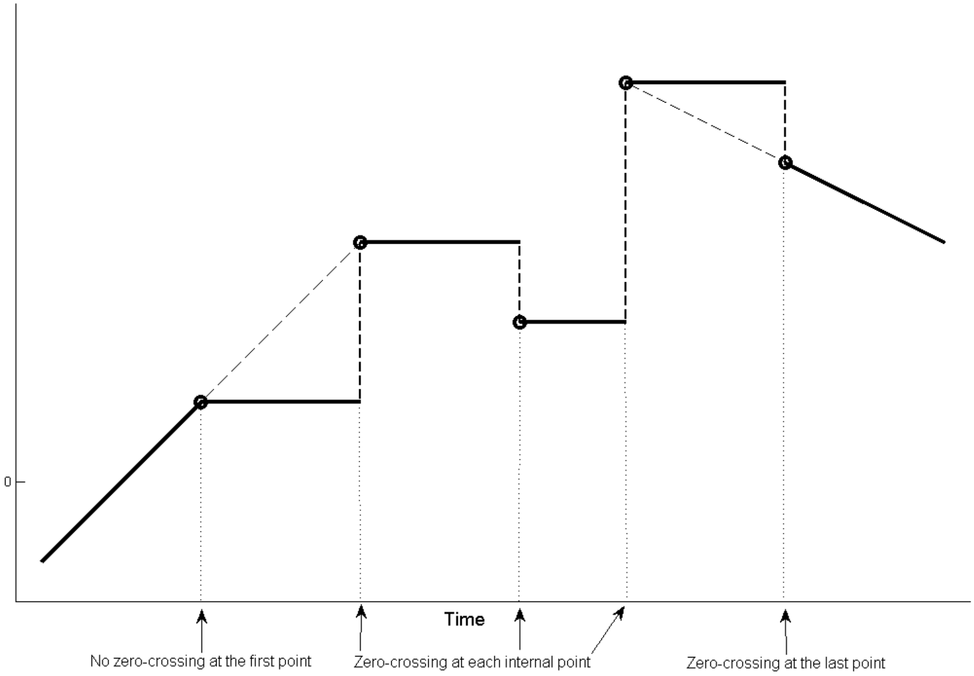
- Zero-crossing occurs for the first time stamp
- For time stamps between the first and last time stamp
- For the last time stamp

Time Stamp	When Zero-Crossing Detection Occurs
First	**Data extrapolation before first data point** is set to Ground value.

Time Stamp	When Zero-Crossing Detection Occurs
Between first and last	**Data interpolation within time range** is set to `Zero-order hold`.
Last	One or both of these settings occur: • **Data extrapolation after last data point** is set to `Ground value`. • **Data interpolation within time range** is set to `Zero-order hold`.

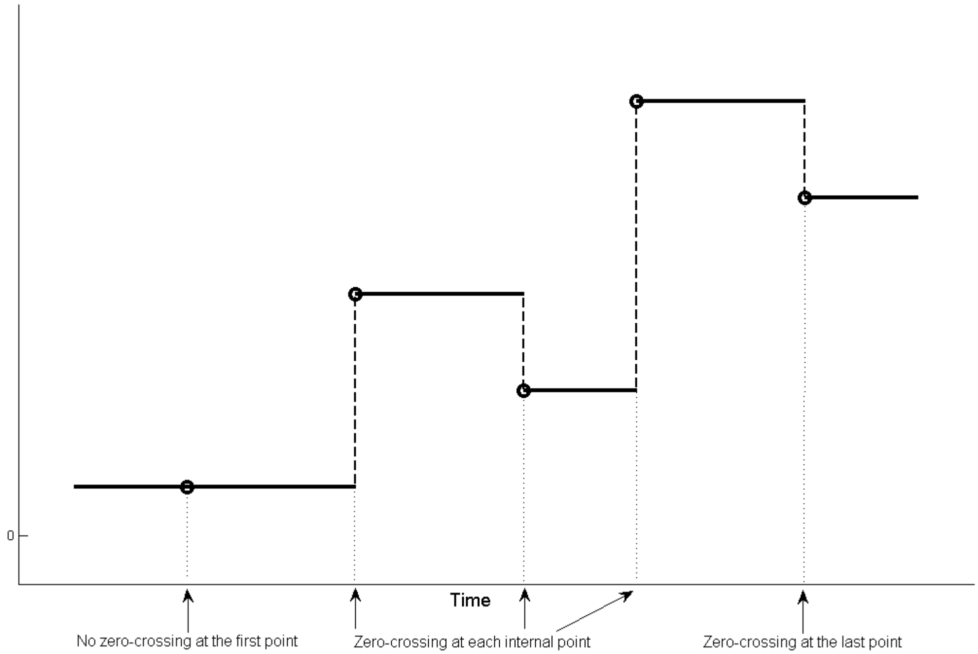
The following figure illustrates zero-crossing detection for data accessed by a From Spreadsheet block that has these settings:

- **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 these 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`



Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`[a] | `enumerated`
Direct Feedthrough	`No`

Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

[a] Supports up to 32-bit fixed-point data types.

Extended Capabilities

C/C++ Code Generation

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

- Not recommended for production code.
- Code generation for RSim target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.
- 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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Supports up to 32-bit fixed-point data types.

See Also

From File | From Workspace

Topics

“Overview of Signal Loading Techniques”
“Comparison of Signal Loading Techniques”
“Spreadsheets” (MATLAB)

Introduced in R2015b

From Workspace

Load signal data from workspace
Library: Simulink / Sources

Description

The From Workspace block reads signal data from a workspace and outputs the data as a signal.

The block 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 Workspace Data

In the From Workspace block dialog box, use the **Data** parameter to specify the workspace data to load. You can specify a MATLAB expression (for example, the name of a variable in the MATLAB workspace) that evaluates to one of these options:

- 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 contributes to the fundamental design of the model in the Design Data section of the dictionary. Design data includes 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')`.

Ports

Output

`Port_1` — Workspace data

scalar | vector | matrix

Signal created from workspace data. The block outputs real or complex signals of any type that Simulink supports, including fixed-point and enumerated data types.

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

Parameters

`Data` — Workspace data to load

`simin` (default) | `timeseries` object | structure of `timeseries` objects | structure | 2-D matrix

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

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`.

For additional information, see "Specify the Workspace Data".

Programmatic Use
Block Parameter: `VariableName`
Type: character vector
Values: `timeseries` object | structure of `timeseries` objects | structure | 2-D matrix
Default: `'simin'`

`Output data type` — Output data type

`Inherit: auto` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class_name>` | `Bus: <bus_object>` | `<data type expression>`

Required data type for the workspace data 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".

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>'` | `'<data type expression>'`
Default: `'Inherit: auto'`

`Sample time` — Sample rate of loaded data

`0` (default) | scalar | vector

Sample rate of loaded workspace data. For details, see "Specify Sample Time".

Command-Line Information
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'0'`

`Interpolate data` — Interpolate or use last known data point

`on` (default) | `off`

When you select this option, the block performs linear interpolation at time hits for which no corresponding workspace data exist. 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.

Programmatic Use
Block Parameter: `Interpolate`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | `off`

When you select **Enable zero-crossing detection**, and 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.

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Form output after final data value by` — Determine block output after final time hit

`Extrapolation` (default) | `Setting to zero` | `Holding final value` | `Cyclic repetition`

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**

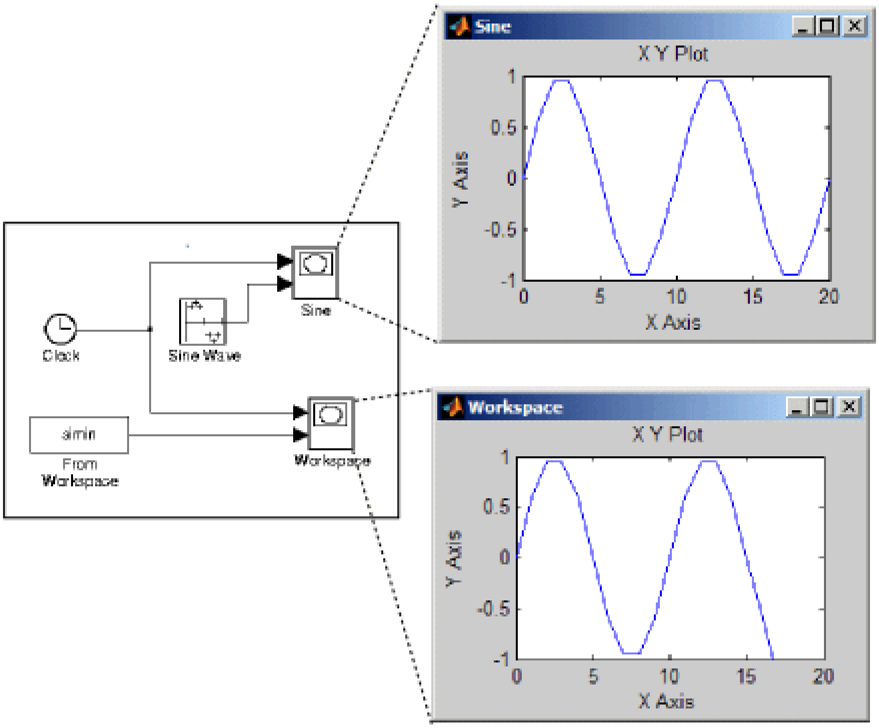
This table lists the block output, based on the values of the two options.

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 Wave 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.

Programmatic Use
Block Parameter: `OutputAfterFinalValue`
Type: character vector
Values: `'Extrapolation'` | `'Setting to zero'` | `'Holding final value'` | `'Cyclic repetition'`
Default: `'Extrapolation'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`[a] | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`Yes`

[a] Supports input via fi objects created using Fixed-Point Designer.

Extended Capabilities

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Supports input via `fi` objects created using Fixed-Point Designer.

See Also

From File | From Spreadsheet | To File | To Workspace

Topics

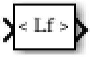
“Overview of Signal Loading Techniques”
“Comparison of Signal Loading Techniques”
“Determine Where to Store Variables and Objects for Simulink Models”
“Use From Workspace Block for Test Case”
“Load Data Using the From Workspace Block”
“Load Signal Data That Uses Units”
“Load Signal Data for Simulation”

Introduced before R2006a

Function-Call Feedback Latch

Break feedback loop involving data signals between function-call blocks

Library: Simulink / Ports & Subsystems

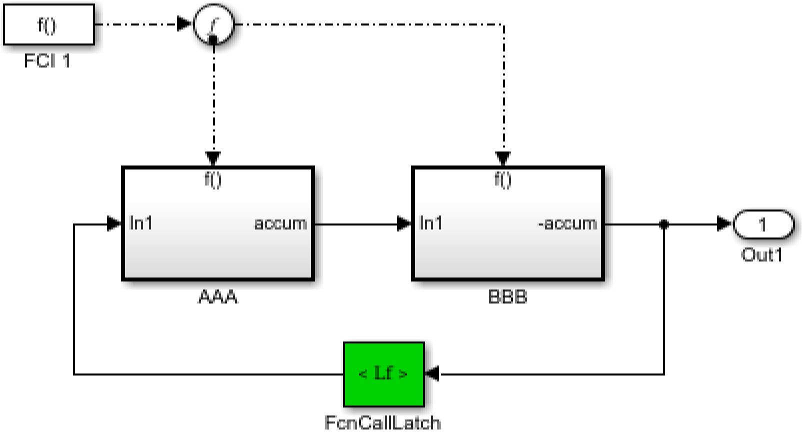


Description

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**

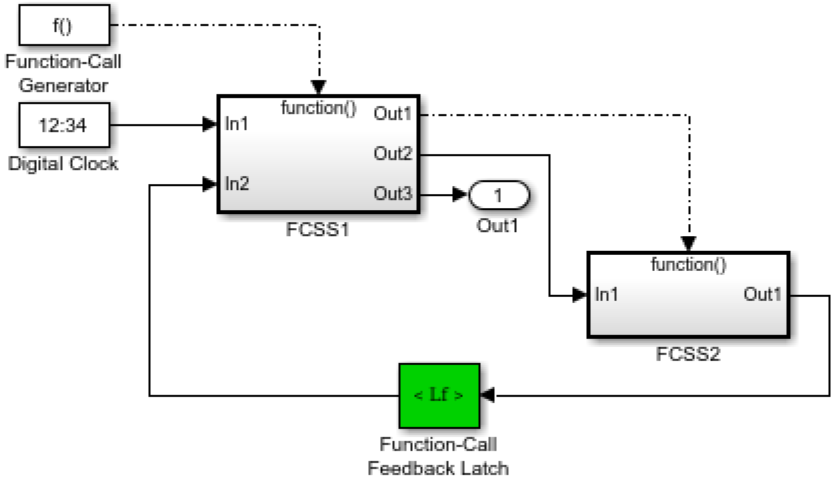
  Place the Function-Call Feedback Latch block on the feedback signal between the branched blocks. As a result, the latch block delays the signal at the input of the destination function-call block, and the destination function-call block executes prior to the source function-call block of the latch block.



To run this model, see "Function-Call Blocks Connected to Branches of the Same Function-Call Signal" on page 14-179.

- **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).



  To run this model, see “Function-Call Feedback Latch on Feedback Signal Between Child and Parent” on page 14-180.

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`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`

Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Function-Call Subsystem | Trigger

Topics

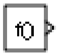
“Using Function-Call Subsystems”

Introduced in R2011a

Function-Call Generator

Provide function-call signal to control execution of a subsystem or model

Library: Simulink / 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) | `Ts` | `[Ts, To]`

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
Values: `'-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
Block Parameter: `numberOfIterations`
Type: character vector
Values: `'1'` | `'<integer>'`
Default: `'1'`

Block Characteristics

Data Types	
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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: Simulink / 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.

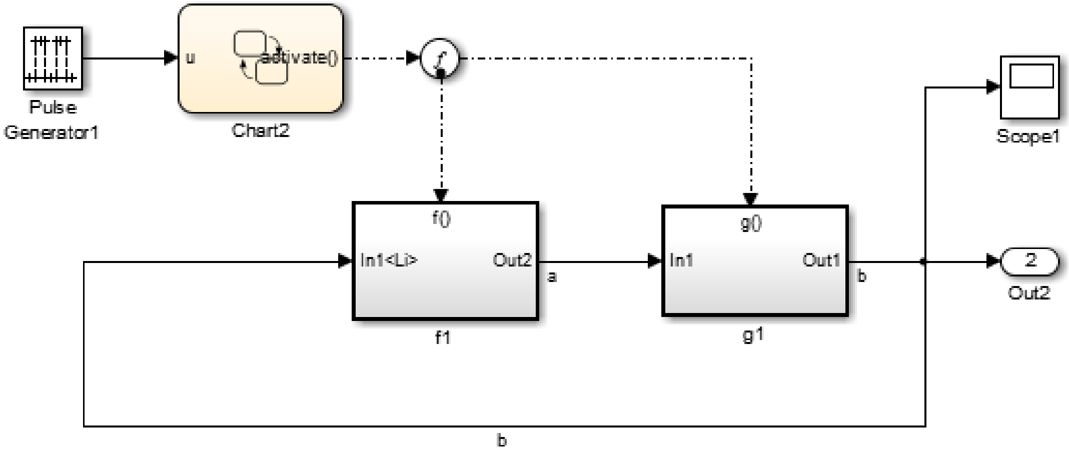
The function-call subsystem or function-call model connected to the output port of the Function-Call Split block that is marked with a dot execute before the subsystems or models connected to other output ports. If data dependencies between subsystems or models do not support the specified execution 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
Block Parameter: `IconShape`
Type: character vector
Values: `'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
Block Parameter: `NumOutputPorts`
Type: character vector
Values: `'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
Block Parameter: `OutputPortLayout`
Type: character vector
Values: `'default'` | `'reverse'`
Default: `'default'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem

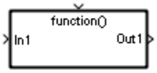
Topics

“Using Function-Call Subsystems”

Introduced in R2010a

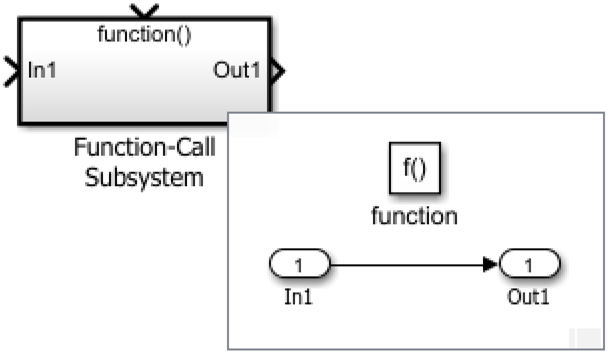
Function-Call Subsystem

Subsystem whose execution is triggered by external function call input
Library: Simulink / 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`

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[a] | `string`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`Yes`[a]
Zero-Crossing Detection	`No`

[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 code generation support depends on block implementation.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Topics

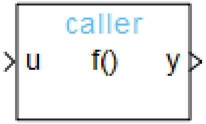
"Using Function-Call Subsystems"

Introduced before R2006a

Function Caller

Call Simulink or exported Stateflow function

Library: Simulink / 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 Function Blocks”.

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 Function Blocks”.

This specification must match the Simulink Function block data type specified with the **Data type** parameter of the Argument Outport 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
Values: `'-1'` | `'Ts'` | `'[Ts, To]'`
Default: `'-1'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

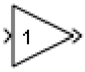
Argument Inport | Argument Outport | Chart | Function Caller | Function-Call Subsystem | Inport | MATLAB Function | Outport | Subsystem | Trigger

Topics

"Simulink Functions"
"Scoped Simulink Function Blocks in Models"
"Using Simulink Function Blocks and Exported Stateflow Functions"

Gain

Multiply input by constant
Library: Simulink / Commonly Used Blocks
Simulink / 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 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.

Gain is converted from doubles to the data type 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

The Gain block accepts real or complex-valued scalar, vector, or matrix input. The Gain block supports fixed-point data types. If the input of the Gain block is real and gain is complex, the output is complex.

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

Output

Port_1 — Input multiplied by gain

scalar | vector | matrix

The Gain block outputs the input multiplied by a constant gain value. When the input to the Gain block is real and gain is complex, the output is complex.

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

Parameters

Main

Gain — Value by which to multiply the input

1 (default) | real or complex-valued scalar, vector, or matrix

Specify the value by which to multiply the input. Gain can be a real or complex-valued scalar, vector, or matrix.

Programmatic Use
Block Parameter: Gain
Type: character vector
Values: '1' | real- or complex-valued scalar, vector, or matrix
Default: '1'

Multiplication — Specify the multiplication mode

Element-wise(K.*u) (default) | Matrix(K*u) | Matrix(u*K) | Matrix(K*u) (u vector)

Specify one of these multiplication modes:

- Element-wise(K.*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*u) — The input and gain are matrix-multiplied with the input as the second operand.
- Matrix(u*K) — The input and gain are matrix-multiplied with the input as the first operand.

- Matrix(K*u) (u vector) — The input and gain are matrix multiplied with the input as the second operand. This mode is identical to Matrix(K*u), except for how dimensions are determined.

  Suppose that K is an m-by-n matrix. Matrix(K*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*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.

Programmatic Use
Parameter: Multiplication
Type: character vector
Value: 'Element-wise(K.*u)' | 'Matrix(K*u)' | 'Matrix(u*K)' | 'Matrix(K*u) (u vector)'
Default: 'Element-wise(K.*u)'

Sample time — Specify sample time as a value other than -1

-1 (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '-1'

Signal Attributes

Output minimum — Minimum output value for range checking

[] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `Inherit: Same as input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

When you select an inherited option, the block exhibits these behaviors:

- `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)`, in other words, 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.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Same as input'` | `'Inherit: Inherit via back propagation'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'`, `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Integer rounding mode — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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	Rationale	Impact on Overflows	Example
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Parameter Attributes

`Parameter minimum` — Specify the minimum value of gain

`[]` (default) | scalar

Specify the minimum value of gain. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)

- Automatic scaling of fixed-point data types

Programmatic Use
Block Parameter: `ParamMin`
Type: character vector
Value: scalar
Default: `'[ ]'`

`Parameter maximum` — Specify the maximum value of gain

`[]` (default) | scalar

Specify the maximum value of gain. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Programmatic Use
Block Parameter: `ParamMax`
Type: character vector
Value: scalar
Default: `'[ ]'`

`Parameter data type` — Specify the data type of the Gain parameter

`Inherit: Inherit via internal rule` (default) | `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)` | `<data type expression>`

Specify the data type of the **Gain** parameter.

Programmatic Use
Block Parameter: `ParamDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Same as input'` | `'Inherit: Inherit via back propagation'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'`, `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	Yes
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™.

For more information on HDL code generation, see Gain.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Product | Slider Gain

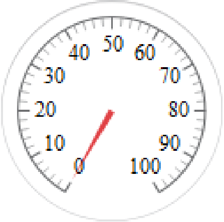
Topics

“Model a Continuous System”

Introduced before R2006a

Gauge

Display input value on circular scale
Library: Simulink / Dashboard



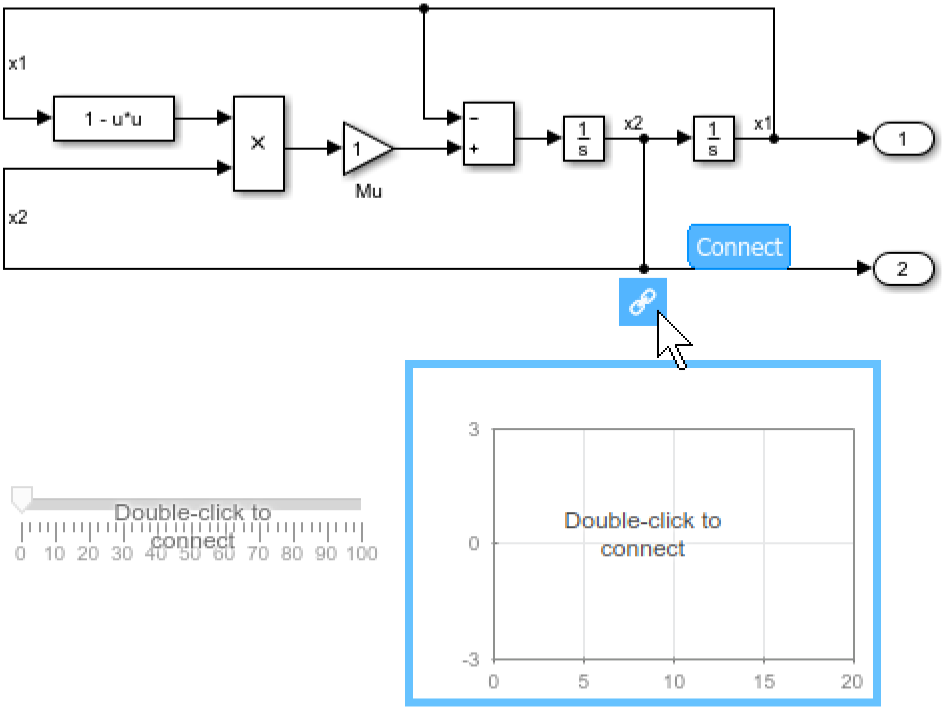
Description

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 change the appearance of the dial to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

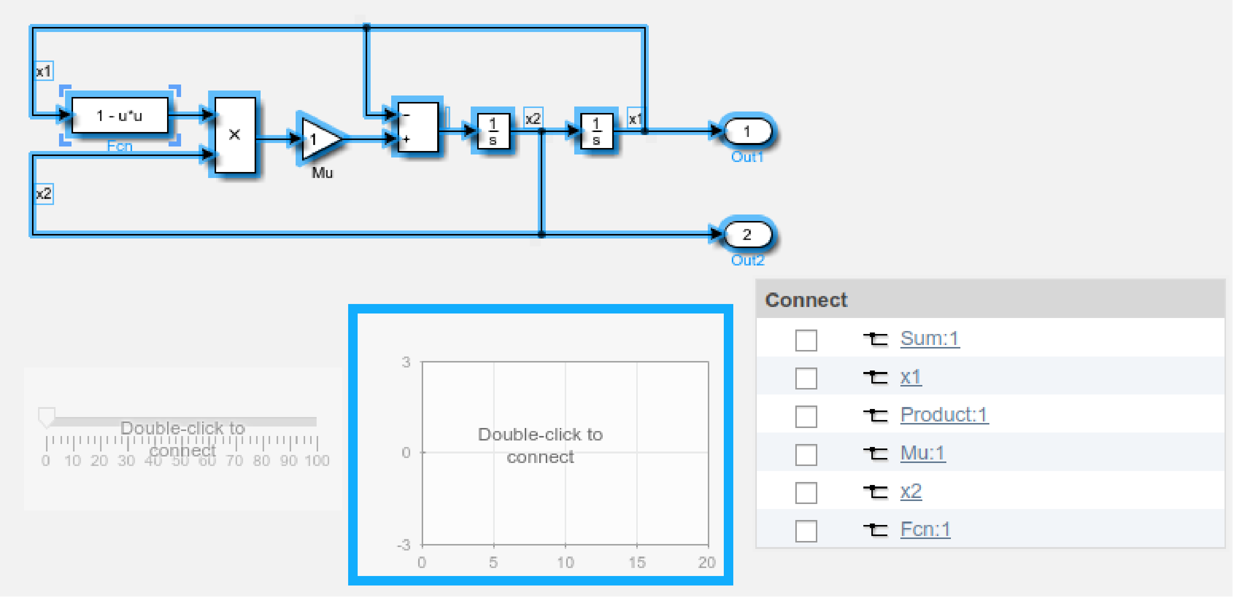
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect	
□	Sum:1
□	x1
□	Product:1
□	Mu:1
□	x2
□	Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced 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.

Parameters

`Connection` — Select a signal to connect and display

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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 arc. 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 arc. 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 arc. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

`Scale Colors` — Color indications on gauge arc

colors for arc ranges

Color specifications for ranges on the arc. 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.

See Also

Custom Gauge | Half Gauge | Linear Gauge | Quarter Gauge

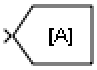
Topics

“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: Simulink / 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` (default) — From and Goto blocks that use the same tag must be in the same subsystem. A local tag name is enclosed in brackets (`[]`).
- `scoped` — From and Goto blocks that use 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, in other words, the boundary of an atomic, conditionally executed, or function-call subsystem or a model reference. A scoped tag name is enclosed in braces (`{}`).
- `global` — 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array

Input signal to be passed to the corresponding From block, specified as a scalar, vector, matrix, or N-D array.

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

Parameters

`Goto tag` — Block identifier

A (default) | ...

The Goto block identifier. This parameter identifies the Goto block whose scope is defined in this block.

Programmatic Use
Block Parameter: `GotoTag`
Type: character vector

Values: `'A'` | `...`
Default: `'A'`

`Tag visibility` — Scope of the Goto block tag

`local` (default) | `scoped` | `global`

The scope of the Goto block tag, specified as `local`, `scoped`, or `global`. When you set this parameter to `scoped`, you must use a Goto Tag Visibility block to define the scope of tag visibility.

Programmatic Use
Block Parameter: `TagVisibility`
Type: character vector
Values: `'local' | 'scoped' | 'global'`
Default: `'local'`

`Icon display` — Text to display on block icon

`Tag` (default) | `Signal name` | `Tag and signal name`

Specifies the text to display on the block icon. The options are the block tag, the name of the signal that the block represents, or both the tag and the signal name.

Programmatic Use
Block Parameter: `IconDisplay`
Type: character vector
Values: `'Signal name' | 'Tag' | 'Tag and signal name'`
Default: `'Tag'`

`Rename All` — Propagate name throughout model

button

Rename the Goto tag. The new name propagates to all From and Goto Tag Visibility blocks that are listed in the **Corresponding blocks** box.

`Corresponding blocks` — Blocks connected to this Goto block

block path | ...

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.

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated | bus | string
Direct Feedthrough	No
Multidimensional Signals	Yes
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™.

For more information on HDL code generation, see Goto.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

From | Goto Tag Visibility

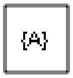
Topics

“Signal Label Propagation”

Introduced before R2006a

Goto Tag Visibility

Define scope of Goto block tag
Library: Simulink / Signal Routing



Description

The Goto Tag Visibility block defines the accessibility of Goto block tags that have `scoped` visibility. The value you specify for the **Goto tag** block 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 (`{}`).

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.

Parameters

`Goto tag` — Goto block tag whose visibility is defined by the location of this block

A (default) | ...

The Goto block tag whose visibility is defined by the location of this block. From and Goto blocks using the specified 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, in other words, the boundary of an atomic, conditionally executed, or function-call subsystem or a model reference. A scoped tag name is enclosed in braces ({}).

Tip 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.

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

From | Goto

Topics

“Signal Label Propagation”

Introduced before R2006a

Ground

Ground unconnected input port

Library: Simulink / Commonly Used Blocks
Simulink / Sources

Description

The Ground block connects to blocks whose input ports do not connect to other blocks. If you run a simulation with blocks that have unconnected input ports, Simulink issues warnings. Using a Ground block to ground those unconnected blocks can prevent these warnings.

Working with Fixed-Point Data Types

When working with fixed-point data types, there may be instances where the fixed-point data type cannot represent zero exactly. In these cases, 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. These 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

Working with Enumerated Data Types

When working with enumerated data types, the Ground block outputs the default value of the enumeration. This behavior applies whether:

- 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.

Ports

Output

`Port_1` — Ground signal
scalar

The Ground block outputs a scalar signal with zero value, and the same data type as the port to which it connects.

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

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
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™.

For more information on HDL code generation, see Ground.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

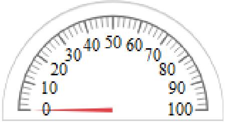
Topics

“Unconnected block input ports”

Introduced before R2006a

Half Gauge

Display input value on semicircular scale
Library: Simulink / Dashboard



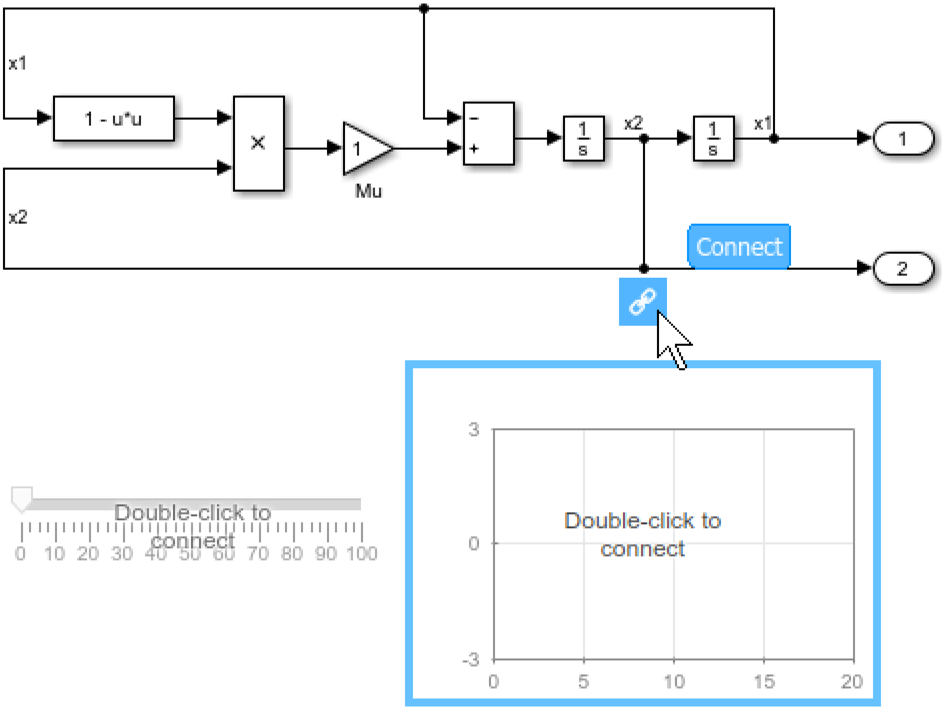
Description

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.

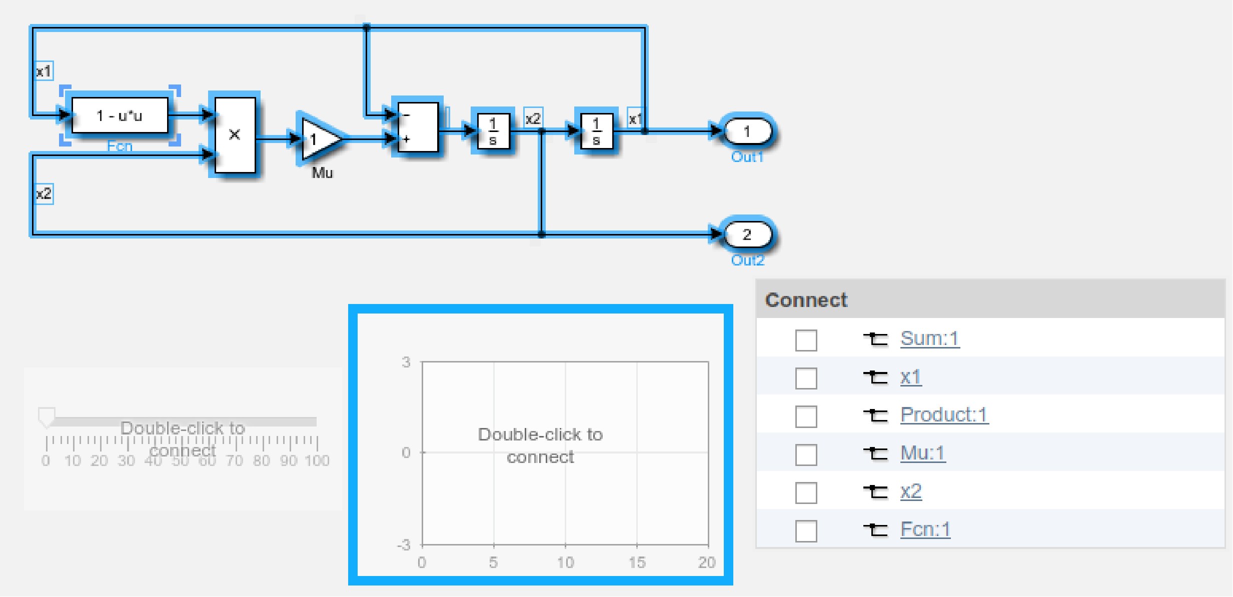
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect	
□	Sum:1
□	x1
□	Product:1
□	Mu:1
□	x2
□	Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced 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.

Parameters

`Connection` — Select a signal to connect and display

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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 arc. 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 arc. 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 arc. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

`Scale Colors` — Color indications on gauge arc

colors for arc ranges

Color specifications for ranges on the arc. 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.

See Also

Custom Gauge | Gauge | Linear Gauge | Quarter Gauge

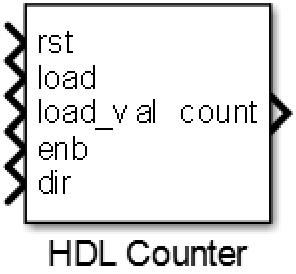
Topics

"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`

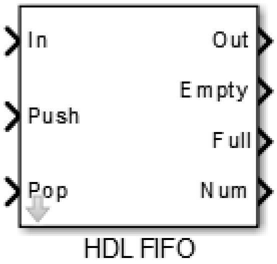
Counter value.

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

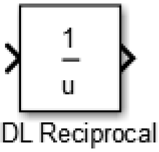
See Also

Dual Rate Dual Port RAM

Introduced in R2014a

HDL Reciprocal

Calculate reciprocal with Newton-Raphson approximation method



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:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + (x_i - a{x_i}^2) = x_i.(2 - ax_i)$$

$f'(x)$ 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
`fi` (*value*, 0, *word_length*, *fraction_length*)	`fi` (*value*, 0, *word_length*, *word_length*–*fraction_length*–1)
`fi` (*value*, 1, *word_length*, *fraction_length*)	`fi` (*value*, 1, *word_length*, *word_length*–*fraction_length*–2)

See Also

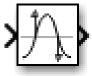
Divide | Math Function

Introduced in R2014b

Hit Crossing

Detect crossing point

Library: Simulink / 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.

You can configure the block to output a 1 or 0 `signal` or a SimEvents® `message`. See “Output” on page 1-916 for more information.

Ports

Input

`Port_1` — Input signal
scalar | vector

Input signal that the block detects when it reaches the offset in the specified direction.

Data Types: `double`

Output

`Port_1` — Output signal
scalar | vector | SimEvents message

Output indicating if the input signal crossed the hit offset. This port is only visible when you select the **Show output port** parameter check box.

Signal Output

If you select the **Show output port** check box and set the `Output type` parameter to `Signal`, 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 in the specified direction, the block continues to output a value of 1.
- If the input signals at two adjacent points brackets the offset value, 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 initial signal is equal to the offset value, the block outputs 1 only if the **Hit crossing direction** property is set to `either`.
- If Boolean logic signals are enabled, then the output is a `Boolean`.

SimEvents Message Output

The Hit Crossing block can also output a SimEvents message when the `Output Type` is set to `Message`.

- If the input signal crosses the offset value in the specified direction, the block outputs a message.
- If the input signal reaches the offset value in the specified direction and remains there, block outputs one message at the hit time and one message when the signal leaves the offset value.
- If the initial input signal is equal to the offset value, the block outputs a message with `Crossing Type` value `None` only if the **Hit crossing direction** is set to `either`.

The SimEvents message output signal is a `struct` with four fields.

`CrossingType` — Direction of zero-crossing

`None` | `NegativeToPositive` | `NegativeToZero` | `ZeroToPositive` | `PositiveToNegative` | `PositiveToZero` | `ZeroToNegative`

This field shows the direction in which the signal crosses the **Hit crossing offset** value. Negative, Zero, and Positive are defined relative to the offset value. The data type is `slHitCrossingType` which is an enumerated data type. See “Use Enumerated Data in Simulink Models” for more information. For example, if `HitCrossingOffset` is set to 2, a rising signal crossing this offset value would be recorded as a `NegativeToPositive` hit crossing.

Note A hit crossing is recorded based on the **Hit crossing direction** setting. In other words, if you set **Hit crossing direction** to detect a `falling` hit crossing, a `NegativeToPositive` hit is not recorded.

Note In a SimEvents block, if the `Crossing Type` of an entity is a `NegativeToPositive` hitcrossing then `entity.CrossingType == slHitCrossingType.NegativeToPositive` returns logical 1 `(true)`.

If the signal reaches the `HitCrossingOffset` value and holds it, a single `NegativeToZero` or `PositiveToZero`, depending on the direction, hit is registered at the time of the hit crossing.

Data Types: `slHitCrossingType`

Index — Index of the input signal at which the hit crossing event occurs
nonnegative integer

For *n* signals being passed to the Hit Crossing block, this field denotes which signal had a hit crossing event. For a matrix input, this field follows MATLAB linear indexing. See “Array Indexing” (MATLAB).

Data Types: `uint32`

Time — Time of hit crossing event
real, finite

Time *T* of the hit crossing event.

Data Types: `double`

`Offset` — Hit crossing value for detection
`0` (default) | real values

Hit crossing offset value as specified by the “Hit crossing offset” on page 1-0 parameter.

Data Types: `double`

Data Types: `double` | `Boolean` | `struct`

Note If the SimEvents message output signal crosses model reference boundaries or is used as an input to a Stateflow chart, you need to create a bus object for the message. See “Tips” on page 1-921.

Parameters

`Hit crossing offset` — Hit crossing value for detection

`0` (default) | real values

Specify the value the block detects when the input crosses in the direction specified by **Hit crossing direction**.

Programmatic Use
Block Parameter: `HitCrossingOffset`
Type: character vector
Values: real values
Default: `'0'`

`Hit crossing direction` — Input signal direction to hit crossing

`either` (default) | `falling` | `rising`

Direction from which the input signal approaches the hit crossing offset for a crossing to be detected.

When 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 **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.

Programmatic Use
Block Parameter: `HitCrossingDirection`
Type: character vector
Values: `'either'` | `'rising'` |`'falling'`
Default: `'either'`

`Show output port` — Display an output port

`off` (default) | `on`

If selected, create an output port on the block icon.

Programmatic Use
Block Parameter: `ShowOutputPort`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

`Output type` — Choose signal or message output

signal (default for Simulink) | message (default for SimEvents)

When **Output type** is set to `Signal`, the output signal is set to one whenever the input signal crosses the **Hit crossing offset** value in the **Hit crossing direction** and is zero at other times.

When the **Output type** is set to `Message`, the output signal becomes a SimEvents message.

Programmatic Use
Block Parameter: `HitCrossingOutputType`
Type: character vector
Values: `'Signal'` | `'Message'`
Default: `'Signal'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than -1

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Tips

If the Hit Crossing block is configured to output a SimEvents message and the output signal:

- Crosses into or out of a referenced model
- Is fed to the input of a Stateflow chart

then you need to create a bus object for the message signal. In the MATLAB Command Window, run `Simulink.createHitCrossMessage` to check for and, if needed, create a hit crossing message bus object in the base workspace.

Set the data type of the corresponding port to `Bus: HitCrossMessage`.

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

Does not support non-floating data type for ert targets.

HDL Code Generation

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

Not supported for SimEvents messages.

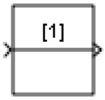
See Also

“Zero-Crossing Detection” | “Implement logic signals as Boolean data (vs. double)”

Introduced before R2006a

IC

Set initial value of signal
Library: Simulink / 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}). To do so, the block outputs 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.

The IC block is useful for providing an initial guess for the algebraic state variables in a loop. For more information, see “Algebraic Loops”.

Behavior for Nonzero Sample Time Offset

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}.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. The block sets the initial condition of this signal to the **Initial value** you specify.

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

Output

`Port_1` — Output signal

scalar | vector | matrix | N-D array

Output signal provided as the **Initial value** you specify, followed by the actual values of the input signal. See “Behavior for Nonzero Sample Time Offset” on page 1-923 for more information.

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

Parameters

`Initial value` — Initial value

`1` (default) | real, finite scalar, vector, matrix, or N-D array

Specify the initial value of the input signal as a finite, real-valued scalar, vector, matrix, or N-D array. The value must be a scalar, or have the same dimensions as the input signal.

Programmatic Use
Block Parameter: `Value`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'1'`

Sample time — Specify sample time as a value other than -1

-1 (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '-1'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

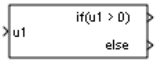
Topics

“What Is Sample Time?”
“Algebraic Loops”
“Classic Initialization Mode”

Introduced before R2006a

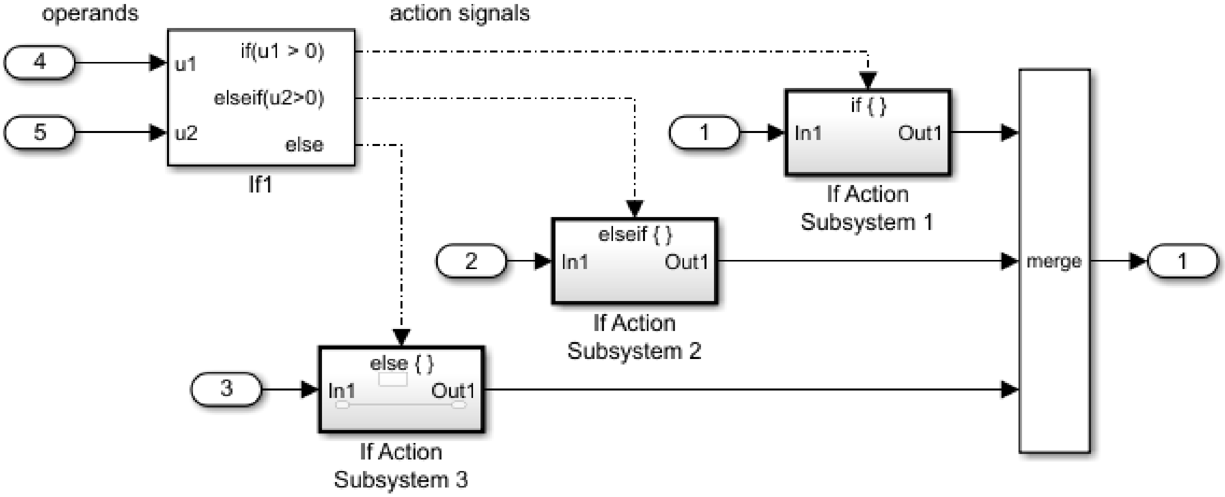
If

Select subsystem execution using logic similar to if-else statement
Library: Simulink / Ports & Subsystems



Description

The If block, along with If Action Subsystem blocks containing an Action Port block, implements if-else logic to control subsystem execution. For an example using the If block, see If Action Subsystems.



Limitations

The If block has the following limitations:

- It does not support tunable parameters. Values for an `if` or `elseif` expression cannot be tuned during a simulation in normal or accelerator mode, or when running generated code.

  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 of the If block or add the tunable parameter as an input to the If block.
- It does not support custom storage classes. See “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).
- The **If expression** and **Elseif expressions** cannot accept certain operators, such as `+, -, *`, and `/`.

Ports

Input

`Logical operands` — Values for evaluating logical expressions

scalar | vector

Inputs `u1,u2,...,un` must have the same data type. The inputs cannot be of any user-defined type, such as an enumerated type.

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. See Support for Fixed-Point Data Type in If Action Subsystems.

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

Output

`Action` — Action signal for an If Action Subsystem block

scalar

Outputs from the `if`, `else`, and `elseif` ports are action signals to If Action Subsystem blocks.

Support for Fixed-Point Data Type

Parameters

`Number of inputs` — Specify number of input ports

1 (default) | integer

1

Specify one input port.

integer

Specify the number of input ports. Block ports are labeled with a `'u'` character followed by a number, `1,2,...,n`, where `n` equals the number of inputs that you specify.

Programmatic Use
Block Parameter: `NumInputs`
Type: character vector
Values: `'1'` | `'<integer>'`
Default: `'1'`

`if expression` — Specify logical expression

`u1 > 0` (default) | logical expression

The If Action Subsystem attached to the `if` port executes when its associated expression evaluates to true.

`u1 > 0`

Specify sending an action signal on the output port when the input `u1` is greater than `0`.

logical expression

Specify logical expression. This expression appears on the If block adjacent to the **if** output port.

The expression can include only the operators `<, <=, ==, ~=, >, >=, &, |, ~, (), unary-minus`. Operators such as `+, -, *, /`, and `^` are not allowed. 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`.

Programmatic Use
Block Parameter: `IfExpression`
Type: character vector
Values: `'u1 > 0'` | `'<logical expression>'`
Default: `'u1 > 0'`

`Elseif expressions` — Specify logical expression

empty (default) | list of logical expressions

The If Action Subsystem attached to an **elseif** port executes when its expression evaluates to `true` and all `if` and `elseif` expressions are false.

empty

Logical expressions not specified.

list of logical expressions

Specify a list of logical expressions delimited by commas. The expressions appear on the If block below the **if** port and above the **else** port when you select the **Show else condition** check box.

Expressions can include only the operators `<, <=, ==, ~=, >, >=, &, |, ~, (), unary-minus`. Operators such as `+, -, *, /`, and `^` are not allowed. The expressions 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`.

Programmatic Use
Block Parameter: `ElseIfExpressions`
Type: character vector
Values: `''` | `'<list of logical expressions>'`
Default: `''`

`Show else condition` — Control display of else port

on (default) | off

When the **if** port and all **elseif** port expressions are `false`, the **else** port sends an action signal to execute the attached If Action Subsystem block.

☑ on

Display **else** port.

□ off

Hide **else** port.

Programmatic Use
Block Parameter: `ShowElse`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`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
Values: `'on'` | `'off'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks

Action Port | If Action Subsystem | Subsystem

Topics

Select Subsystem Execution

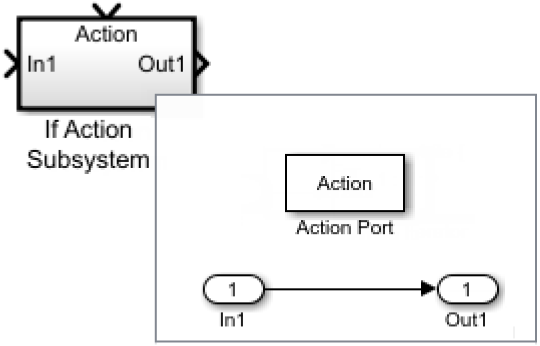
Introduced before R2006a

If Action Subsystem

Subsystem whose execution is enabled by an If block
Library: Simulink / Ports & Subsystems

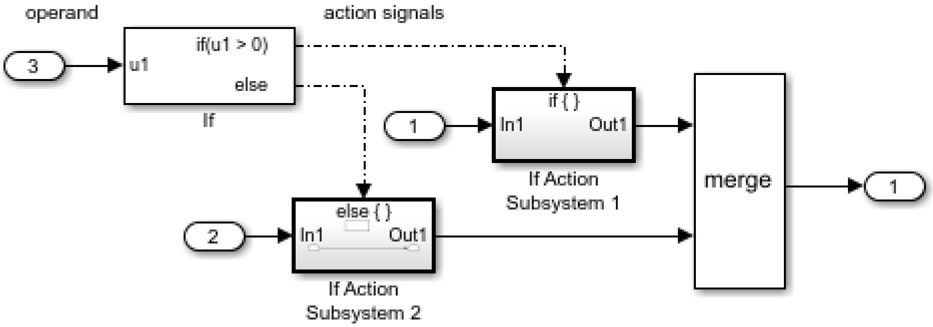
Description

The If Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is controlled by an If block. The If block evaluates a logical expression and then, depending on the result of the evaluation, outputs an action signal.



Merge signals from If Action Subsystem blocks

This example shows how to merge signals controlled by an If block. The If block selects the execution of an If Action Subsystem block from a set of subsystems. Regardless of which subsystem the If block selects, you can create a single signal with a Merge block.
Open model

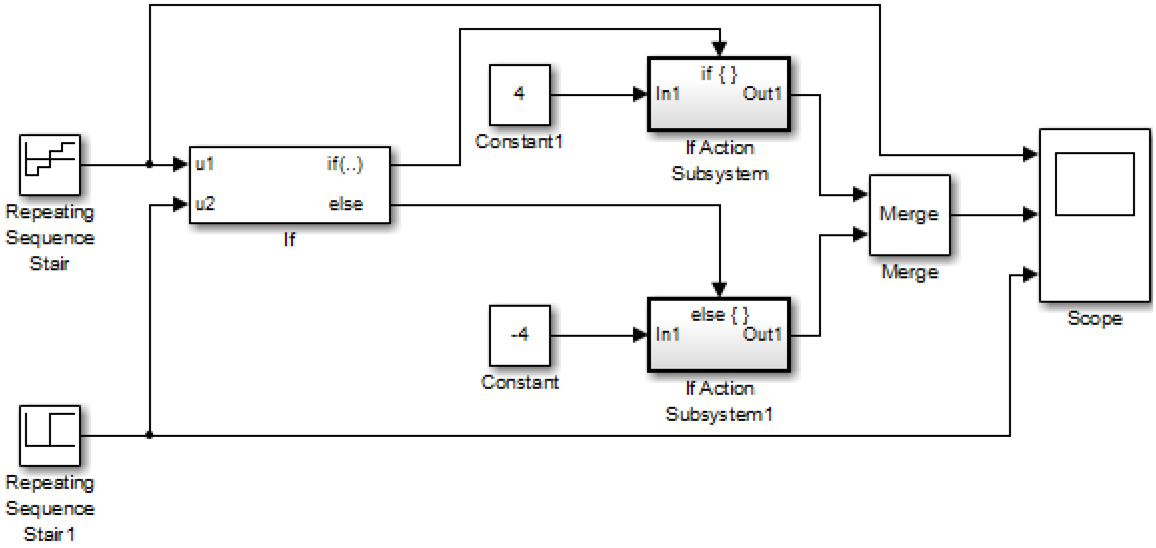


All blocks in an If Action Subsystem block must execute at the same rate as the driving If block. You can satisfy this requirement by setting the sample time parameter for each block to either inherited (-1) or the same value as the If block sample time.

Support for Fixed-Point Data Type

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.

Consider the following floating-point model without fixed-point data types:



In this model, the If Action Subsystem blocks use their default configurations. The simulation parameters are set to their default values except for the parameters listed in the following table.

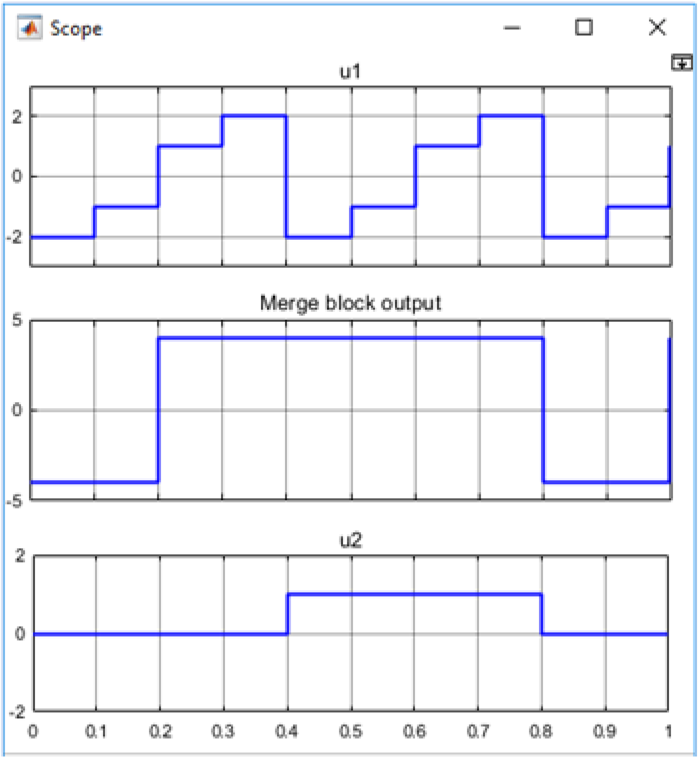
Configuration Parameter Pane	Parameter	Setting
Solver	**Start time**	`0.0`
	Stop time	`1.0`
	Type	`Fixed-step`
	Solver	`discrete (no continuous states)`
	Fixed-step size	`0.1`

The block parameters are set to their default values except for the parameters listed in the following table.

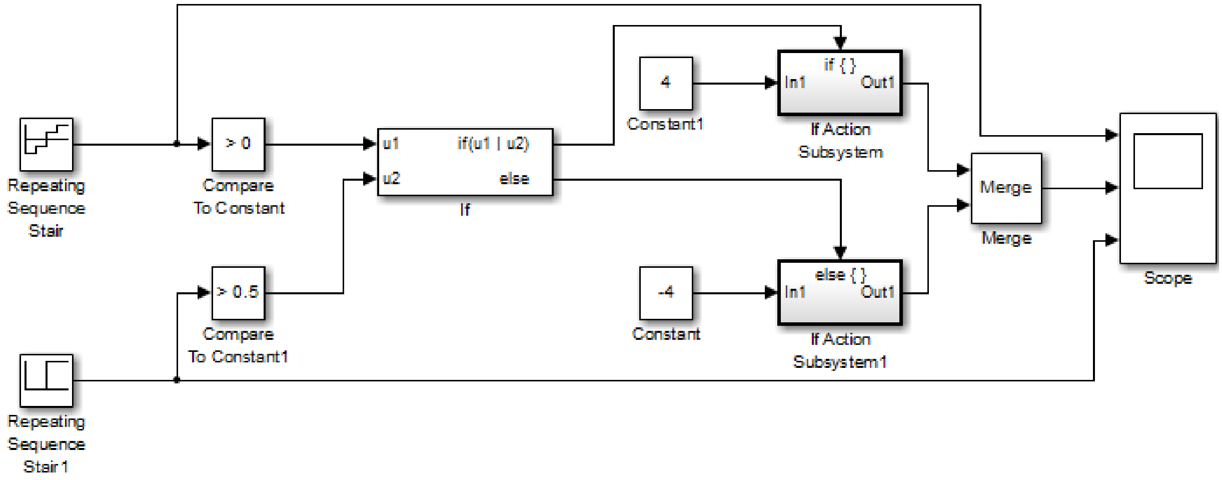
Block	Parameter	Setting
Repeating Sequence Stair	**Vector of output values**	`[-2 -1 1 2].'`

Block	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) | (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, when 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 displays the output from the Merge block with inputs `u1`, and `u2`.



You can implement this block diagram as a model with fixed-point data types:



The Repeating Sequence Stair blocks 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 in this model.

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`

Block	Parameter	Setting
	Enable zero-crossing detection	off
If	**Number of inputs**	2
	If expression	`u1|u2`

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 an If Action Subsystem.

Dot-dash lines from a Switch Case block to an Switch Case Action Subsystem block represent *action* signals. An action signal is a control signal connected to the action port of a Switch Case Action Subsystem block. A message on the action signal initiates execution of the 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`

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[a] | `string`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`Yes`[a]
Zero-Crossing Detection	`No`

[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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Blocks
Action Port | If | Subsystem

Topics

Select Subsystem Execution

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

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.

To reduce the number of bus element signals displayed in the Block Parameters dialog box, use the **Filter** box. The **Filter** box supports regular expressions. To use a regular expression character as a literal, include an escape character (\). For example, to use a question mark: `sig\?1`.

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`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Blocks

Bus Selector | Inport | Out Bus Element

Topics

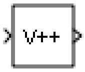
“Simplify Subsystem Bus Interfaces”
“Composite Signal Techniques”
“Select a Composite Signal Technique”
“Getting Started with Buses”

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` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output is the real-world value of the input signal increased by one. Overflows always wrap. The output has the same data type and dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Block Characteristics

Data Types	double | single | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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™.

For more information about HDL code generation, see Increment Real World.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Decrement Stored Integer | Decrement Time To Zero | Decrement To Zero | Increment Real World

Topics

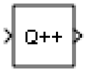
“Fixed-Point Numbers”

Introduced before R2006a

Increment Stored Integer

Increase stored integer value of signal by one

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output is the stored integer value of the input signal increased by one. Floating-point signals also increase by one, and overflows always wrap. The output has the same data type and dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Block Characteristics

Data Types	double | single | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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™.

For more information about HDL code generation, see Increment Stored Integer.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Decrement Stored Integer | Increment Real World

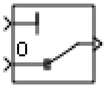
Topics

"Fixed-Point Numbers"

Introduced before R2006a

Index Vector

Switch output between different inputs based on value of first input
Library: Simulink / Signal Routing



Description

The Index Vector block is a special configuration of the 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.

To configure a Multiport Switch block to work as an Index Vector block set **Number of data ports** to `1` and **Data port order** to `Zero-based contiguous`.

For more information about the Multiport Switch block, see the Multiport Switch block reference page.

Ports

Input

`Port_1` — Control signal
scalar

Control signal, specified as a scalar. The control signal can be of any data type that Simulink supports, including fixed-point and enumerated types. When the control input is not an integer value, the block truncates the value to an integer by rounding to zero.

For information on control signals of enumerated type, see “Guidelines on Setting Parameters for Enumerated Control Port” on page 1-1288 on the Multiport Switch block ref page.

Limitations

- If the control signal is numeric, the control signal 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 input, an error occurs.

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

`0 or 1` — First data input

scalar | vector

First data input, specified as a scalar or vector. The port is labeled **0** when you set **Data port order** to `Zero-based contiguous`, and labeled **1** when you set **Data port order** to `One-based contiguous`. The input data signal can be of any data type that Simulink supports.

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

Output

`Port_1` — Selected data input, based on control signal value

scalar | vector | matrix | N-D array

The block outputs the selected value from the input data vector, according to the control signal value. The output is a scalar.

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

Parameters

Main

Data port order — Type of ordering for data input ports

Zero-based contiguous (default) | One-based contiguous | Specify indices

Specify the type of ordering for your data input ports.

- Zero-based contiguous — Block uses zero-based indexing for ordering contiguous data ports. This is the default value of the Index Vector block.
- One-based contiguous — Block uses one-based indexing for ordering contiguous data ports. This is the default value of the Multiport Switch block.
- Specify indices — Block uses noncontiguous indexing for ordering data ports. This value is supported only for configurations with two or more input data ports.

Tips

- 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:	The block produces:
Zero-based contiguous or One-based contiguous	A warning
Specify indices	An error

Dependencies

Selecting Zero-based contiguous or One-based contiguous enables the **Number of data ports** parameter.

Selecting `Specify indices` enables the **Data port indices** parameter.

Programmatic Use
Block Parameter: `DataPortOrder`
Type: character vector
Values: `'Zero-based contiguous' | 'One-based contiguous' | 'Specify indices'`
Default: `'Zero-based contiguous'`

`Number of data ports` — Number of data input ports

`1` (default) | integer between 1 and 65536

Specify the number of data input ports to the block.

Dependencies

To enable this parameter, set **Data port order** to `Zero-based contiguous` or `One-based contiguous`.

Programmatic Use
Block Parameter: `Inputs`
Type: character vector
Values: integer between 1 and 65536
Default: `'1'`

Signal Attributes

`Require all data port inputs to have the same data type` — Require all inputs to have the same data type

`off` (default) | `on`

Select this check box to require that all data input ports have the same data type. When you clear this check box, the block allows data port inputs to have different data types.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

When you select an inherited option, the block behaves as follows:

- `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.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule`|`'Inherit: Inherit via back propagation'`|`'double'`|`'single'`|`'int8'`|`'uint8'`|`'int16'`|`'uint16'`, `'int32'`|`'uint32'`|`'fixdt(1,16)'`|`'fixdt(1,16,0)'`|`'fixdt(1,16,2^0,0)'`|`'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off'`|`'on'`
Default: `'off'`

`Integer rounding mode` — Specify the rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Choose one of these rounding modes.

`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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

See Also

For more information, see "Rounding" (Fixed-Point Designer).

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Allow different data input sizes (Results in variable-size output signal)` — Allow input signals with different sizes
`off` (default) | `on`

Select this check box to allow input signals with different sizes.

- `On` — Allows input signals with different sizes, and propagate the input signal size to the output signal. In this mode, the block produces a variable-size output signal.
- `Off` — Requires that all nonscalar data input signals be the same size.

Programmatic Use
Parameter: `AllowDiffInputSizes`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`

Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Index Vector.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Multiport Switch | Switch

Topics

“Variable-Size Signal Basics”

Introduced before R2006a

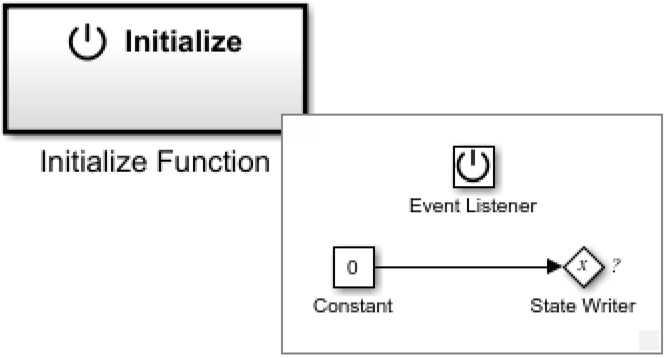
Initialize Function

Executes contents on a model initialize event

Library: Simulink / 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.

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[a] | `string`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[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 code generation support depends on block implementation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Event Listener | Reset Function | State Reader | State Writer | Terminate Function

Topics

"Customize Initialize, Reset, and Terminate Functions"

“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: Simulink / Commonly Used Blocks
Simulink / Ports & Subsystems
Simulink / Sources

Description

Inport blocks link signals 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 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.

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 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, use the `createInputDataset` function to create a `Dataset` object that contains elements corresponding to root-level Inport blocks.

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. The block continues to receive its signal from the same block outside the subsystem.

Inport blocks inside a subsystem support signal label propagation, but root-level Inport blocks do not.

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. 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.

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.

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, 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”.

Ports

Output

`Port_1` — Inport signal

scalar | vector

Input signal that flows through the inport into the system.

You can use a subsystem inport to supply fixed-point data in a structure or any other format.

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

Parameters

Main

`Port number` — Port number of block

`1` (default)

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

Programmatic Use
Block Parameter: `Port`
Type: character vector
Values: real integer
Default: `'1'`

`Icon display` — Icon display

`Port number` (default) | `Signal name` | `Port number and signal name`

Specify the information to be displayed on the icon of this port.

Programmatic Use
Block Parameter: `IconDisplay`
Type: character vector
Values: `'Signal name'` | `'Port number'` | `'Port number and signal name'`
Default: `'Port number'`

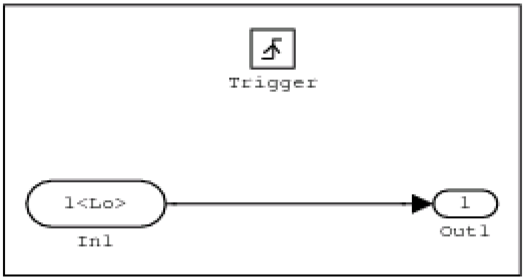
`Latch input by delaying outside signal` — Latch signal by delay

`off` (default) | `on`

Select to specify the block outputs the value of the input signal at the previous time step.

Selecting this check box enables Simulink to resolve data dependencies among triggered subsystems that are part of a loop.

The Inport block indicates that this option is selected by displaying `<Lo>`.



Dependency

Enabled in a triggered subsystem.

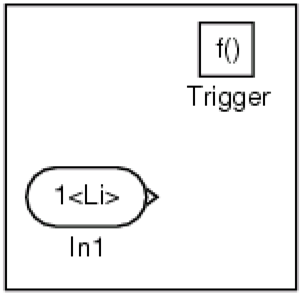
Programmatic Use
Block Parameter: `LatchByDelaying OutsideSignal`
Type: character vector
Values: `'on' | 'off'`
Default: `'off'`

`Latch input for feedback signals of function-call subsystem outputs` — Latch signal from changing

`off` (default) | `on`

Select to specify the block latches the value of the input to this subsystem and prevents 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 breaks a loop formed by a signal fed back from one of these function-call subsystems into the other. This option prevents any change to the values of a feedback signal from a function-call subsystem that is invoked during the execution of this subsystem.

The Inport block indicates that this option is selected by displaying `<Li>`.



Dependency

Enabled when Inport block is in a function-call subsystem.

Programmatic Use
Block Parameter: `LatchInputFor FeedbackSignals`
Type: character vector
Values: `'on' | 'off'`
Default: `'off'`

`Interpolate data` — Interpolate output data

on (default) | ogg

When loading data from the workspace to a root-level Inport block, specify whether the block linearly interpolates and extrapolates 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 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”.

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-size 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.

Programmatic Use
Block Parameter: `Interpolate`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

`Connect Input` — Tool to help map signals to inports

button

To import, visualize, and map signal and bus data to root-level inports, click this button. The Root Inport Mapper tool displays.

Dependency

This button appears only if this block is a root inport block.

Signal Attributes

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Output function call` — Output function-call trigger signal

`off` (default) | `on`

Specify that the input signal outputs 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.

Dependency

Enabled in an asynchronous function call.

`Minimum` — Minimum output value

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Maximum` — Maximum output value

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Data type` — Output data type

`Inherit: auto` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class name>` | `Bus: <object name>` | `<data type expression>`

Specify the output data type of the external input. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. Do not specify a bus object as the expression.

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Output as nonvirtual bus in parent model` — Outport as nonvirtual bus in parent model

`off` (default) | `on`

Specify the outport bus to be nonvirtual in the 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**.

Clear this parameter if you want the bus emerging in the parent model to be virtual.

Tips

- 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 Outport 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.

Dependency

To enable this parameter, select **Data type** > `Bus: <object name>`.

Programmatic Use

Block Parameter: `BusOutputAsStruct`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Unit (e.g., m, m/s^2, N*m)` — Physical unit of the input signal to the block

`inherit` (default) | `<Enter unit>`

Specify the physical unit of the input signal to the block. 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.

Programmatic Use
Block Parameter: `Unit`
Type: character vector
Values: `'inherit'` | `'<Enter unit>'`
Default: `'inherit'`

`Port dimensions (-1 for inherited)` — Port dimensions

`-1` (default) | integer | [integer, integer]

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

`-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.

Programmatic Use
Block Parameter: `PortDimensions`
Type: character vector
Values: `'-1'` | integer | [integer, integer]
Default: `'-1'`

`Variable-size signal` — Allow variable-size signals

`Inherit` (default) | `No` | `Yes`

Specify the type of signals allowed out of this port. To allow variable-size and fixed-size signals, select `Inherit`. To allow only variable-size signals, select `Yes`. Not to allow variable-size signals, select `No`.

Dependencies

When the signal at this port is a variable-size signal, the **Port dimensions** parameter specifies the maximum dimensions of the signal.

Command-Line Information
Parameter: `VarSizeSig`
Type: character vector

Value: `'Inherit'` | `'No'` | `'Yes'`
Default: `'Inherit'`

`Sample time (-1 for inherited)` — Specify sample time

`-1` (default) | scalar

Specify the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

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”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

`Signal type` — Output signal type

`auto` (default) | `real` | `complex`

Specify the numeric type of the signal output. To choose the numeric type of the signal that is connected to its input, select `auto`. Otherwise, choose a real or complex signal type.

Programmatic Use
Block Parameter: `SignalType`
Type: character vector
Values: `'auto'` | `'real'` | `'complex'`
Default: `'auto'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`

Multidimensional Signals	`Yes`
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™.

For information about HDL Code Generation, see Inport.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Asynchronous Task Specification | Outport

Topics

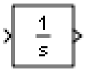
"Load Data to Root-Level Input Ports"
"Using Function-Call Subsystems"

“Map Data Using Root Inport Mapper Tool”
“Load Big Data for Simulations”
“Comparison of Signal Loading Techniques”
“Map Root Inport Signal Data”

Introduced before R2006a

Integrator

Integrate signal
Library: Simulink / Commonly Used Blocks
Simulink / Continuous



Description

The Integrator block outputs the value of the integral of its input signal with respect to time.

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 several different numerical integration methods to compute the output of the block, 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:

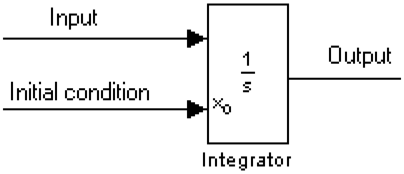
- Define upper and lower limits on the integral
- Create an input that resets the block's output (state) to its initial value, depending on how the input changes
- Create an optional state output so that the value of the block's output can trigger a block reset

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.



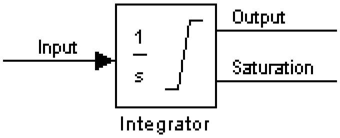
Note If the integrator limits its output (see “Limiting the Integral” on page 1-982), the initial condition must fall inside the integrator's saturation limits. If the initial condition is outside the block saturation limits, the block displays an error message.

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

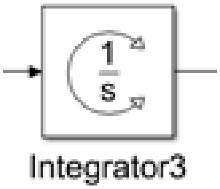
Several physical phenomena are cyclic, periodic, or rotary in nature. Objects or machinery that exhibit rotational movement and oscillators are examples of such phenomena.

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 –π 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:

$$y = \begin{cases} x & x \in [x_l, x_u) \\ x - (x_u - x_l)\left\lfloor \dfrac{x - x_l}{x_u - x_l} \right\rfloor & \text{otherwise} \end{cases}$$

where:

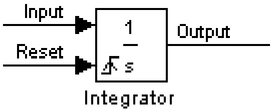
- x_l is the lower value of the wrapped state.
- x_u is the upper value of the wrapped state.
- y is the output.

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.



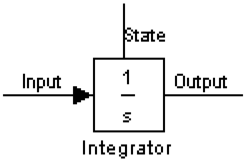
- Select `rising` to reset the state when the reset signal rises from a negative or zero value to a positive value.
- Select `falling` to reset the state when the reset signal falls from a positive value to a zero or negative value.
- Select `either` to reset the state when the reset signal changes from 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.



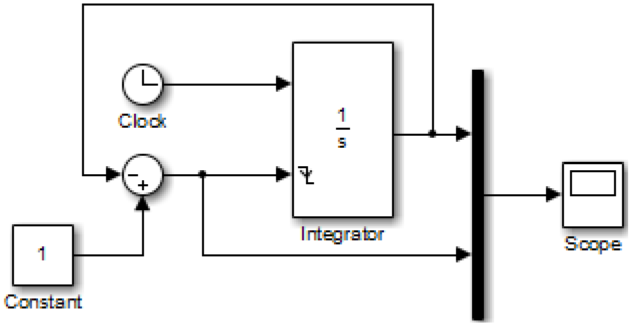
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. 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 1-986)
- Handing off a state from one enabled subsystem to another (see “Handing Off States Between Enabled Subsystems” on page 1-987)

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.

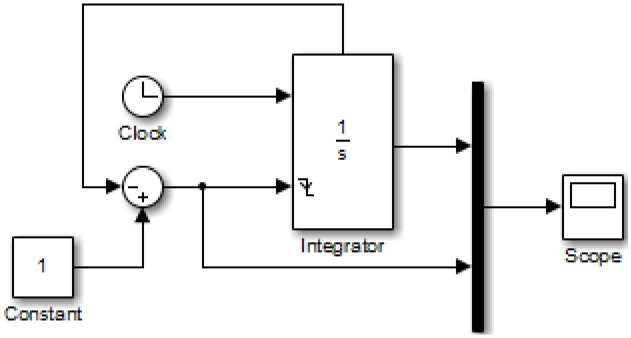
Creating Self-Resetting Integrators

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.

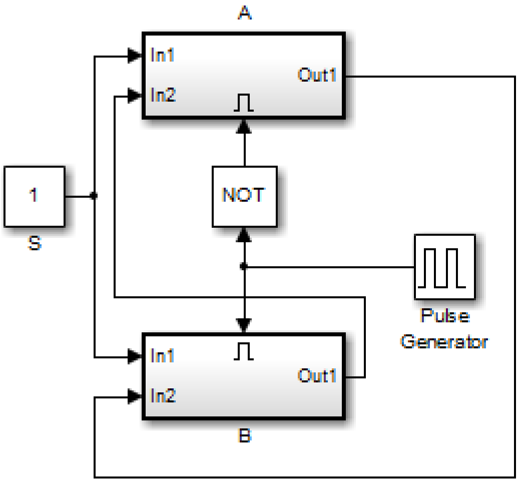
The following model uses the integrator's state port to avoid the algebraic loop.



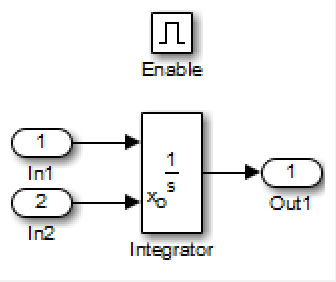
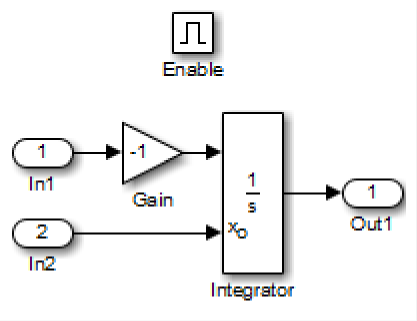
In this version, the value of the reset signal depends on the value of the state port. The value of the state port is available earlier in the current time step than the value of the integrator block's output port. Therefore, Simulink can determine whether the block needs to be reset before computing 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.



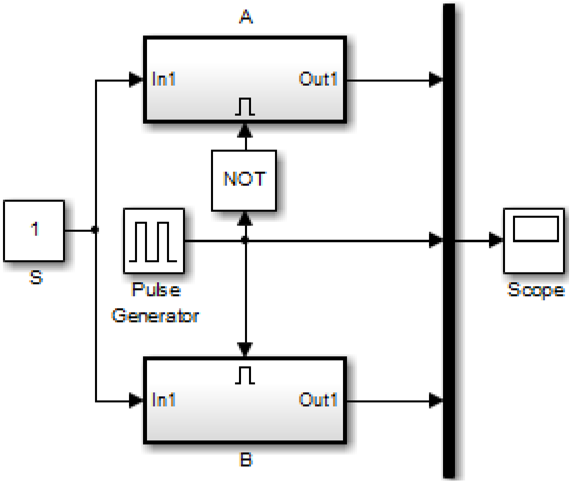
The enabled subsystems, A and B, contain the following blocks:

Subsystem A	Subsystem B
 Enable 1 In1 2 In2 1/s x0 Integrator 1 Out1	 Enable 1 In1 -1 Gain 2 In2 1/s x0 Integrator 1 Out1

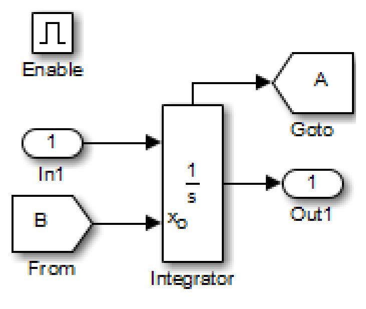
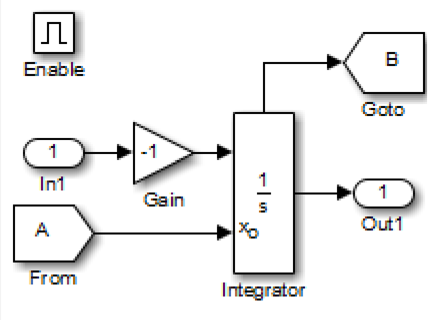
In this model, a constant input signal drives two enabled subsystems that integrate the signal. A pulse generator generates an enabling signal that causes execution to alternate between the two subsystems. The enable port of each subsystem is set to reset, which causes the subsystem to reset its integrator 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:

Subsystem A	Subsystem B
 Enable 1 In1 B From 1/s xo Integrator A Goto 1 Out1	 Enable 1 In1 -1 Gain A From 1/s xo Integrator B Goto 1 Out1

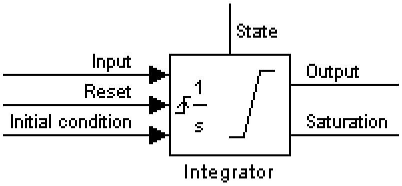
In this model, the initial condition of the integrator in A depends on the value of the state port of the integrator in B, and vice versa. The values of the state ports are updated earlier in the simulation time step than the values of the integrator output ports. Therefore, Simulink can compute the 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.



Ports

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`.

Input

`Port_1` — Integrand signal

real scalar or array

Signal that needs to be integrated.

Data Types: `double`

`External Reset` — Reset state to initial conditions

real scalar or array

Reset the state to the specified initial conditions based on an external signal. See "Resetting the State" on page 1-984.

Dependencies

To enable this port, enable the **External Reset** parameter.

Data Types: `Boolean`

x_0 — Initial condition

real scalar or array

Set the initial condition of the block's state from an external signal.

Dependencies

To enable this port, set the **Initial Conditions** parameter to `external`.

Data Types: `double`

Output

`Port_1` — Output signal

real scalar or array

Output the integrated state.

Data Types: `double`

`Port_2` — Show output saturation

-1 | 0 | 1

Indicate when the state is being limited. The signal has a value of `1` when the integral is limited by the specified **Upper saturation limit**. When the signal is limited by the **Lower saturation limit**, the signal value is `-1`. When the integral is between the saturation limits, the signal value is `0`. See “Limiting the Integral” on page 1-982.

Data Types: `double`

`Port_3` — State

real scalar or array

Output the state of the block. See “About the State Port” on page 1-985.

Dependencies

Enable this port by enabling the **Show state port** parameter.

Data Types: `double`

Parameters

`External reset` — Reset states to their initial conditions

`none` (default) | `rising` | `falling` | `either` | `level` | `level hold`

Specify the type of trigger to use for the external reset signal.

- 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.

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector , string
Values: `'none'` | `'rising'` | `'falling'` | `'either'` | `'level'` | `'level hold'`
Default: `'none'`

`Initial condition source` — Select source of initial condition

`internal` (default) | `external`

Select source of initial condition:

- `internal` — Get the initial conditions of the states from the **Initial condition** block parameter.
- `external` — Get the initial conditions of the states from an external block, via the **IC** input port.

Dependencies

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter and enables the **IC** input port.

Programmatic Use
Block Parameter: `InitialConditionSource`
Type: character vector, string
Values: `'internal'` | `'external'`
Default: `'internal'`

`Initial condition` — Initial state

0 (default) | real scalar or array

Set the initial state of the Integrator block.

Tips

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

Setting **Initial condition source** to `internal` enables this parameter.

Setting **Initial condition source** to `external` disables this parameter.

Programmatic Use

Block Parameter: `InitialCondition`
Type: scalar or vector
Default: `'0'`

`Limit output` — Limit block output values to specified range

`off` (default for Integrator) | `on` (default for Integrator Limited)

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

- Selecting this check box limits the block output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.
- Clearing this check box does not limit the block output values.

Dependencies

Selecting this parameter enables the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use

Block Parameter: `LimitOutput`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Upper saturation limit` — Upper limit for the integral

`inf` (default) | scalar | vector | matrix

Specify the upper limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

Dependencies

To enable this parameter, select the **Limit output** check box.

Programmatic Use

Block Parameter: `UpperSaturationLimit`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'inf'`

`Lower saturation limit` — Lower limit for the integral

`-inf` (default) | scalar | vector | matrix

Specify the lower limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

Dependencies

To enable this parameter, select the **Limit output** check box.

Programmatic Use

Block Parameter: `LowerSaturationLimit`
Type: character vector , string
Values: scalar | vector | matrix
Default: `'-inf'`

`Wrap state` — Enable wrapping of states

`off` (default) | `on`

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.

If you specify **Wrapped state upper value** as `inf` and **Wrapped state lower value** as `-inf`, wrapping does not occur.

Dependencies

Selecting this parameter enables **Wrapped state upper value** and **Wrapped state lower value** parameters.

Programmatic Use

Block Parameter: `WrapState`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Wrapped state upper value` — Specify upper value for wrapped state

`pi` (default) | scalar or vector

Upper limit of the block output.

Dependencies

Selecting **Wrap state** enables this parameter.

Programmatic Use
Block Parameter: `WrappedStateUpperValue`
Type: scalar or vector
Values: `'2*pi'`
Default: `'pi'`

`Wrapped state lower value` — Specify lower value for wrap state

`-pi` (default) | scalar or vector

Lower limit of the block output.

Dependencies

Selecting **Wrap state** enables this parameter.

Programmatic Use
Block Parameter: `WrappedStateLowerValue`
Type: scalar or vector
Values: `'0'`
Default: `'-pi'`

`Show saturation port` — Enable saturation output port

`off` (default) | `on`

Select this check box to add a saturation output port to the block. When you clear this check box, the block does not have a saturation output port.

Dependencies

Selecting this parameter enables a saturation output port.

Programmatic Use
Block Parameter: `ShowSaturationPort`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Show state port` — Enable state output port

`off` (default) | `on`

Select this check box to add a state output port to the block. When you clear this check box, the block does not have a state output port.

Dependencies

Selecting this parameter enables a state output port.

Programmatic Use

Block Parameter: `ShowStatePort`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Absolute tolerance` — Absolute tolerance for block states

`auto` (default) | `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.

Programmatic Use

Block Parameter: `AbsoluteTolerance`
Type: character vector, string, scalar, or vector
Values: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

`Ignore limit and reset when linearizing` — Treat block as unresettable and output unlimited

`off` (default) | `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 reset and output limitation options of the block.

Tip

Use this check box to linearize a model around an operating point that causes the integrator to reset or saturate.

Programmatic Use

Block Parameter: `IgnoreLimit`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use

Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`State Name (e.g., 'position')` — Assign unique name to each state

`' '` (default) | character vector | string

- 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, string, cell array, or structure.

Programmatic Use

Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined

Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

- Consider discretizing the model
- Not recommended for production code

See Also

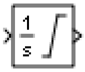
Discrete-Time Integrator | Second-Order Integrator

Introduced before R2006a

Integrator Limited

Integrate signal

Library: Simulink / Continuous



Description

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 1-1002 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 several different numerical integration methods to compute the output of the block, 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:

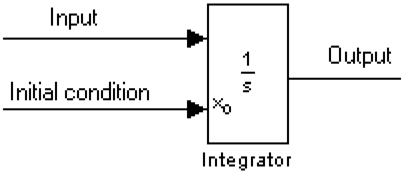
- Define upper and lower limits on the integral
- Create an input that resets the block's output (state) to its initial value, depending on how the input changes
- Create an optional state output so that the value of the block's output can trigger a block reset

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 1-1002), 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.

Wrapping Cyclic States

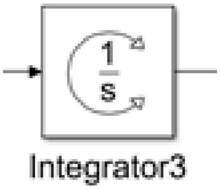
Several physical phenomena are cyclic, periodic, or rotary in nature. Objects or machinery that exhibit rotational movement and oscillators are examples of such phenomena.

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 –π 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:

$$y = \begin{cases} x & x \in [x_l, x_u) \\ x - (x_u - x_l)\left\lfloor \dfrac{x - x_l}{x_u - x_l} \right\rfloor & \text{otherwise} \end{cases}$$

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.

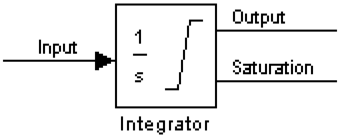
- It eliminates simulation instability when your model approaches large angles and large state values.
- It reduces the number of solver resets during simulation and eliminates the need for zero-crossing detection, improving simulation time.
- It eliminates large angle values, speeding up computation of trigonometric functions on angular states.
- It improves solver accuracy and performance and enables unlimited simulation time.

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:

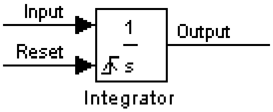
- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- –1 indicates that the lower limit is being applied.

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`.

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.



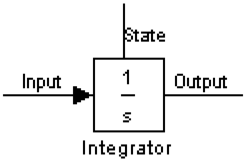
- Select `rising` to reset the state when the reset signal rises from a negative or zero value to a positive value.
- Select `falling` to reset the state when the reset signal falls from a positive value to a zero or negative value.
- Select `either` to reset the state when the reset signal changes from 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.



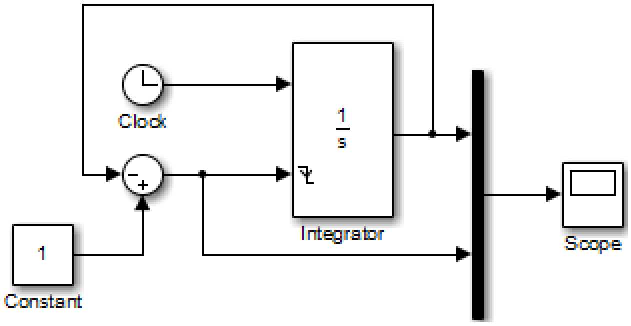
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. 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 1-1005)
- Handing off a state from one enabled subsystem to another (see “Handing Off States Between Enabled Subsystems” on page 1-1006)

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.

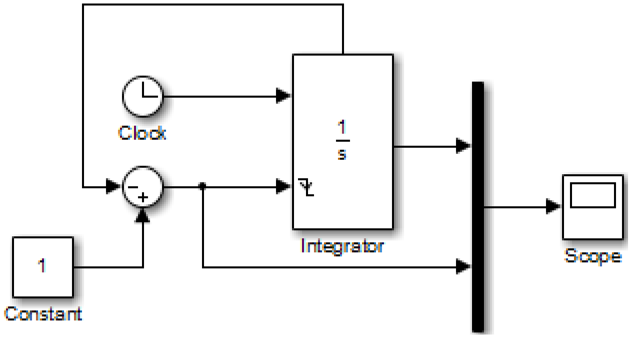
Creating Self-Resetting Integrators

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.

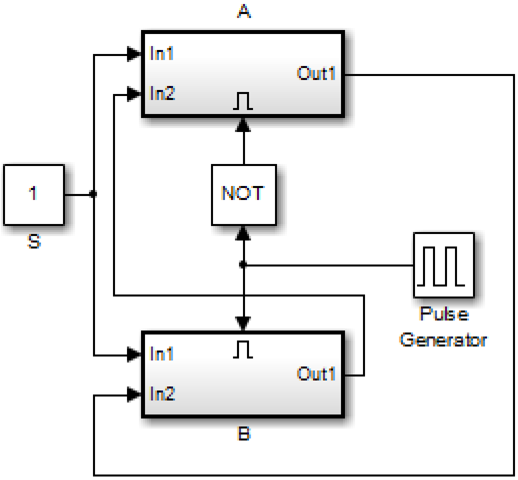
The following model uses the integrator's state port to avoid the algebraic loop.



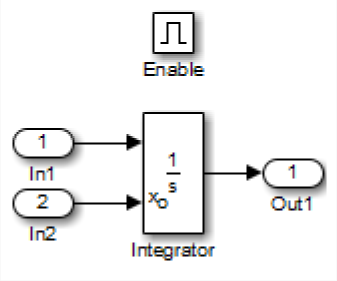
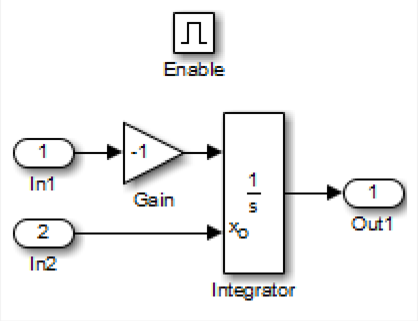
In this version, the value of the reset signal depends on the value of the state port. The value of the state port is available earlier in the current time step than the value of the integrator block's output port. Therefore, Simulink can determine whether the block needs to be reset before computing 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.



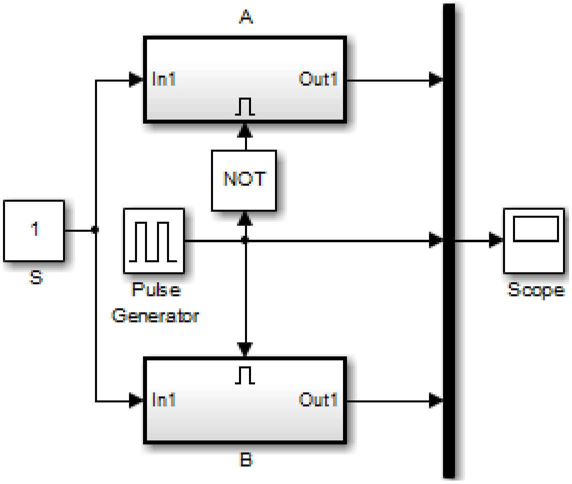
The enabled subsystems, A and B, contain the following blocks:

Subsystem A	Subsystem B
 Enable 1 In1 2 In2 1/s x0 Integrator 1 Out1	 Enable 1 In1 -1 Gain 2 In2 1/s x0 Integrator 1 Out1

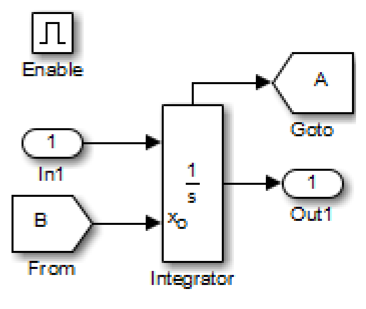
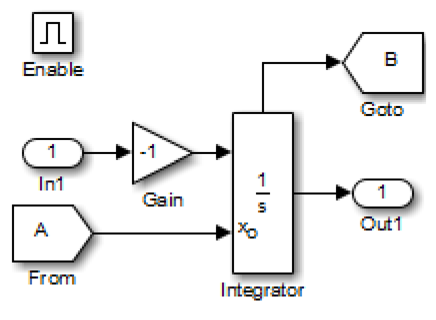
In this model, a constant input signal drives two enabled subsystems that integrate the signal. A pulse generator generates an enabling signal that causes execution to alternate between the two subsystems. The enable port of each subsystem is set to reset, which causes the subsystem to reset its integrator 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:

Subsystem A	Subsystem B
 Enable 1 In1 B From 1 s x0 Integrator A Goto 1 Out1	 Enable 1 In1 -1 Gain A From 1 s x0 Integrator B Goto 1 Out1

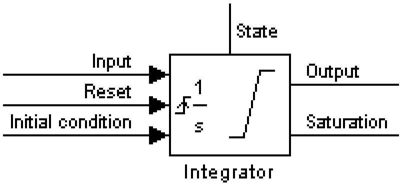
In this model, the initial condition of the integrator in A depends on the value of the state port of the integrator in B, and vice versa. The values of the state ports are updated earlier in the simulation time step than the values of the integrator output ports. Therefore, Simulink can compute the 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.



Ports

Input

`Port_1` — Integrand signal

real scalar or array

Signal that needs to be integrated.

Data Types: `double`

`External Reset` — Reset state to initial conditions

real scalar or array

Reset the state to the specified initial conditions based on an external signal. See "Resetting the State" on page 1-1003.

Dependencies

To enable this port, enable the **External Reset** parameter.

Data Types: `Boolean`

x_0 — Initial condition

real scalar or array

Set the initial condition of the block's state from an external signal.

Dependencies

To enable this port, set the **Initial Conditions** parameter to `external`.

Data Types: `double`

Output

`Port_1` — Output signal

real scalar or array

Output the integrated state.

Data Types: `double`

`Port_2` — Show output saturation

-1 | 0 | 1

Indicate when the state is being limited. The signal has a value of `1` when the integral is limited by the specified **Upper saturation limit**. When the signal is limited by the **Lower saturation limit**, the signal value is `-1`. When the integral is between the saturation limits, the signal value is `0`. See “Limiting the Integral” on page 1-1002.

Data Types: `double`

`Port_3` — State

real scalar or array

Output the state of the block. See “About the State Port” on page 1-1004.

Dependencies

Enable this port by enabling the **Show state port** parameter.

Data Types: `double`

Parameters

`External reset` — Reset states to their initial conditions

`none` (default) | `rising` | `falling` | `either` | `level` | `level hold`

Specify the type of trigger to use for the external reset signal.

- 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.

Programmatic Use

Block Parameter: `ExternalReset`
Type: character vector , string
Values: `'none'` | `'rising'` | `'falling'` | `'either'` | `'level'` | `'level hold'`
Default: `'none'`

`Initial condition source` — Select source of initial condition

`internal` (default) | `external`

Select source of initial condition:

- `internal` — Get the initial conditions of the states from the **Initial condition** block parameter.
- `external` — Get the initial conditions of the states from an external block, via the **IC** input port.

Dependencies

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter and enables the **IC** input port.

Programmatic Use

Block Parameter: `InitialConditionSource`
Type: character vector, string
Values: `'internal'` | `'external'`
Default: `'internal'`

`Initial condition` — Initial state

0 (default) | real scalar or array

Set the initial state of the Integrator block.

Tips

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

Setting **Initial condition source** to `internal` enables this parameter.

Setting **Initial condition source** to `external` disables this parameter.

Programmatic Use

Block Parameter: `InitialCondition`
Type: scalar or vector
Default: `'0'`

`Limit output` — Limit block output values to specified range

`off` (default for Integrator) | `on` (default for Integrator Limited)

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

- Selecting this check box limits the block output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.
- Clearing this check box does not limit the block output values.

Dependencies

Selecting this parameter enables the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use

Block Parameter: `LimitOutput`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Upper saturation limit` — Upper limit for the integral

`inf` (default) | scalar | vector | matrix

Specify the upper limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

Dependencies

To enable this parameter, select the **Limit output** check box.

Programmatic Use

Block Parameter: `UpperSaturationLimit`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'inf'`

`Lower saturation limit` — Lower limit for the integral

`-inf` (default) | scalar | vector | matrix

Specify the lower limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

Dependencies

To enable this parameter, select the **Limit output** check box.

Programmatic Use

Block Parameter: `LowerSaturationLimit`
Type: character vector , string
Values: scalar | vector | matrix
Default: `'-inf'`

`Wrap state` — Enable wrapping of states

`off` (default) | `on`

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.

If you specify **Wrapped state upper value** as `inf` and **Wrapped state lower value** as `-inf`, wrapping does not occur.

Dependencies

Selecting this parameter enables **Wrapped state upper value** and **Wrapped state lower value** parameters.

Programmatic Use

Block Parameter: `WrapState`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Wrapped state upper value` — Specify upper value for wrapped state

`pi` (default) | scalar or vector

Upper limit of the block output.

Dependencies

Selecting **Wrap state** enables this parameter.

Programmatic Use
Block Parameter: `WrappedStateUpperValue`
Type: scalar or vector
Values: `'2*pi'`
Default: `'pi'`

`Wrapped state lower value` — Specify lower value for wrap state

`-pi` (default) | scalar or vector

Lower limit of the block output.

Dependencies

Selecting **Wrap state** enables this parameter.

Programmatic Use
Block Parameter: `WrappedStateLowerValue`
Type: scalar or vector
Values: `'0'`
Default: `'-pi'`

`Show saturation port` — Enable saturation output port

`off` (default) | `on`

Select this check box to add a saturation output port to the block. When you clear this check box, the block does not have a saturation output port.

Dependencies

Selecting this parameter enables a saturation output port.

Programmatic Use
Block Parameter: `ShowSaturationPort`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Show state port` — Enable state output port

`off` (default) | `on`

Select this check box to add a state output port to the block. When you clear this check box, the block does not have a state output port.

Dependencies

Selecting this parameter enables a state output port.

Programmatic Use

Block Parameter: `ShowStatePort`
Type: character vector , string
Values: `'off'` | `'on'`
Default: `'off'`

`Absolute tolerance` — Absolute tolerance for block states

`auto` (default) | `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.

Programmatic Use

Block Parameter: `AbsoluteTolerance`
Type: character vector, string, scalar, or vector
Values: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

`Ignore limit and reset when linearizing` — Treat block as unresettable and output unlimited

`off` (default) | `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 reset and output limitation options of the block.

Tip

Use this check box to linearize a model around an operating point that causes the integrator to reset or saturate.

Programmatic Use

Block Parameter: `IgnoreLimit`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`State Name (e.g., 'position')` — Assign unique name to each state

`' '` (default) | character vector | string

- 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, string, cell array, or structure.

Programmatic Use
Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined

Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

- Consider discretizing the model
- Not recommended for production code

See Also

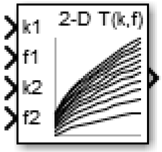
Discrete-Time Integrator | Second-Order Integrator

Introduced before R2006a

Interpolation Using Prelookup

Use precalculated index and fraction values to accelerate approximation of N-dimensional function

Library: Simulink / Lookup Tables



Description

The Interpolation Using Prelookup block is most efficient when used with the Prelookup block. The Prelookup block calculates the index and interval fraction that specify how its input value `u` relates to the breakpoint data set. 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”.

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 for the default column-major algorithm. Therefore, the value must be less than or equal to the **Number of table dimensions**.

For row-major algorithms, the interpolation starts from the first or lowest dimension of the table data.

For nonzero values, the subtable selection behavior is optimized for row-major layout when you select the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter.

Suppose that you have 3-D table data in your Interpolation Using Prelookup block. This behavior applies for the column-major algorithm.

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 `s3` that you use to select and interpolate 2-D tables

Number of Selection Dimensions	Action by the Block	Block Appearance
2	Interpolates the first dimension and selects the second and third dimensions	Displays two input ports with the labels s2 and s3 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.

For 2-D or n-D interpolation without subtable selection, the column-major and row-major algorithms may differ in the order of output calculations, causing slight different numerical results.

Ports

Input

k1 — Index, k, for the first dimension of the table
scalar | vector | matrix

Zero-based index, k, specifying the interval containing the input u for the first dimension of the table.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

f1 — Fraction, f, for the first dimension of the table
scalar | vector | matrix

Fraction, f, representing the normalized position of the input within the interval, k, for the first dimension of the table.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

kn — Index, k, for the n-th dimension of the table
scalar | vector | matrix

Zero-based index, k, specifying the interval containing the input u for the n-th dimension of the table.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

fn — Fraction, f, for the n-th dimension of the table

scalar | vector | matrix

Fraction, f, representing the normalized position of the input within the interval, k, for the n-th dimension of the table.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

kf1 — Input containing index k and fraction f

bus object

Inputs to the **kf1** port contain index k and fraction f specified as a bus object.

Dependencies

To enable this port, select the **Require index and fraction as a bus** check box.

The number of available kf input ports depends on the value of the **Number of dimensions** and **Number of sub-table selection dimensions** parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus

kfn — Input containing index k and fraction f

bus object

Inputs to the **kfn** port contain index k and fraction f for the n-th dimension of the input, specified as a bus object.

Dependencies

To enable this port, select the **Require index and fraction as a bus** check box.

The number of available **kf** input ports depends on the value of the **Number of dimensions** and **Number of sub-table selection dimensions** parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus

`sm` — Select and interpolate a subset of the table data

scalar | vector | matrix

The block uses inputs to the s*m*, s*m*-1... port to perform selection and interpolation within the subtables. *m* equals the **Number of dimensions - Number of sub-table selection dimensions**.

Dependencies

To enable this port, the **Number of sub-table selection dimensions** must be a positive integer less than or equal to the **Number of dimensions**.

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

`T` — Table data

scalar | vector | matrix | n-d array

Table data values provided as input to port **T**. 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 (`k`) and interval fraction (`f`) values fed from Prelookup blocks.

Dependencies

To enable this port, set **Source** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Approximation of N-dimensional function

scalar | vector | matrix

Approximation of N-dimensional function, computed by interpolating (or extrapolating) table data using values from the input index, `k`, and fraction, `f`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Main

Table data

`Number of dimensions` — Number of table data dimensions

2 (default) | integer between 1 and 30

Specify the number of dimensions that the table data must have. The **Number of dimensions** defines the number of independent variables for the table.

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.

Programmatic Use
Block Parameter: `NumberOfTableDimensions`
Type: character vector
Values: `'1' | '2' | '3' |'4' | ... | '30'`
Default: `'2'`

`Require index and fraction as bus` — Index and fraction inputs can be combined in a bus

`off` (default) | `on`

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

Programmatic Use
Block Parameter: `RequireIndexFractionAsBus`

Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Specification` — Choose how to enter table data

`Explicit values` (default) | `Lookup table object`

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.

Programmatic Use

Block Parameter: `TableSpecification`
Type: character vector
Values: `'Explicit values' | 'Lookup table object'`
Default: `'Explicit values'`

`Source` — Source of table data

`Dialog` (default) | `Input port`

Specify whether to enter table data in the dialog box or to inherit the data from an input port. 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

Dependencies

To enable this parameter, set **Specification** to `Explicit values`.

Programmatic Use

Block Parameter: `TableSource`
Type: character vector
Values: `'Dialog' | 'Input port'`
Default: `'Dialog'`

`Value` — Specify table data values

`sqrt([1:11]' * [1:11])` (default) | multidimensional array of table data

Specify table data as an `N`-D array, where `N` is the value of the **Number of dimensions** parameter. 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, the block inherits table attributes from the **T** input port.

To edit lookup tables using the Lookup Table Editor, click **Edit** (see “Edit Lookup Tables”).

Dependencies

To enable this parameter and explicitly specify table values on the dialog box, you must set **Specification** to `Explicit values` and **Source** to `Dialog`.

Programmatic Use

Block Parameter: `Table`
Type: character vector
Values: `scalar, vector, matrix, or N-D array`
Default: `'sqrt([1:11]' * [1:11])'`

Name — Name of an existing `Simulink.LookupTable` object

`Simulink.LookupTable object`

Specify the name of an existing `Simulink.LookupTable` object. An existing lookup table object references Simulink breakpoint objects.

Dependencies

To enable this parameter, set **Specification** to `Lookup table object`.

Programmatic Use

Block Parameter: `LookupTableObject`
Type: character vector
Value: `Simulink.LookupTable object`
Default: `''`

Algorithm

Interpolation method — Select `Linear point-slope`, `Flat`, `Nearest`, or `Linear Lagrange`

`Linear point-slope` (default) | `Nearest` | `Flat` | `Linear Lagrange`

Specify the method the block uses to interpolate table data. You can select `Linear point-slope`, `Flat`, `Nearest`, or `Linear Lagrange`. See “Interpolation Methods” for more information.

Programmatic Use
Block Parameter: `InterpMethod`
Type: character vector
Values: `'Flat' | 'Linear point-slope' | 'Nearest' | 'Linear Lagrange'`
Default: `'Linear point-slope'`

`Extrapolation method` — Method of handling input that falls outside the range of the breakpoint data set

`Linear` (default) | `Clip`

Specify the method the block uses to extrapolate values for all inputs that fall outside the range of the breakpoint data set. You can select `Clip` or `Linear`. See “Extrapolation Methods” for more information.

If the extrapolation method is `Linear`, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is Linear Lagrange, the extrapolation method inherits the Linear Lagrange equation to compute the extrapolated value.

Dependencies

To enable the **Extrapolation method** parameter, set the **Interpolation method** to `Linear`.

The Interpolation Using Prelookup block does not support `Linear` extrapolation when the input or output signals specify integer or fixed-point data types.

Programmatic Use
Block Parameter: `ExtrapMethod`
Type: character vector
Values: `'Clip' | 'Linear'`
Default: `'Linear'`

`Valid index input may reach last index` — Allow inputs to access the last elements of table data

`off` (default) | `on`

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
on	Returns the value of the last element in a dimension of its table when: • k indexes the last table element in the corresponding dimension • f is 0
off	Returns the value of the last element in a dimension of its table when: • k indexes the next-to-last table element in the corresponding dimension • f is 1

Dependencies

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.

Programmatic Use

Block Parameter: `ValidIndexMayReachLast`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Diagnostic for out-of-range input` — Block action when input is out of range

`None` (default) | `Warning` | `Error`

Specify whether to produce a warning or error when the input is out of range. Options include:

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `DiagnosticForOutOfRangeInput`
Type: character vector
Values: `'None' | 'Warning' | 'Error'`
Default: `'None'`

`Number of sub-table selection dimensions` — Number of dimensions of the output computation subtable

`0` (default) | positive integer, less than or equal to the number of table 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 nonzero values, the subtable selection behavior is optimized for row-major layout when you select the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter.

For more information, see “How The Block Interpolates a Subset of Table Data” on page 1-1019.

Programmatic Use
Block Parameter: `NumSelectionDims`
Type: character vector
Values: `'0' | '1' | '2' | '3' | '4' | ... | Number of table dimensions`
Default: `'0'`

Code generation

`Remove protection against out-of-range index in generated code` — Remove code that checks for out-of-range index inputs

`off` (default) | `on`

Check Box	Result	When to Use
`on`	Generated code does not include conditional statements to check for out-of-range index 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
`off`	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”.

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.

Programmatic Use

Block Parameter: `RemoveProtectionIndex`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Data Types

`Table data` — Data type of table values

`Inherit: Same as output` (default) | `Inherit: Inherit from 'Table data'` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

---

Programmatic Use

Block Parameter: `TableDataTypeStr`
Type: character vector
Values: `'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)'` | `'<data type expression>'`
Default: `'Inherit: Same as input'`

`Table minimum` — Minimum value of table data

`[]` (default) | scalar

Specify the minimum value for table data as a finite, real, double, scalar. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `TableMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Table maximum` — Maximum value of table data

`[]` (default) | scalar

Specify the maximum value for table data as a finite, real, double, scalar. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `TableMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Intermediate results` — Data type of intermediate results

`Inherit: Inherit via internal rule` (default) | `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)` | `<data type expression>`

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

Tip Use this parameter to specify higher precision for internal computations than for table data or output data.

Programmatic Use
Block Parameter: `IntermediateResultsDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | '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)' | '<data type expression>'`
Default: `'Inherit: Same as input'`

`Output` — Output data type

`Inherit: Inherit from 'Table data'` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'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)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'`
Default: `'Inherit: Inherit from table data'`

`Output minimum` — Minimum value the block can output

`[]` (default) | scalar

Specify the minimum value that the block should output as a finite, real-valued scalar. 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” (Simulink Coder).

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Output maximum` — Maximum value the block can output

`[]` (default) | scalar

Specify the maximum value that the block should output as a finite, real-valued scalar. 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” (Simulink Coder).

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Internal rule priority` — Internal rule for intermediate calculations

`Speed` (default) | `Precision`

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.

Programmatic Use
Block Parameter: `InternalRulePriority`
Type: character vector
Values: `'Speed' | 'Precision'`
Default: `'Speed'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Convergent` | `Ceiling` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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 (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
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™.

For more information about HDL code generation, see “Supported Blocks” (HDL Coder).

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Prelookup

Topics

“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”
“Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks” (Simulink Coder)
“Interpolation with Subtable Selection Algorithm for Row-Major Array Layout” (Simulink Coder)

Introduced in R2006b

Interpreted MATLAB Function

Apply MATLAB function or expression to input

Library: Simulink / 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.

Ports

Input

In — Input of a Interpreted MATLAB function

scalar | vector | matrix

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.

Data Types: `double`

Output

`Out` — Output of a Interpreted MATLAB function

scalar | vector | matrix

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.

Data Types: `double`

Parameters

`MATLAB Function` — Specify the function or expression

function (default)

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

scalar | matrix | vector | inherited

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 block output signal type

auto (default) | real | complex

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` — Output a 2-D array as a 1-D array
off (default) | on

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` — Specify sample time in the block
scalar

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”.

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

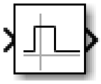
See Also

Fcn | MATLAB Function | MATLAB System

Introduced in R2011a

Interval Test

Determine if signal is in specified interval
Library: Simulink / Logic and Bit Operations



Description

The Interval Test block outputs true (1) if the input is between the values specified by the **Lower limit** and **Upper limit** parameters. The block outputs false (0) 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 you select the **Interval closed on left** and **Interval closed on right** check boxes.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.

Limitations

When the input signal is an enumerated type, the **Upper limit** and **Lower limit** values must be of the same enumerated type.

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

Output

Port_1 — Output signal
scalar | vector | matrix | N-D array

Output signal indicating whether the input values fall within the specified interval. You can specify the **Output data type** as `boolean` or `uint8`.

Data Types: `uint8` | `Boolean`

Parameters

`Interval closed on right` — Include upper limit value

`on` (default) | `off`

When you select this check box, the **Upper limit** is included in the interval for which the block outputs true (`1`).

Programmatic Use
Block Parameter: `IntervalClosedRight`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

`Upper limit` — Upper limit of interval

`0.5` (default) | scalar | vector | matrix | N-D array

The upper limit of the interval for which the block outputs true (`1`).

Programmatic Use
Block Parameter: `uplimit`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'0.5'`

`Interval closed on left` — Include lower limit value

`on` (default) | `off`

When you select this check box, the **Lower limit** is included in the interval for which the block outputs true (`1`).

Programmatic Use
Block Parameter: `IntervalClosedLeft`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

`Lower limit` — Lower limit of interval

`-0.5` (default) | scalar | vector | matrix | N-D array

The lower limit of the interval for which the block outputs true (`1`).

Programmatic Use
Block Parameter: `lowlimit`
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: `'-0.5'`

`Output data type` — Output data type

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean' | 'uint8'`
Default: `'boolean'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

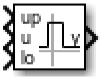
See Also

Interval Test Dynamic

Introduced before R2006a

Interval Test Dynamic

Determine if signal is in specified interval
Library: Simulink / Logic and Bit Operations



Description

The Interval Test Dynamic block outputs true (`1`) if the input is between the values of the external signals **up** and **lo**. The block outputs false (`0`) if the input is outside those values. To control how the block handles input values that are equal to the signal **lo** or the signal **up**, use the **Interval closed on left** and **Interval closed on right** check boxes.

Ports

Input

up — Upper limit of interval
scalar | vector | matrix | N-D array

Upper limit of interval, specified as a scalar, vector, matrix, or N-D array.

Limitations

When the input signal is an enumerated type, the **up** and **lo** signals must be of the same enumerated type.

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

u — Input signal
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.

Limitations

When the input signal is an enumerated type, the **up** and **lo** signals must be of the same enumerated type.

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

`lo` — Lower limit of interval

scalar | vector | matrix | N-D array

Lower limit of interval, specified as a scalar, vector, matrix, or N-D array.

Limitations

When the input signal is an enumerated type, the **up** and **lo** signals must be of the same enumerated type.

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

Output

y — Output signal

scalar | vector | matrix | N-D array

Output signal indicating whether the input values fall within the specified interval. You can specify the **Output data type** as `boolean` or `uint8`.

Data Types: `uint8` | `Boolean`

Parameters

`Interval closed on right` — Include upper limit value

on (default) | `off`

When you select this check box, the value of the signal connected to the **up** input port is included in the interval for which the block outputs true (1).

Programmatic Use

Block Parameter: `IntervalClosedRight`
Type: character vector

Values: `'on' | 'off'`
Default: `'on'`

`Interval closed on left` — Include lower limit value

`on` (default) | `off`

When you select this check box, the value of the signal connected to the **lo** input port is included in the interval for which the block outputs true (`1`).

Programmatic Use
Block Parameter: `IntervalClosedLeft`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

`Output data type` — Output data type

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'boolean' | 'uint8'`
Default: `'boolean'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

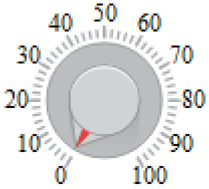
Interval Test

Introduced before R2006a

Knob

Tune parameter value with dial

Library: Simulink / Dashboard



Description

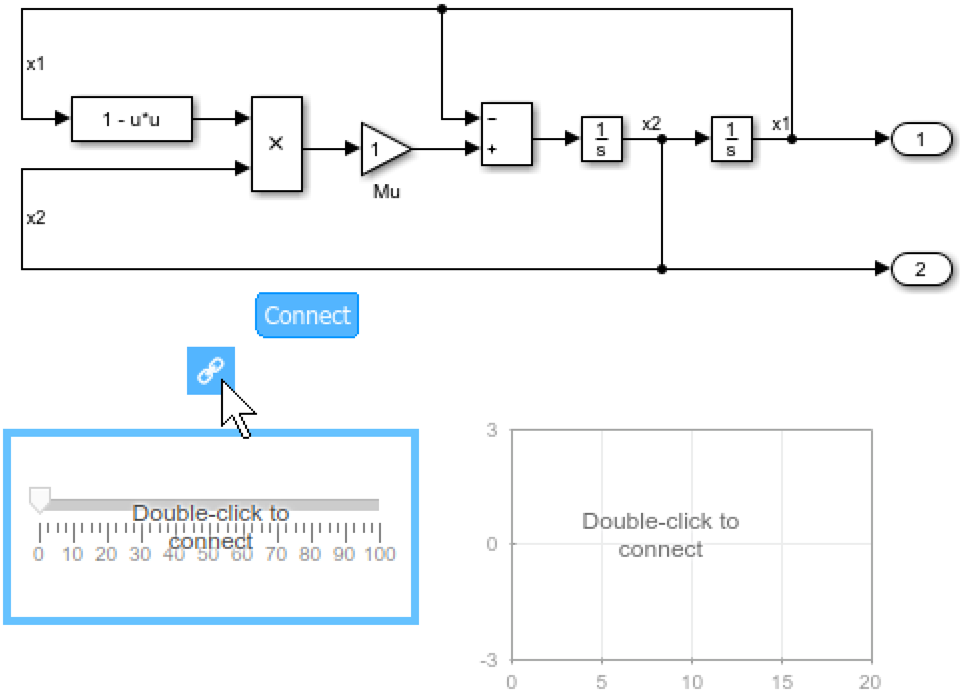
The Knob block tunes the value of the connected block parameter to during simulation. For example, you can connect the Knob block to a Gain block in your model and adjust its value during simulation. You can modify the range of the Knob block's scale to fit your data. Use the Knob block with other Dashboard blocks to create an interactive dashboard to control your model.

Connecting Dashboard Blocks

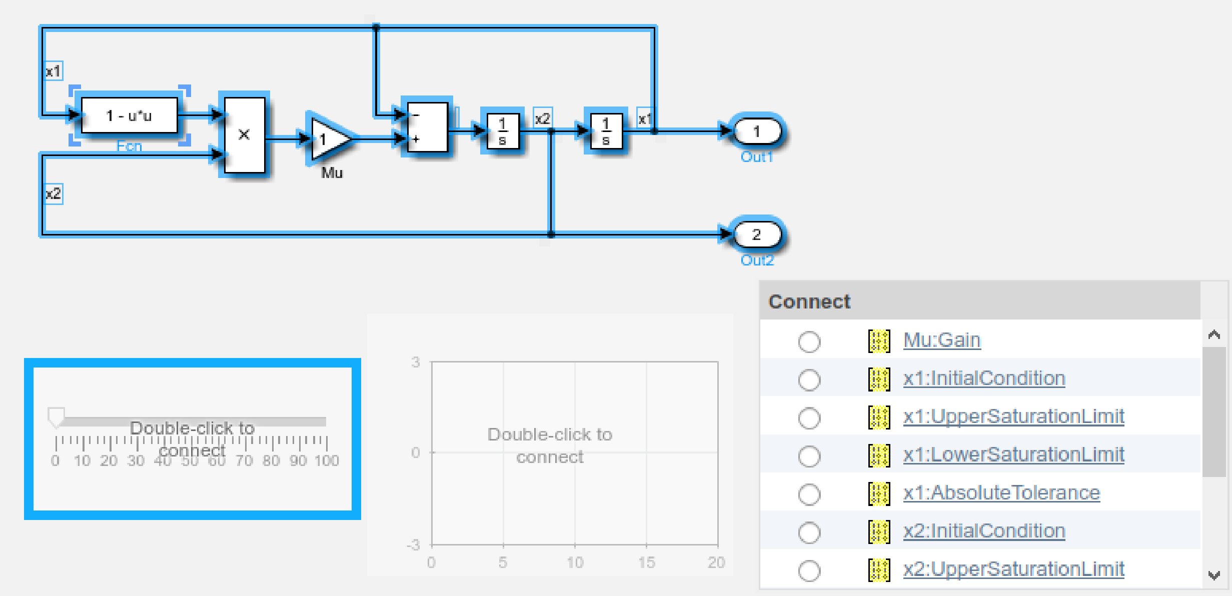
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○	Mu:Gain
○	x1:InitialCondition
○	x1:UpperSaturationLimit
○	x1:LowerSaturationLimit
○	x1:AbsoluteTolerance
○	x2:InitialCondition
○	x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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 arc. 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 arc. 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 arc. 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.

See Also

Rotary Switch | Slider

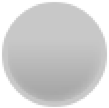
Topics

"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



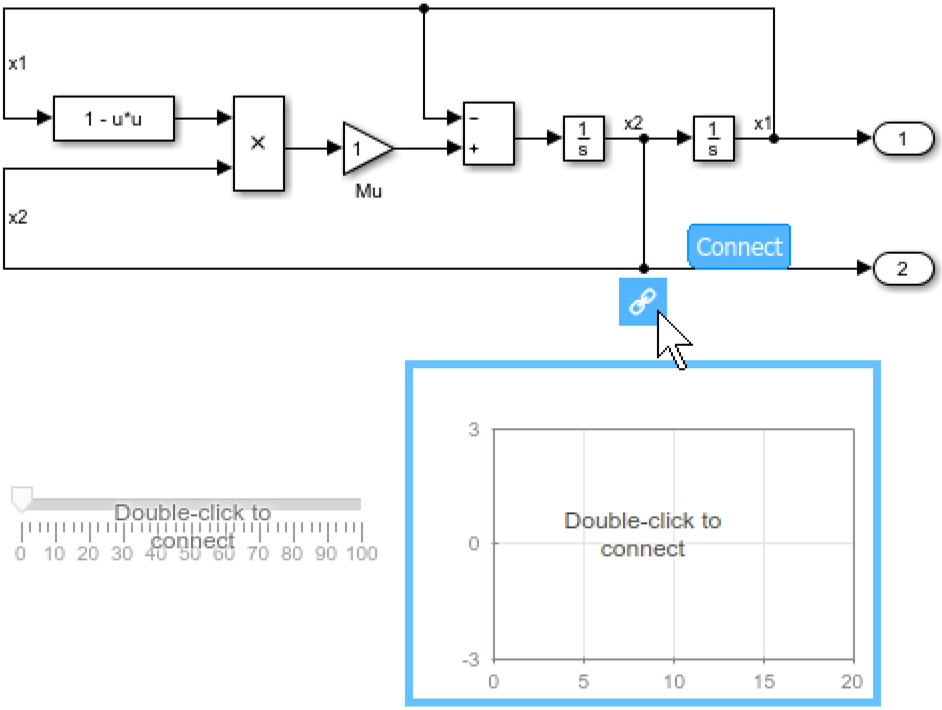
Description

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.

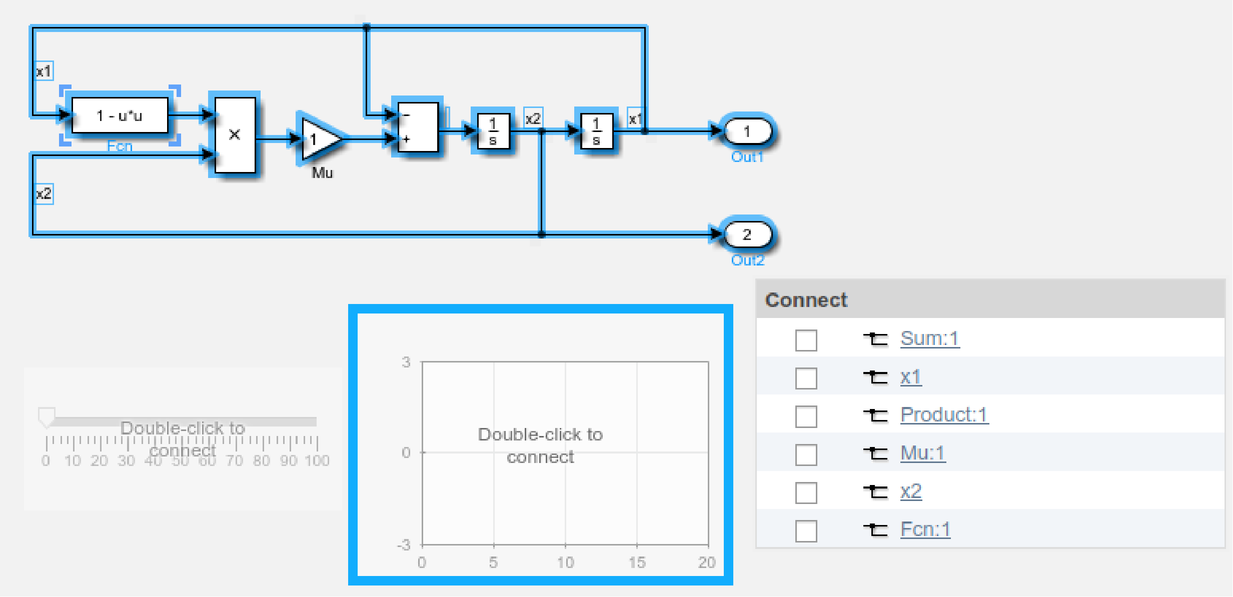
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect	
□	Sum:1
□	x1
□	Product:1
□	Mu:1
□	x2
□	Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced 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.

Parameters

`Connection` — Select a signal to connect and display

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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.

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: Simulink / 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

matlabfile (default) | S-Function name

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).

Programmatic Use
Block Parameter: `FunctionName`
Type: character vector
Values: `'matlabfile'` | S-Function name
Default: `'matlabfile'`

`Parameters` — Specify values of parameters for this block
no default (default)

Specify values of parameters for this block.

Programmatic Use
Block Parameter: `Parameters`
Type: character vector
Values: values of block parameters
Default: `' '`

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a]
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No

[a] Level-1 MATLAB S-functions support only the double data type. Level-2 MATLAB S-functions support all data types that Simulink supports.

Extended Capabilities

C/C++ Code Generation

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

- If a corresponding TLC file is available, the Level-2 MATLAB S-Function block uses the TLC file to generate code, otherwise code generation throws an error.
- Real-time code generation does not support S-functions that call MATLAB.
- Actual data type or capability support depends on block implementation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Level-1 MATLAB S-functions support only the `double` data type. Level-2 MATLAB S-functions support all data types that Simulink supports.

See Also

Blocks

MATLAB Function | MATLAB System | S-Function | S-Function Builder | Simulink Function | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

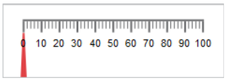
Topics

“What Is an S-Function?”
“Write Level-2 MATLAB S-Functions”
“Design and Create a Custom Block”

Introduced in R2010b

Linear Gauge

Display input value on linear scale
Library: Simulink / Dashboard



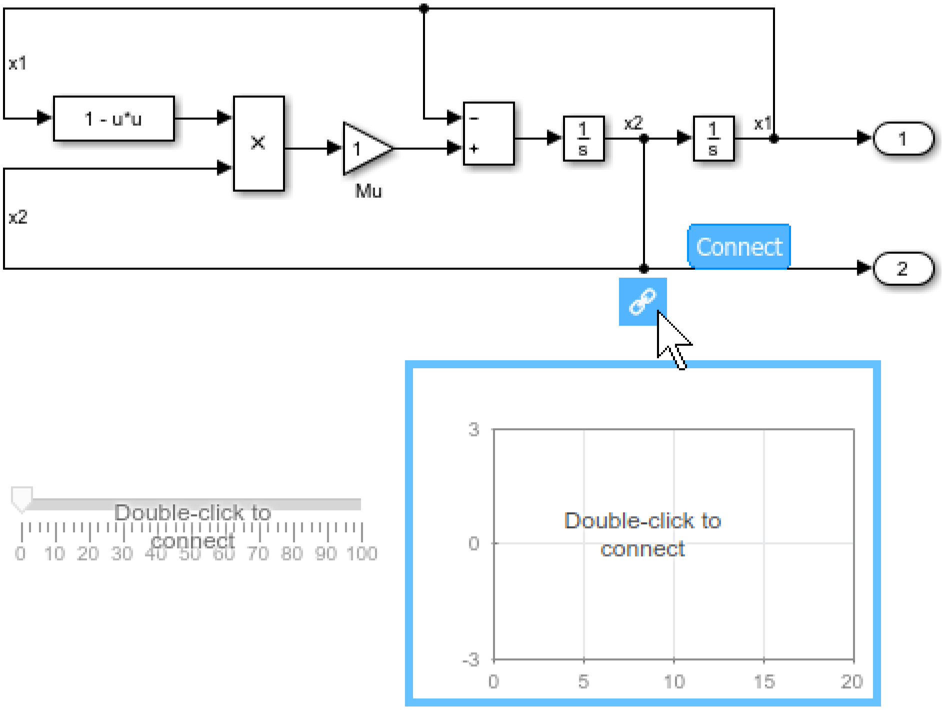
Description

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.

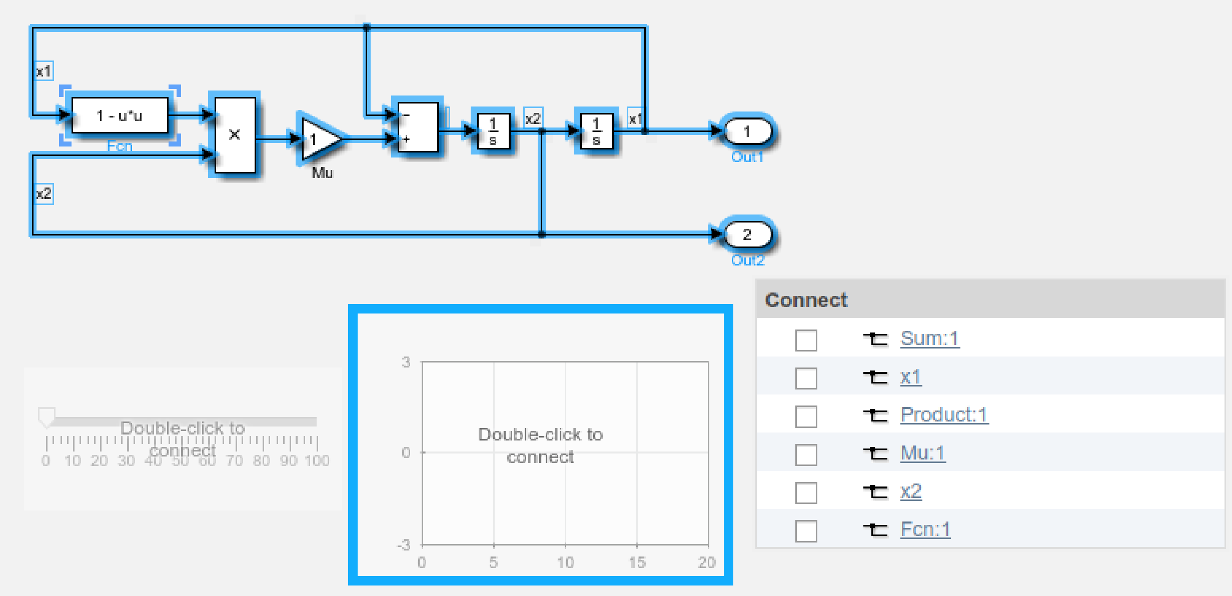
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

☐	Sum:1
☐	x1
☐	Product:1
☐	Mu:1
☐	x2
☐	Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced 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.

Parameters

`Connection` — Select a signal to connect and display

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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 arc. 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 arc. 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 arc. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

`Scale Colors` — Color indications on gauge arc

colors for arc ranges

Color specifications for ranges on the arc. 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.

See Also

Custom Gauge | Gauge | Half Gauge | Quarter Gauge

Topics

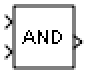
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a

Logical Operator

Perform specified logical operation on input

Library: Simulink / Commonly Used Blocks
Simulink / Logic and Bit Operations



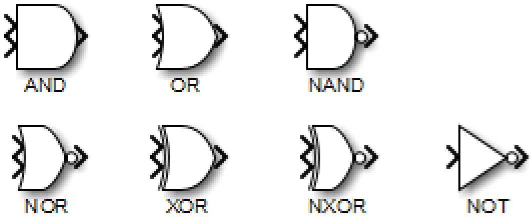
Description

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:

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 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.

Ports

Input

`Port_1` — First input signal

scalar | vector | matrix

First input signal, specified as a scalar, vector, or matrix.

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

`Port_2` — Second input signal

scalar | vector | matrix

Second input signal, specified as a scalar, vector, or matrix.

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

`Port_N` — Nth input signal

scalar | vector | matrix

Nth input signal, specified as a scalar, vector, or matrix.

Dependencies

To enable additional input ports, use the **Number of input ports** parameter.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal, consisting of zeros and ones, with the same dimensions as the input. You control the output data type with the **Require all inputs and output to have the same data type** and **Output data type** parameters.

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

Parameters

Main

`Operator` — Logical operator

AND (default) | OR | NAND | NOR | XOR | NXOR | NOT

Select the logical operator to apply to block inputs.

- `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
- `XOR` — TRUE if the input is FALSE

Programmatic Use
Block Parameter: `Operator`
Type: character vector
Values: `'AND' | 'OR' | 'NAND' | 'NOR' | 'XOR' | 'NXOR' | 'NOT'`
Default: `'AND'`

`Number of input ports` — Number of inputs

2 (default) | positive integer

Specify the number of block inputs as a positive integer.

Programmatic Use
Block Parameter: `Inputs`
Type: character vector
Values: positive integer
Default: `'2'`

Dependencies

This parameter is not available when you set the **Operator** to `NOT`.

`Icon shape` — Icon shape

`rectangular` (default) | `distinctive`

Specify the shape of the block icon.

- `rectangular` — Results 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.

Programmatic Use
Block Parameter: `IconShape`
Type: character vector
Values: `'rectangular' | 'distinctive'`
Default: `'rectangular'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Data Type

`Require all inputs and output to have the same data type` — Require all ports to have same data type

`off` (default) | `on`

To require that all block inputs and the output have the same data type, select this check box. When you clear this check box, the inputs and output can have different data types.

Programmatic Use
Block Parameter: `AllPortsSameDT`
Type: character vector

Values: `'off' | 'on`
Default: `'off'`

`Output data type` — Output data type

`boolean` (default) | `Inherit: Logical (see Configuration Parameters: Optimization)` | `fixdt(1,16)` | `<data type expression>`

Specify the output data type. When you select:

- `boolean` — the block output has data type `boolean`.
- `Inherit: Logical (see Configuration Parameters: Optimization)` — the block 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.
- `fixdt(1,16)` — the block output has the specified fixed-point data type `fixdt(1,16)`.

  Tip Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.
- `<data type expression>` — the block output has the data type you specify as a data type expression, 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`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Logical (see Configuration Parameters: Optimization)' | 'boolean' | 'fixdt(1,16)' | '<data type expression>'`

Default: `'boolean'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Logical Operator.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

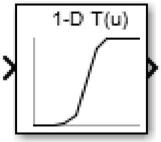
“Implement logic signals as Boolean data (vs. double)” | Combinatorial Logic | Relational Operator

Introduced before R2006a

1-D Lookup Table

Approximate one-dimensional function

Library: Simulink / Lookup Tables



Description

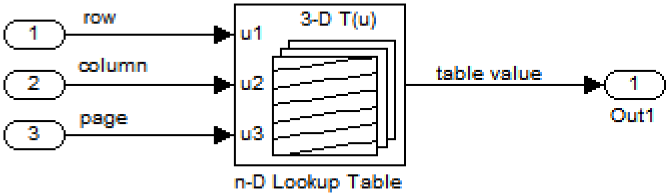
Supported Block Operations

The 1-D, 2-D, and n-D Lookup Table blocks evaluate a sampled representation of a function in N variables

$$y = F(x_1, x_2, x_3, \ldots, x_N)$$

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 (Linear point-slope), Lagrange (Linear Lagrange), nearest, 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” for a description of the port order for various block orientations.

When the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter is set, the 2-D and n-D Lookup Table block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms may differ in the order of the output calculations, possibly resulting in slightly different numerical values. This capability requires a Simulink Coder or Embedded Coder license. For more information on row-major support, see “Code Generation of Matrices and Arrays” (Simulink Coder).

Specification of Breakpoint and Table Data

These 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` and “Identify questionable fixed-point operations” (Embedded Coder).

How the Block Generates Output

The n-D, 1-D and 2-D Lookup Table blocks generate 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

When block inputs...	The n-D Lookup Table block...
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 = table(k)`, interpolation is unnecessary.

Ports

Input

u1 — First-dimension (row) inputs

scalar | vector | matrix

Real-valued inputs to the **u1** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: `0:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output computed by looking up or estimating table values

scalar | vector | matrix

Output generated 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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Table and Breakpoints

`Number of table dimensions` — Number of lookup table dimensions

1 (default) | 2 | 3 | 4 | ... | 30

Enter the number of dimensions of the lookup table. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

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.

Programmatic Use

Block Parameter: `NumberOfTableDimensions`
Type: character vector
Values: `'1' | '2' | '3' | '4' | ... | 30`
Default: `'1'`

`Data specification` — Method of table and breakpoint specification

`Table and breakpoints` (default) | `Lookup table object`

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**
  - **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.

Programmatic Use
Block Parameter: `DataSpecification`
Type: character vector
Values: `'Table and breakpoints' | 'Lookup table object'`
Default: `'Table and breakpoints'`

`Name` — Name of the lookup table object

`[]` (default) | `Simulink.LookupTable` object

Enter the name of the lookup table (`Simulink.LookupTable`) object.

Dependencies

To enable this parameter, set **Data specification** to `Lookup table object`.

Programmatic Use
Block Parameter: `LookupTableObject`
Type: character vector
Values: name of a `Simulink.LookupTable` object
Default: `''`

`Table data` — Define the table of output values

`tanh([-5:5])` (default) | vector of values

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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `Table`
Type: character vector
Values: vector of table values
Default: `'tanh([-5:5])'`

`Breakpoints specification` — Method of breakpoint specification

`Explicit values` (default) | `Even spacing`

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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `BreakpointsSpecification`
Type: character vector
Values: `'Explicit values' | 'Even spacing'`
Default: `'Explicit values'`

`Breakpoints` — Explicit breakpoint values, or first point and spacing of breakpoints

`[10,22,31]` (default) | 1-by-n or n-by-1 vector of monotonically increasing values

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 `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.
- 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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1`
Type: character vector
Values: 1-by-n or n-by-1 vector of monotonically increasing values
Default: `'[10, 22, 31]'`

`First point` — First point in evenly spaced breakpoint data

1 (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite, scalar. This parameter is available when **Breakpoints specification** is set to `Even spacing`.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`, and **Breakpoints specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1FirstPoint`
Type: character vector
Values: real-valued, finite, scalar
Default: `'1'`

Spacing — Spacing between evenly spaced breakpoints

1 (default) | scalar

Specify the spacing between points in your evenly spaced breakpoint data.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`, and **Breakpoints specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1Spacing`
Type: character vector
Values: positive, real-valued, finite, scalar
Default: `'1'`

`Edit table and breakpoints` — Launch Lookup Table Editor dialog box

button

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.

Algorithm

Lookup method

`Interpolation method` — Method of interpolation between breakpoint values

`Linear point-slope` (default) | `Flat` | `Nearest` | `Linear Lagrange` | `Cubic spline`

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. For more information on interpolation methods, see "Interpolation Methods".

Dependencies

If you select `Cubic spline`, the block supports only scalar signals. The other interpolation methods support nonscalar signals.

Programmatic Use

Block Parameter: `InterpMethod`
Type: character vector

Values: `'Linear point-slope' | 'Flat' | 'Nearest' | 'Linear Lagrange' | 'Cubic spline'`
Default: `'Linear point-slope'`

`Extrapolation method` — Method of handling input values that fall outside the range of a breakpoint data set

`Clip` (default) | `Linear` | `Cubic spline`

Select `Clip`, `Linear`, or `Cubic spline`. See “Extrapolation Methods” for more information.

If the extrapolation method is `Linear`, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is Linear Lagrange, the extrapolation method inherits the Linear Lagrange equation to compute the extrapolated value.

Dependencies

To select `Cubic spline` for **Extrapolation method**, you must also select `Cubic spline` for **Interpolation method**.

Programmatic Use

Block Parameter: `ExtrapMethod`
Type: character vector
Values: `'Linear' | 'Clip' | 'Cubic spline'`
Default: `'Linear'`

`Index search method` — Method of calculating table indices

`Evenly spaced points` (default) | `Linear search` | `Binary search`

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`.

Programmatic Use
Block Parameter: `IndexSearchMethod`
Type: character vector
Values: `'Binary search' | 'Evenly spaced points' | 'Linear search'`
Default: `'Binary search'`

`Begin index search using previous index result` — Start using the index from the previous time step

`off` (default) | `on`

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.

Dependencies

To enable this parameter, set **Index search method** to `Linear search` or `Binary search`.

Programmatic Use
Block Parameter: `BeginIndexSearchUsing PreviousIndexResult`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Diagnostic for out-of-range input` — Block action when input is out of range

`None` (default) | `Warning` | `Error`

Specify whether to produce a warning or error when the input is out of range. Options include:

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `DiagnosticForOutOfRangeInput`
Type: character vector
Values: `'None' | 'Warning' | 'Error'`
Default: `'None'`

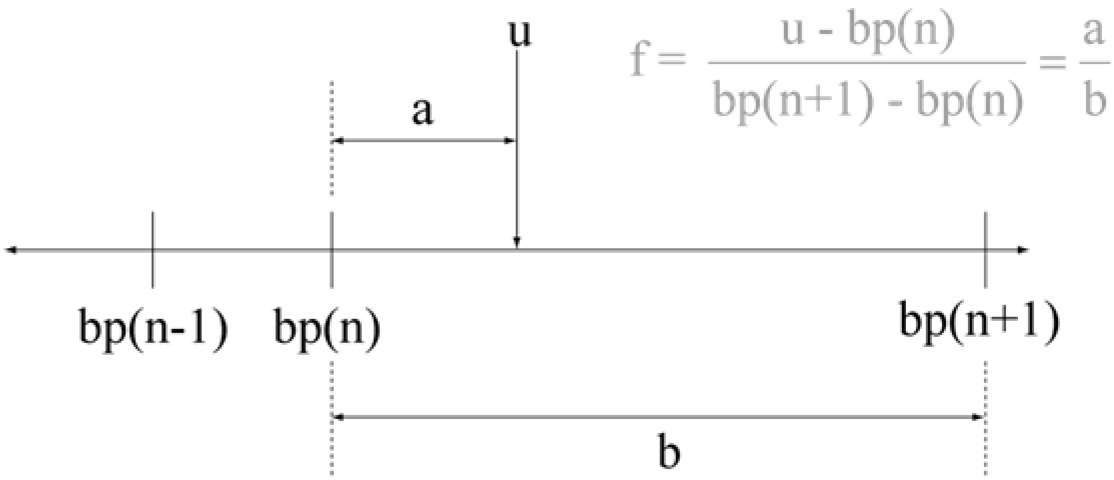
`Use last table value for inputs at or above last breakpoint` — Method for computing output for inputs at or above last breakpoint

`off` (default) | `on`

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.

Check Box	Block Uses Index Of The...	Interval Fraction
Selected	Last element of breakpoint data on the **Table and Breakpoints** tab	0
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 *f* 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 check box, the index is that of the next-to-last element (`4`) and the interval fraction is 1.

Dependencies

To enable this parameter, set:

- **Interpolation method** to `Linear`.
- **Extrapolation method** to `Clip`.

Programmatic Use

Block Parameter: `UseLastTableValue`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Input settings

`Use one input port for all input data` — Use only one input port

`off` (default) | `on`

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.

Programmatic Use
Block Parameter: `UseOneInputPortForAllInputData`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Code generation

Remove protection against out-of-range input in generated code — Remove code that checks for out-of-range input values

`off` (default) | `on`

Specify whether or not to include code that checks for out-of-range input values.

Check Box	Result	When to Use
`on`	Generated code does not include conditional statements to check for out-of-range breakpoint inputs. When the input is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.	For code efficiency
`off`	Generated code includes conditional statements to check for out-of-range 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”.

Programmatic Use

Block Parameter: `RemoveProtectionInput`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Support tunable table size in code generation` — Enable tunable table size in the generated code

`off` (default) | `on`

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.

Dependencies

If you set **Interpolation method** to `Cubic spline`, this check box is not available.

Programmatic Use

Block Parameter: `SupportTunableTableSize`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see "Specify Sample Time".

Dependencies

This parameter is not visible unless it is explicitly set to a value other than `-1`. To learn more, see "Blocks for Which Sample Time Is Not Recommended".

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

`Maximum indices for each dimension` — Maximum index value for each table dimension

`[]` (default) | `scalar or vector of positive integer values`

Example: `[4 6]` for a 5-by-7 table

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`

Examples of valid specifications include:

- `[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 1-1100.

Dependencies

To enable this parameter, 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.

Programmatic Use
Block Parameter: `MaximumIndicesForEachDimension`
Type: character vector
Values: scalar or vector of positive integer values
Default: `'[]'`

Data Types

`Table data` — Data type of table data

`Inherit: Same as output` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

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

Programmatic Use
Block Parameter: `TableDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`
Default: `'Inherit: Same as output'`

`Table data Minimum` — Minimum value of the table data

`[]` | `scalar`

Specify the minimum value for table data. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `TableMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Table data Maximum` — Maximum value of the table data

`[]` | `scalar`

Specify the maximum value for table data. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `TableMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Breakpoints` — Breakpoint data type

`Inherit: Same as corresponding input` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class name>` | `<data type expression>`

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 class, for example, an enumerated data type class
- 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)`

Tip

- Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.
- If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
- For enumerated data, **Extrapolation method** must be `Clip`.
- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

This is a limitation for using enumerated data with this block:

- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

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

Programmatic Use
Block Parameter: `BreakpointsForDimension1DataTypeStr | BreakpointsForDimension2DataTypeStr| ... | BreakpointsForDimension30DataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`
Default: `'Inherit: Same as corresponding input'`

`Breakpoints Minimum` — Minimum value breakpoint data can have

`[]` | `scalar`

Specify the minimum value that a set of breakpoint data can have. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `BreakpointsForDimension1Min | BreakpointsForDimension2Min | ... | BreakpointsForDimension30Min`
Type: character vector
Values: scalar
Default: `'[]'`

`Breakpoints Maximum` — Maximum value breakpoint data can have

`[]` | `scalar`

Specify the maximum value that a set of breakpoint data can have. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `BreakpointsForDimension1Max | BreakpointsForDimension2Max | ... | BreakpointsForDimension30Max`
Type: character vector
Values: scalar
Default: `'[]'`

`Fraction` — Fraction data type

`Inherit: Inherit via internal rule` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `FractionDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'double' | 'single' | 'fixdt(1,16,0)'|'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Intermediate results` — Intermediate results data type

`Inherit: Same as output` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

Tip Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.

Programmatic Use
Block Parameter: `IntermediateResultsDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`
Default: `'Inherit: Same as output'`

Output — Output data type

`Inherit: Same as input` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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 data type attributes. For more information, see "Specify Data Types Using Data Type Assistant".

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression'`
Default: `'Inherit: Same as first input'`

Output Minimum — Minimum value the block can output

`[]` | `scalar`

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Output Maximum` — Maximum value the block can output

`[]` | `scalar`

Specify the maximum value that the block can 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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Internal rule priority` — Internal rule for intermediate calculations

`Speed` (default) | `Precision`

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.

Programmatic Use
Block Parameter: `InternalRulePriority`
Type: character vector
Values: `'Speed' | 'Precision'`
Default: `'Speed'`

`Require all inputs to have the same data type` — Require all inputs to have the same data type

`on` (default) | `off`

Select to require all inputs to have the same data type.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Simplest` (default) | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Zero`

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).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Simplest'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box (`on`).	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.
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Check for Signal Range Errors".	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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Definitions

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';
  ```
- These block parameters apply in the n-D Lookup Table block dialog box.

Parameter	Value
Number of table dimensions	2
Table data	`p.Table`
Breakpoints 1	`p.BP1`
Breakpoints 2	`p.BP2`
Support tunable table size in code generation	`on`
Maximum indices for each dimension	`p.MaxIdx`

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.

Enumerated Values in Lookup Tables

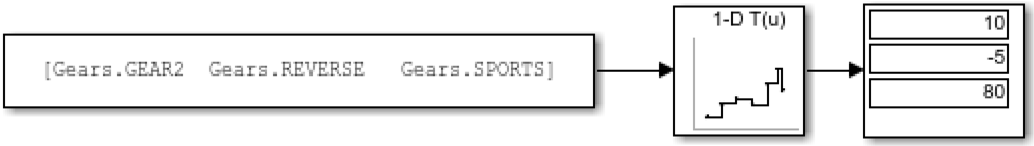
Suppose that you have a lookup table with an enumerated class like this defined:

```
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
        NEUTRAL(0)
    end
end
```

n-D Lookup block has these settings:

- **Number of dimensions** to `1`.
- **Table data** value is `[5 10 20 40 80 -5 0]`.
- **Breakpoints 1** value is `enumeration('Gears')`.
- Interpolation method is Flat.
- For an unordered search, set **Index search method** to `Linear search` and clear the **Begin index search using previous index result** check box.

Simulation produces a vector `[10 -5 80]`, which correspond to `GEAR2`, `REVERSE`, and `SPORTS`.



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™.

The 1-D, 2-D, and N-D Lookup Table blocks have restrictions for HDL code generation. For more information, see “Restrictions” (HDL Coder).

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Direct Lookup Table (n-D) | Interpolation Using Prelookup | Lookup Table Dynamic | Prelookup | `Simulink.Breakpoint` | `Simulink.LookupTable`

Topics

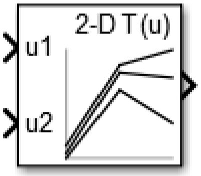
"Import Lookup Table Data from MATLAB"
"About Lookup Table Blocks"
"Anatomy of a Lookup Table"
"Enter Breakpoints and Table Data"
"Guidelines for Choosing a Lookup Table"

Introduced in R2011a

2-D Lookup Table

Approximate two-dimensional function

Library: Simulink / Lookup Tables



Description

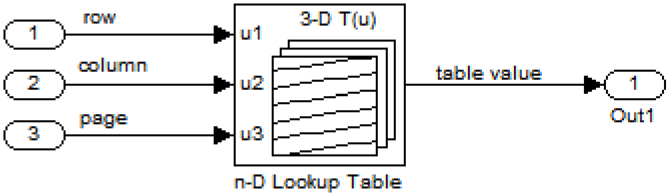
Supported Block Operations

The 1-D, 2-D, and n-D Lookup Table blocks evaluate a sampled representation of a function in N variables

$$y = F(x_1, x_2, x_3, \ldots, x_N)$$

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 (Linear point-slope), Lagrange (Linear Lagrange), nearest, 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” for a description of the port order for various block orientations.

When the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter is set, the 2-D and n-D Lookup Table block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms may differ in the order of the output calculations, possibly resulting in slightly different numerical values. This capability requires a Simulink Coder or Embedded Coder license. For more information on row-major support, see “Code Generation of Matrices and Arrays” (Simulink Coder).

Specification of Breakpoint and Table Data

These 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` and “Identify questionable fixed-point operations” (Embedded Coder).

How the Block Generates Output

The n-D, 1-D and 2-D Lookup Table blocks generate 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

When block inputs...	The n-D Lookup Table block...
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 = table(k)`, interpolation is unnecessary.

Ports

Input

u1 — First-dimension (row) inputs
scalar | vector | matrix

Real-valued inputs to the **u1** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: `0:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

u2 — Second-dimension (column) inputs
scalar | vector | matrix

Real-valued inputs to the **u2** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: `0:10`

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output computed by looking up or estimating table values
scalar | vector | matrix

Output generated 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

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Table and Breakpoints

Number of table dimensions — Number of lookup table dimensions
2 (default) | 1 | 3 | 4 | ... | 30

Enter the number of dimensions of the lookup table. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

To specify...	Do this...
1, 2, 3, or 4	Select the value from the drop-down list.

To specify...	Do this...
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.

Programmatic Use
Block Parameter: `NumberOfTableDimensions`
Type: character vector
Values: `'1' | '2' | '3' | '4' | ... | 30`
Default: `'2'`

`Data specification` — Method of table and breakpoint specification

`Table and breakpoints` (default) | `Lookup table object`

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**
  - **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.

Programmatic Use
Block Parameter: `DataSpecification`
Type: character vector
Values: `'Table and breakpoints' | 'Lookup table object'`
Default: `'Table and breakpoints'`

Name — Name of the lookup table object

`[]` (default) | `Simulink.LookupTable` object

Enter the name of the lookup table (`Simulink.LookupTable`) object.

Dependencies

To enable this parameter, set **Data specification** to `Lookup table object`.

Programmatic Use

Block Parameter: `LookupTableObject`
Type: character vector
Values: name of a `Simulink.LookupTable` object
Default: `''`

`Table data` — Define the table of output values

`[4 5 6; 16 19 20; 10 18 23]` (default) | matrix of values

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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `Table`
Type: character vector
Values: matrix of table values
Default: `'[4 5 6; 16 19 20; 10 18 23]'`

`Breakpoints specification` — Method of breakpoint specification

`Explicit values` (default) | `Even spacing`

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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `BreakpointsSpecification`
Type: character vector
Values: `'Explicit values' | 'Even spacing'`
Default: `'Explicit values'`

`Breakpoints` — Explicit breakpoint values, or first point and spacing of breakpoints

`[1:3]` (default) | 1-by-n or n-by-1 vector of monotonically increasing values

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 `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.
- 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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1`
Type: character vector
Values: 1-by-n or n-by-1 vector of monotonically increasing values
Default: `'[1:3]'`

`First point` — First point in evenly spaced breakpoint data

`1` (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite, scalar. This parameter is available when **Breakpoints specification** is set to `Even spacing`.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`, and **Breakpoints specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1FirstPoint | BreakpointsForDimension2FirstPoint`
Type: character vector
Values: real-valued, finite, scalar
Default: `'1'`

`Spacing` — Spacing between evenly spaced breakpoints

1 (default) | scalar

Specify the spacing between points in your evenly-spaced breakpoint data.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`, and **Breakpoints specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1Spacing | BreakpointsForDimension2Spacing`
Type: character vector
Values: positive, real-valued, finite, scalar
Default: `'1'`

`Edit table and breakpoints` — Launch Lookup Table Editor dialog box

button

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.

Algorithm

Lookup method

`Interpolation method` — Method of interpolation between breakpoint values

`Linear point-slope` (default) | `Flat` | `Nearest` | `Linear Lagrange` | `Cubic spline`

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. For more information on interpolation methods, see “Interpolation Methods”.

Dependencies

If you select `Cubic spline`, the block supports only scalar signals. The other interpolation methods support nonscalar signals.

Programmatic Use
Block Parameter: `InterpMethod`
Type: character vector
Values: `'Linear point-slope' | 'Flat' | 'Nearest' | 'Linear Lagrange' | 'Cubic spline'`
Default: `'Linear point-slope'`

`Extrapolation method` — Method of handling input values that fall outside the range of a breakpoint data set

`Clip` (default) | `Linear` | `Cubic spline`

Select `Clip`, `Linear`, or `Cubic spline`. See “Extrapolation Methods” for more information.

If the extrapolation method is `Linear`, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is Linear Lagrange, the extrapolation method inherits the Linear Lagrange equation to compute the extrapolated value.

Dependencies

To select `Cubic spline` for **Extrapolation method**, you must also select `Cubic spline` for **Interpolation method**.

Programmatic Use
Block Parameter: `ExtrapMethod`

Type: character vector
Values: `'Linear' | 'Clip' | 'Cubic spline'`
Default: `'Linear'`

`Index search method` — Method of calculating table indices

`Evenly spaced points` (default) | `Linear search` | `Binary search`

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`.

Programmatic Use
Block Parameter: `IndexSearchMethod`
Type: character vector
Values: `'Binary search' | 'Evenly spaced points' | 'Linear search'`
Default: `'Binary search'`

`Begin index search using previous index result` — Start using the index from the previous time step

`off` (default) | `on`

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.

Dependencies

To enable this parameter, set **Index search method** to `Linear search` or `Binary search`.

Programmatic Use
Block Parameter: `BeginIndexSearchUsing PreviousIndexResult`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Diagnostic for out-of-range input` — Block action when input is out of range

`None` (default) | `Warning` | `Error`

Specify whether to produce a warning or error when the input is out of range. Options include:

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `DiagnosticForOutOfRangeInput`
Type: character vector
Values: `'None' | 'Warning' | 'Error'`
Default: `'None'`

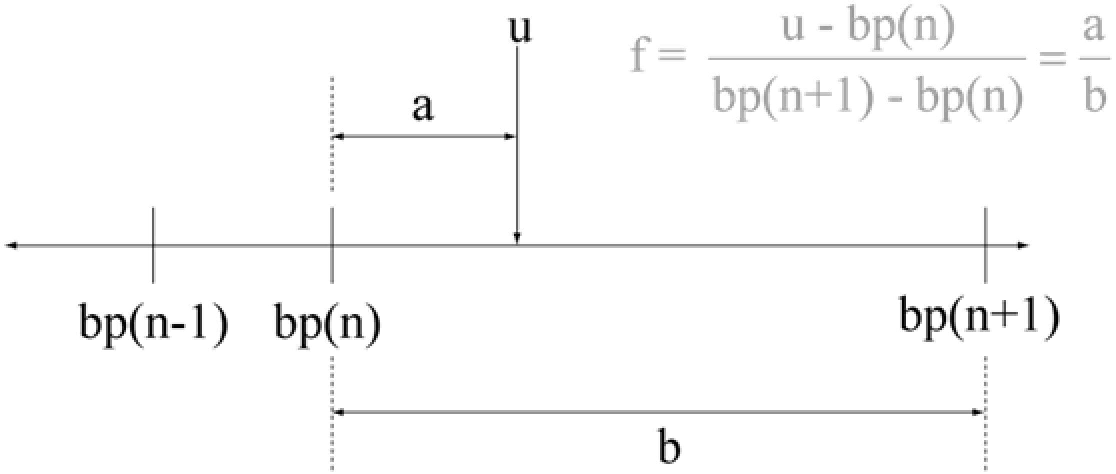
`Use last table value for inputs at or above last breakpoint` — Method for computing output for inputs at or above last breakpoint

`off` (default) | `on`

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.

Check Box	Block Uses Index Of The...	Interval Fraction
Selected	Last element of breakpoint data on the **Table and Breakpoints** tab	0
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 *f* 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 check box, the index is that of the next-to-last element (4) and the interval fraction is 1.

Dependencies

To enable this parameter, set:

- **Interpolation method** to `Linear`.
- **Extrapolation method** to `Clip`.

Programmatic Use

Block Parameter: `UseLastTableValue`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Input settings

`Use one input port for all input data` — Use only one input port

`off` (default) | `on`

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.

Programmatic Use

Block Parameter: `UseOneInputPortForAllInputData`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Code generation

`Remove protection against out-of-range input in generated code` — Remove code that checks for out-of-range input values

`off` (default) | `on`

Specify whether or not to include code that checks for out-of-range input values.

Check Box	Result	When to Use
`on`	Generated code does not include conditional statements to check for out-of-range breakpoint inputs. When the input is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.	For code efficiency
`off`	Generated code includes conditional statements to check for out-of-range 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”.

Programmatic Use
Block Parameter: `RemoveProtectionInput`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Support tunable table size in code generation` — Enable tunable table size in the generated code

`off` (default) | `on`

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.

Dependencies

If you set **Interpolation method** to `Cubic spline`, this check box is not available.

Programmatic Use
Block Parameter: `SupportTunableTableSize`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

`Maximum indices for each dimension` — Maximum index value for each table dimension

`[]` (default) | `scalar or vector of positive integer values`

Example: `[4 6]` for a 5-by-7 table

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`

Examples of valid specifications include:

- `[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 1-1130.

Dependencies

To enable this parameter, 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.

Programmatic Use
Block Parameter: `MaximumIndicesForEachDimension`
Type: character vector
Values: scalar or vector of positive integer values
Default: `'[]'`

Data Types

`Table data` — Data type of table data

`Inherit: Same as output` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

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

Programmatic Use

Block Parameter: `TableDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`
Default: `'Inherit: Same as output'`

Table data Minimum — Minimum value of the table data

[] | scalar

Specify the minimum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMin
Type: character vector
Values: scalar
Default: '[]'

Table data Maximum — Maximum value of the table data

[] | scalar

Specify the maximum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMax
Type: character vector
Values: scalar
Default: '[]'

Breakpoints — Breakpoint data type

Inherit: Same as corresponding input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

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 class, for example, an enumerated data type class
- 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)

Tip

- Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.

- If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
- For enumerated data, **Extrapolation method** must be `Clip`.
- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

This is a limitation for using enumerated data with this block:

- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

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

---

Programmatic Use
Block Parameter: `BreakpointsForDimension1DataTypeStr | BreakpointsForDimension2DataTypeStr| ... | BreakpointsForDimension30DataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'
Default: 'Inherit: Same as corresponding input'

Breakpoints Minimum — Minimum value breakpoint data can have

[] | scalar

Specify the minimum value that a set of breakpoint data can have. The default value is [] (unspecified).

Programmatic Use
Block Parameter: BreakpointsForDimension1Min | BreakpointsForDimension2Min | ... | BreakpointsForDimension30Min
Type: character vector
Values: scalar
Default: '[]'

Breakpoints Maximum — Maximum value breakpoint data can have

[] | scalar

Specify the maximum value that a set of breakpoint data can have. The default value is [] (unspecified).

Programmatic Use
Block Parameter: BreakpointsForDimension1Max | BreakpointsForDimension2Max | ... | BreakpointsForDimension30Max
Type: character vector
Values: scalar
Default: '[]'

Fraction — Fraction data type

Inherit: Inherit via internal rule (default) | double | single | fixdt(1,16,0) | <data type expression>

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

Programmatic Use
Block Parameter: `FractionDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'double' | 'single' | 'fixdt(1,16,0)'|'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Intermediate results` — Intermediate results data type

`Inherit: Same as output` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

Tip Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.

Programmatic Use
Block Parameter: `IntermediateResultsDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`

Default: `'Inherit: Same as output'`

`Output` — Output data type

`Inherit: Same as input` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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 data type attributes. For more information, see "Specify Data Types Using Data Type Assistant".

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression'`
Default: `'Inherit: Same as first input'`

`Output Minimum` — Minimum value the block can output

`[]` | `scalar`

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Output Maximum` — Maximum value the block can output

`[]` | `scalar`

Specify the maximum value that the block can 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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Internal rule priority` — Internal rule for intermediate calculations

`Speed` (default) | `Precision`

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.

Programmatic Use
Block Parameter: `InternalRulePriority`
Type: character vector

Values: `'Speed' | 'Precision'`
Default: `'Speed'`

`Require all inputs to have the same data type` — Require all inputs to have the same data type

`on` (default) | `off`

Select to require all inputs to have the same data type.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Simplest` (default) | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Zero`

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).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Simplest'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box (`on`).	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.
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`

Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	double | single | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Definitions

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';
```

- These block parameters apply in the n-D Lookup Table block dialog box.

Parameter	Value
Number of table dimensions	2
Table data	`p.Table`
Breakpoints 1	`p.BP1`
Breakpoints 2	`p.BP2`
Support tunable table size in code generation	`on`
Maximum indices for each dimension	`p.MaxIdx`

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.

Enumerated Values in Lookup Tables

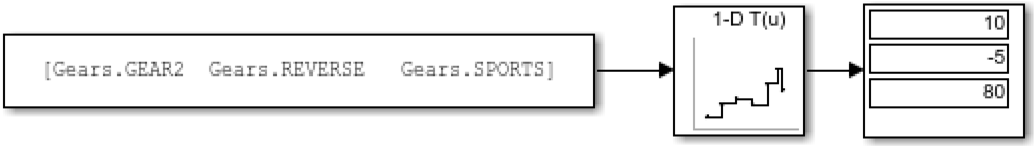
Suppose that you have a lookup table with an enumerated class like this defined:

```
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
        NEUTRAL(0)
    end
end
```

n-D Lookup block has these settings:

- **Number of dimensions** to 1.
- **Table data** value is `[5 10 20 40 80 -5 0]`.
- **Breakpoints 1** value is `enumeration('Gears')`.
- Interpolation method is Flat.
- For an unordered search, set **Index search method** to `Linear search` and clear the **Begin index search using previous index result** check box.

Simulation produces a vector `[10 -5 80]`, which correspond to `GEAR2`, `REVERSE`, and `SPORTS`.



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™.

The 1-D, 2-D, and N-D Lookup Table blocks have restrictions for HDL code generation. For more information, see “Restrictions” (HDL Coder).

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Direct Lookup Table (n-D) | Interpolation Using Prelookup | Lookup Table Dynamic | Prelookup | `Simulink.Breakpoint` | `Simulink.LookupTable`

Topics

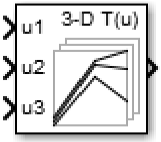
"Import Lookup Table Data from MATLAB"
"About Lookup Table Blocks"
"Anatomy of a Lookup Table"
"Enter Breakpoints and Table Data"
"Guidelines for Choosing a Lookup Table"

Introduced in R2011a

n-D Lookup Table

Approximate n-dimensional function

Library: Simulink / Lookup Tables



Description

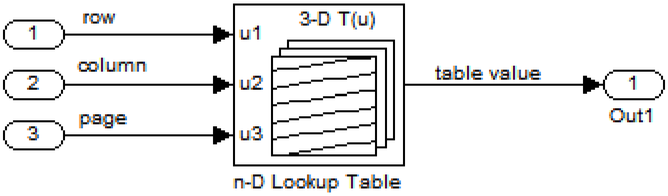
Supported Block Operations

The 1-D, 2-D, and n-D Lookup Table blocks evaluate a sampled representation of a function in N variables

$$y = F(x_1, x_2, x_3, \ldots, x_N)$$

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 (Linear point-slope), Lagrange (Linear Lagrange), nearest, 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” for a description of the port order for various block orientations.

When the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter is set, the 2-D and n-D Lookup Table block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms may differ in the order of the output calculations, possibly resulting in slightly different numerical values. This capability requires a Simulink Coder or Embedded Coder license. For more information on row-major support, see "Code Generation of Matrices and Arrays" (Simulink Coder).

Specification of Breakpoint and Table Data

These 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` and "Identify questionable fixed-point operations" (Embedded Coder).

How the Block Generates Output

The n-D, 1-D and 2-D Lookup Table blocks generate 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

When block inputs...	The n-D Lookup Table block...
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 = table(k)`, interpolation is unnecessary.

Ports

Input

u1 — First-dimension (row) inputs
scalar | vector | matrix

Real-valued inputs to the **u1** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: `0:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

uN — n-th dimension input values
scalar | vector | matrix

Real-valued inputs to the **uN** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: `0:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

Port_1 — Output computed by looking up or estimating table values
scalar | vector | matrix

Output generated 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

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Table and Breakpoints

Number of table dimensions — Number of lookup table dimensions
3 (default) | 1 | 2 | 4 | ... | 30

Enter the number of dimensions of the lookup table. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

To specify...	Do this...
1, 2, 3, or 4	Select the value from the drop-down list.

To specify...	Do this...
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.

Programmatic Use
Block Parameter: `NumberOfTableDimensions`
Type: character vector
Values: `'1' | '2' | '3' | '4' | ... | 30`
Default: `'3'`

`Data specification` — Method of table and breakpoint specification

`Table and breakpoints` (default) | `Lookup table object`

From the list, select:

- `Table and breakpoints` — Specify the table data and breakpoints. Selecting this option enables the following 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.

Programmatic Use
Block Parameter: `DataSpecification`
Type: character vector
Values: `'Table and breakpoints' | 'Lookup table object'`
Default: `'Table and breakpoints'`

`Name` — Name of the lookup table object

`[]` (default) | `Simulink.LookupTable` object

Enter the name of the lookup table (`Simulink.LookupTable`) object.

Dependencies

To enable this parameter, set **Data specification** to `Lookup table object`.

Programmatic Use
Block Parameter: `LookupTableObject`
Type: character vector
Values: name of a `Simulink.LookupTable` object
Default: `''`

`Table data` — Define the table of output values

`reshape(repmat([4 5 6;16 19 20;10 18 23],1,2),[3,3,2])` (default) | matrix of values with dimensions that match the **Number of table dimensions**

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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use
Block Parameter: `Table`
Type: character vector
Values: matrix of table values
Default: `'reshape(repmat([4 5 6;16 19 20;10 18 23],1,2),[3,3,2])'`

`Breakpoints specification` — Method of breakpoint specification

`Explicit values` (default) | `Even spacing`

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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `BreakpointsSpecification`
Type: character vector
Values: `'Explicit values' | 'Even spacing'`
Default: `'Explicit values'`

`Breakpoints` — Explicit breakpoint values, or first point and spacing of breakpoints

`[10,22,31]` (default) | 1-by-n or n-by-1 vector of monotonically increasing values

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 `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.
- 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.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1 | BreakpointsForDimension2 | ... | BreakpointsForDimension30 |`
Type: character vector
Values: 1-by-n or n-by-1 vector of monotonically increasing values
Default: `'[10, 22, 31]'`

`First point` — First point in evenly spaced breakpoint data

`1` (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite, scalar. This parameter is available when **Breakpoints specification** is set to `Even spacing`.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`, and **Breakpoints specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1FirstPoint | BreakpointsForDimension2FirstPoint | ... | BreakpointsForDimension30FirstPoint |`
Type: character vector
Values: real-valued, finite, scalar
Default: `'1'`

Spacing — Spacing between evenly spaced breakpoints

1 (default) | scalar

Specify the spacing between points in your evenly-spaced breakpoint data.

Dependencies

To enable this parameter, set **Data specification** to `Table and breakpoints`, and **Breakpoints specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsForDimension1Spacing | BreakpointsForDimension2Spacing | ... | BreakpointsForDimension30Spacing |`
Type: character vector
Values: positive, real-valued, finite, scalar
Default: `'1'`

`Edit table and breakpoints` — Launch Lookup Table Editor dialog box

button

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.

Algorithm

Lookup method

`Interpolation method` — Method of interpolation between breakpoint values

`Linear point-slope` (default) | `Flat` | `Nearest` | `Linear Lagrange` | `Cubic spline`

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. For more information on interpolation methods, see “Interpolation Methods”.

Dependencies

If you select `Cubic spline`, the block supports only scalar signals. The other interpolation methods support nonscalar signals.

Programmatic Use
Block Parameter: `InterpMethod`
Type: character vector
Values: `'Linear point-slope' | 'Flat' | 'Nearest' | 'Linear Lagrange' | 'Cubic spline'`
Default: `'Linear point-slope'`

`Extrapolation method` — Method of handling input values that fall outside the range of a breakpoint data set

`Clip` (default) | `Linear` | `Cubic spline`

Select `Clip`, `Linear`, or `Cubic spline`. See “Extrapolation Methods” for more information.

If the extrapolation method is `Linear`, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is Linear Lagrange, the extrapolation method inherits the Linear Lagrange equation to compute the extrapolated value.

Dependencies

To select `Cubic spline` for **Extrapolation method**, you must also select `Cubic spline` for **Interpolation method**.

Programmatic Use
Block Parameter: `ExtrapMethod`

Type: character vector
Values: `'Linear' | 'Clip' | 'Cubic spline'`
Default: `'Linear'`

`Index search method` — Method of calculating table indices

`Evenly spaced points` (default) | `Linear search` | `Binary search`

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`.

Programmatic Use
Block Parameter: `IndexSearchMethod`
Type: character vector
Values: `'Binary search' | 'Evenly spaced points' | 'Linear search'`
Default: `'Binary search'`

`Begin index search using previous index result` — Start using the index from the previous time step

`off` (default) | `on`

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.

Dependencies

To enable this parameter, set **Index search method** to `Linear search` or `Binary search`.

Programmatic Use
Block Parameter: `BeginIndexSearchUsing PreviousIndexResult`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Diagnostic for out-of-range input` — Block action when input is out of range

`None` (default) | `Warning` | `Error`

Specify whether to produce a warning or error when the input is out of range. Options include:

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `DiagnosticForOutOfRangeInput`
Type: character vector
Values: `'None' | 'Warning' | 'Error'`
Default: `'None'`

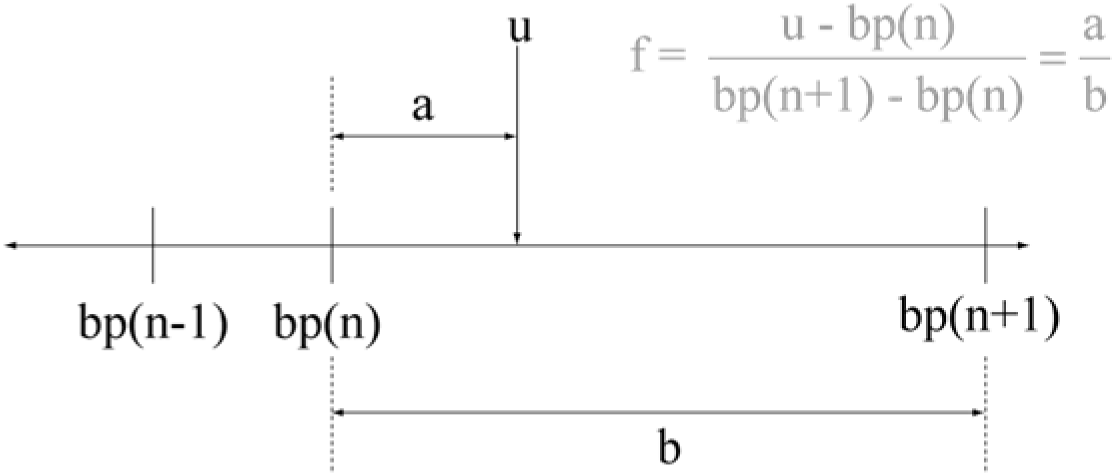
`Use last table value for inputs at or above last breakpoint` — Method for computing output for inputs at or above last breakpoint

`off` (default) | `on`

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.

Check Box	Block Uses Index Of The...	Interval Fraction
Selected	Last element of breakpoint data on the **Table and Breakpoints** tab	0
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 *f* 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 check box, the index is that of the next-to-last element (4) and the interval fraction is 1.

Dependencies

To enable this parameter, set:

- **Interpolation method** to `Linear`.
- **Extrapolation method** to `Clip`.

Programmatic Use

Block Parameter: `UseLastTableValue`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Input settings

`Use one input port for all input data` — Use only one input port

`off` (default) | `on`

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.

Programmatic Use

Block Parameter: `UseOneInputPortForAllInputData`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Code generation

`Remove protection against out-of-range input in generated code` — Remove code that checks for out-of-range input values

`off` (default) | `on`

Specify whether or not to include code that checks for out-of-range input values.

Check Box	Result	When to Use
`on`	Generated code does not include conditional statements to check for out-of-range breakpoint inputs. When the input is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.	For code efficiency
`off`	Generated code includes conditional statements to check for out-of-range 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”.

Programmatic Use
Block Parameter: `RemoveProtectionInput`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Support tunable table size in code generation` — Enable tunable table size in the generated code

`off` (default) | `on`

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.

Dependencies

If you set **Interpolation method** to `Cubic spline`, this check box is not available.

Programmatic Use
Block Parameter: `SupportTunableTableSize`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

`Maximum indices for each dimension` — Maximum index value for each table dimension

`[]` (default) | `scalar or vector of positive integer values`

Example: `[4 6]` for a 5-by-7 table

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`

Examples of valid specifications include:

- `[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 1-1160.

Dependencies

To enable this parameter, 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.

Programmatic Use
Block Parameter: `MaximumIndicesForEachDimension`
Type: character vector
Values: scalar or vector of positive integer values
Default: `'[]'`

Data Types

`Table data` — Data type of table data

`Inherit: Same as output` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

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

Programmatic Use

Block Parameter: `TableDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`
Default: `'Inherit: Same as output'`

Table data Minimum — Minimum value of the table data

[] | scalar

Specify the minimum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMin
Type: character vector
Values: scalar
Default: '[]'

Table data Maximum — Maximum value of the table data

[] | scalar

Specify the maximum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMax
Type: character vector
Values: scalar
Default: '[]'

Breakpoints — Breakpoint data type

Inherit: Same as corresponding input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

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 class, for example, an enumerated data type class
- 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)

Tip

- Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.

- If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
- For enumerated data, **Extrapolation method** must be `Clip`.
- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

This is a limitation for using enumerated data with this block:

- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

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

---

Programmatic Use
Block Parameter: `BreakpointsForDimension1DataTypeStr | BreakpointsForDimension2DataTypeStr| ... | BreakpointsForDimension30DataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`
Default: `'Inherit: Same as corresponding input'`

`Breakpoints Minimum` — Minimum value breakpoint data can have

`[]` | `scalar`

Specify the minimum value that a set of breakpoint data can have. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `BreakpointsForDimension1Min | BreakpointsForDimension2Min | ... | BreakpointsForDimension30Min`
Type: character vector
Values: scalar
Default: `'[]'`

`Breakpoints Maximum` — Maximum value breakpoint data can have

`[]` | `scalar`

Specify the maximum value that a set of breakpoint data can have. The default value is `[]` (unspecified).

Programmatic Use
Block Parameter: `BreakpointsForDimension1Max | BreakpointsForDimension2Max | ... | BreakpointsForDimension30Max`
Type: character vector
Values: scalar
Default: `'[]'`

`Fraction` — Fraction data type

`Inherit: Inherit via internal rule` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

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

Programmatic Use
Block Parameter: `FractionDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule' | 'double' | 'single' | 'fixdt(1,16,0)'|'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Intermediate results` — Intermediate results data type

`Inherit: Same as output` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

Tip Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.

Programmatic Use
Block Parameter: `IntermediateResultsDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`

Default: 'Inherit: Same as output'

Output — Output data type

Inherit: Same as input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

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

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: '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)'|'<data type expression'
Default: 'Inherit: Same as first input'

Output Minimum — Minimum value the block can output

[] | scalar

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[]'`

`Output Maximum` — Maximum value the block can output

`[]` | `scalar`

Specify the maximum value that the block can 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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[]'`

`Internal rule priority` — Internal rule for intermediate calculations

`Speed` (default) | `Precision`

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.

Programmatic Use
Block Parameter: `InternalRulePriority`
Type: character vector

Values: `'Speed' | 'Precision'`
Default: `'Speed'`

`Require all inputs to have the same data type` — Require all inputs to have the same data type

`on` (default) | `off`

Select to require all inputs to have the same data type.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Simplest` (default) | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Zero`

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).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Simplest'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box (`on`).	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.
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`

Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

Data Types	double | single | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Definitions

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';
```

- These block parameters apply in the n-D Lookup Table block dialog box.

Parameter	Value
Number of table dimensions	`2`
Table data	`p.Table`
Breakpoints 1	`p.BP1`
Breakpoints 2	`p.BP2`
Support tunable table size in code generation	`on`
Maximum indices for each dimension	`p.MaxIdx`

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.

Enumerated Values in Lookup Tables

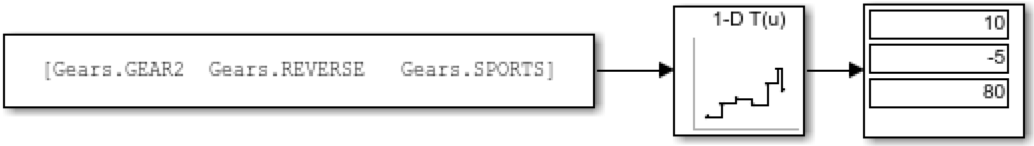
Suppose that you have a lookup table with an enumerated class like this defined:

```
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
        NEUTRAL(0)
    end
end
```

n-D Lookup block has these settings:

- **Number of dimensions** to 1.
- **Table data** value is `[5 10 20 40 80 -5 0]`.
- **Breakpoints 1** value is `enumeration('Gears')`.
- Interpolation method is Flat.
- For an unordered search, set **Index search method** to `Linear search` and clear the **Begin index search using previous index result** check box.

Simulation produces a vector `[10 -5 80]`, which correspond to `GEAR2`, `REVERSE`, and `SPORTS`.



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™.

For more information about HDL Code Generation, see n-D Lookup Table.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Simulink PLC Coder™ has limited support for lookup table blocks. The coder does not support:

- Number of dimensions greater than 2
- Cubic spline interpolation method
- Begin index search using a previous index mode
- Cubic spline extrapolation method

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Direct Lookup Table (n-D) | Interpolation Using Prelookup | Lookup Table Dynamic | Prelookup | `Simulink.Breakpoint` | `Simulink.LookupTable`

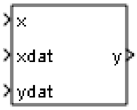
Topics

"Import Lookup Table Data from MATLAB"
"About Lookup Table Blocks"
"Anatomy of a Lookup Table"
"Enter Breakpoints and Table Data"
"Guidelines for Choosing a Lookup Table"
"Interpolation Algorithm for Row-Major Array Layout" (Simulink Coder)

Introduced in R2011a

Lookup Table Dynamic

Approximate a one-dimensional function using dynamic table
Library: Simulink / 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`	Performs linear interpolation and extrapolation of the inputs. • 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. **Note** If you select this lookup method, Simulink Coder software cannot generate code for this block.

Lookup Method	Block Action
`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 `xdat` nearest the current input. The corresponding element in `ydat` is the output.
`Use Input Below`	Finds the element in `xdat` nearest and below the current input. The corresponding element in `ydat` is the output. If there is no element in `xdat` below the current input, the block finds the nearest element.
`Use Input Above`	Finds the element in `xdat` nearest and above the current input. The corresponding element in `ydat` is the output. If there is no element in `xdat` 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.

Ports

Input

x — input vector

`n-dimensional array`

The block accepts real-valued or complex-valued multidimensional inputs.

Example: `2:12`

Dependencies

The `x` input vector and the `xdat` breakpoint data 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`.

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

`xdat` — breakpoint data

`1-by-n vector of strictly monotonically increasing values`

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.

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.

Example: `1:10`

Dependencies

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`.

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

`ydat` — table data

`1-by-n vector`

The `ydat` input is a 1-by-n vector of real-valued or complex-valued table data, which is an evaluation of the function at the breakpoint values.

Example: `[0 3 12 27 48 75 108 147 192 243 300]`

Dependencies

The ydat table data and the y output vector must have the same sign, bias, and fractional slope.

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

Output

y — Approximation of y = f(x) using dynamic table data

1-by-n vector

The block computes an approximation of a function y = f(x) using the xdat and ydat input vectors. The lookup method can use interpolation, extrapolation, or the original values of the input.

Dependencies

The ydat table data and the y output vector must have the same sign, bias, and fractional slope.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Main Tab

Lookup Method — Specify the lookup method

Interpolation-Use End Values (default) | Interpolation-Extrapolation | Use Input Nearest | Use Input Below | Use Input Above

The block computes output by applying the **Lookup Method** you select to the input vectors of breakpoint data (xdat) and table data (ydat). For details, see “How the Block Generates Output” on page 1-1166.

Programmatic Use

Block Parameter: LookUpMeth
Type: character vector

Values: `'Interpolation-Extrapolation' | 'Interpolation-Use End Values' | 'Use Input Nearest' | 'Use Input Below' | 'Use Input Above'`
Default: `'Interpolation-Use End Values'`

Signal Attributes Tab

`Output data type` — Output data type

`double` (default) | `'Inherit: Inherit via back propagation'` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Specify the data type of the output signal `y`.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Dependencies

The `ydat` table data and the `y` output vector must have the same sign, bias, and fractional slope.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'|<data type expression>`
Default: `'double'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`

Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations. For more information, see "Rounding" (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

The Simulink PLC Coder software does not support the SimulinkSimulink Lookup Table Dynamic block. For your convenience, the plclib/Simulink/Lookup Tables library contains an implementation of a dynamic table lookup block using the Prelookup and Interpolation Using Prelookup blocks.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

n-D Lookup Table

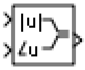
Topics

“Entering Data Using Inports of the Lookup Table Dynamic Block”
“Nonlinearity”
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Guidelines for Choosing a Lookup Table”

Introduced before R2006a

Magnitude-Angle to Complex

Convert magnitude and/or a phase angle signal to complex signal
Library: Simulink / 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 rad.

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 1-1181), the block input for phase angle has the following restrictions:

- For signed fixed-point types, the input angle must fall within the range [–2π, 2π) rad.
- For unsigned fixed-point types, the input angle must fall within the range [0, 2π) rad.

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.

Ports

Input

|u| — Magnitude

scalar | vector | matrix

Magnitude, specified as a real-valued scalar, vector, or matrix.

Dependencies

- To enable this port, set **Input** to `Magnitude and angle`.

Limitations

- If one input has a floating-point data type, the other input must use the same data type. For example, both signals must be `double` or `single`.
- Fixed-point data types are supported only when you set the **Approximation method** to `CORDIC`. When one input has a fixed-point data type, the other input must also have a fixed-point data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

∠u — Radian phase angle

scalar | vector | matrix

Radian phase angle, specified as a real-valued scalar, vector, or matrix. To compute the CORDIC approximation, the input angle must be between:

- [–2π, 2π) rad, for signed fixed-point types
- [0, 2π) rad, for unsigned fixed-point types

For more information, see “Effect of Out-of-Range Input on CORDIC Approximations” on page 1-1174.

Dependencies

- To enable this port, set **Input** to `Magnitude and angle`.

Limitations

- If one input has a floating-point data type, the other input must use the same data type. For example, both signals must be `double` or `single`.
- Fixed-point data types are supported only when you set the **Approximation method** to `CORDIC`. If one input has a fixed-point data type, the other input must also have a fixed-point data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_1` — Magnitude or radian phase angle

scalar | vector | matrix

Magnitude, or radian phase angle, specified as a real-valued scalar, vector, or matrix.

- When you set **Input** to `Magnitude`, you specify the magnitude at the input port, and the angle on the dialog box.
- When you set **Input** to `Angle`, you specify the angle at the input port, and the magnitude on the dialog box.

Dependencies

To enable this port, set **Input** to `Magnitude` or `Angle`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Complex signal

scalar | vector | matrix

Complex signal, formed from the magnitude and phase angle you specify.

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.

Data Types: `single` | `double` | `fixed point`

Parameters

`Input` — Type of input

`Magnitude` (default) | `Angle` | `Magnitude and angle`

Specify the kind of input: a magnitude input, an angle input, or both.

Programmatic Use

Block Parameter: `Input`
Type: character vector
Values: `'Magnitude' | 'Angle' | 'Magnitude and angle'`
Default: `'Magnitude and angle'`

`Angle` — Phase angle of output

`0` (default) | real-valued scalar, vector, or matrix

Constant phase angle of the output signal, in rad. To compute the CORDIC approximation, the input angle must be between:

- [–2π, 2π) rad, for signed fixed-point types
- [0, 2π) rad, for unsigned fixed-point types

For more information, see “Effect of Out-of-Range Input on CORDIC Approximations” on page 1-1174.

Dependencies

To enable this parameter, set **Input** to `Magnitude`.

Programmatic Use

Block Parameter: `ConstantPart`

Type: character vector
Values: constant scalar
Default: `'0'`

`Magnitude` — Magnitude of output

`0` (default) | real-valued scalar, vector, or matrix

Constant magnitude of the output signal, specified as a real-valued scalar, vector, or matrix.

Dependencies

To enable this parameter, set **Input** to `Angle`.

Programmatic Use

Block Parameter: `ConstantPart`
Type: character vector
Values: constant scalar
Default: `'0'`

`Approximation method` — CORDIC or none

`None` (default) | `CORDIC`

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 the input angle:

- For signed fixed-point types, the input angle must fall within the range [–2π, 2π) rad.
- For unsigned fixed-point types, the input angle must fall within the range [0, 2π) rad.

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`	`fixdt`(1, *`WL`* + 2, *`FL`*) where *`WL`* and *`FL`* are the word length and fraction length of the magnitude
Unsigned, fixed point	`CORDIC`	`fixdt`(1, *`WL`* + 3, *`FL`*) where *`WL`* and *`FL`* are the word length and fraction length of the magnitude

Programmatic Use
Block Parameter: `ApproximationMethod`
Type: character vector
Values: `'None' | 'CORDIC'`
Default: `'None'`

`Number of iterations` — Number of iterations for CORDIC algorithm

`11` (default) | positive integer, less than or equal to word length of fixed-point input

Number of iterations to perform the CORDIC algorithm. The range of possible values depends on the data type of the input:

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

Dependencies

To enable this parameter, set **Approximation method** to `CORDIC`.

Programmatic Use
Block Parameter: `NumberOfIterations`
Type: character vector
Values: positive integer, less than or equal to word length of fixed-point input

Default: '11'

Scale output by reciprocal of gain factor — Scale real and imaginary parts of complex output

on (default) | off

Select this check box to scale the real and imaginary parts of the complex output by a factor of (1/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.

Dependencies

To enable this parameter, set **Approximation method** to CORDIC.

Programmatic Use

Block Parameter: ScaleReciprocalGainFactor
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Sample time — Specify sample time as a value other than -1

-1 (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use

Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '-1'

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

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.

References

[1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers* EC-8 (1959); 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): 191–200.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference (1971): 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* 90, no. 5 (1983): 317–325.

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™.

For more information about HDL code generation, see Magnitude-Angle to Complex.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

The Magnitude-Angle to Complex block supports fixed-point and base integer data types when you set **Approximation method** to `CORDIC`.

See Also

Complex to Magnitude-Angle | Complex to Real-Imag | Real-Imag to Complex

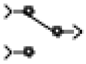
Topics

“Complex Signals”

Introduced before R2006a

Manual Switch

Switch between two inputs
Library: Simulink / 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. You 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.

Note Double-clicking the Manual Switch block does not open the block dialog box. Instead, it toggles the input choice.

Ports

Input

Port_1 — First input signal
scalar | vector

First of two inputs to the Manual Switch block. The block propagates the selected input to the output. To select the input signal, toggle the switch by double-clicking the block.

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

Port_2 — Second input signal
scalar | vector

Second of two inputs to the Manual Switch block. The block propagates the selected input to the output. To select which input signal, toggle the switch by double-clicking the block.

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

Output

`Port_1` — Output signal

scalar | vector

Output signal progagated from either the first or second input signal.

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

Parameters

To view the block parameters, right-click the block and select `Block Parameters (ManualSwitch)`.

`Allow the two inputs to differ in size (Results in variable-size output signal)` — Allow inputs of different sizes

`off` (default) | `on`

Select this check box to allow inputs with different sizes. If you select the box, the block allows inputs with different sizes, and propagates the selected input signal size to the output signal. If you clear the box, the block expands scalar inputs to have the same dimensions as nonscalar inputs. See “Scalar Expansion of Inputs and Parameters”.

Programmatic Use
Parameter: `varsize`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

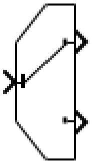
See Also

Manual Variant Sink | Manual Variant Source | Multiport Switch | Switch

Introduced before R2006a

Manual Variant Sink

Switch between multiple variant choices at output
Library: Simulink / 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.

The block can have two or more output ports and has one input port. Each output port is associated with a variant control. To change the number of output ports, right-click the block and select **Mask Parameters**, then type a value in the **Number of choices** box.

To toggle between the variant choices at output, double-click the block. The block displays the active choice with a line connecting the input to the output. The block propagates the active variant choice at output and discards the blocks connected to inactive output ports during simulation.

Note Double-clicking the Manual Variant Sink block does not open the block dialog box instead it toggles the output choice.

Ports

Input

`Port_1` — Input signal
scalar | vector

Input signal passed to the active output port.

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

Output

`Port_1` — First variant output

scalar | vector

First variant output signal. The block passes the input signal to this output port when you connect the toggle switch to this port.

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

`Port_n` — nth variant output

scalar | vector

`n`th variant output signal. The block passes the input signal to this output port when you connect the toggle switch to this port.

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

Parameters

To access the block parameter, right-click the block and select **Mask > Mask Parameters**.

`Number of choices` — Number of output choices

2 (default) | scalar

Specify the number of variant output ports.

Programmatic Use
Block Parameter: `NumChoices`
Type: character vector
Value: `integer`
Default: `'2'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Manual Variant Source | Variant Sink | Variant Source

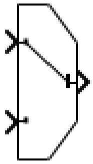
Topics

“Introduction to Variant Controls”
“Define, Configure, and Activate Variants”
“Working with Variant Choices”
“Variant Systems” (Embedded Coder)
“Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
“Variants Example Models”

Introduced in R2016b

Manual Variant Source

Switch between multiple variant choices at input
Library: Simulink / 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 change the number of input ports, right-click the block and select **Mask Parameters**, then type a value in the **Number of choices** box.

To toggle between the variant choices at input, double-click the block. The block displays the active choice with a line connecting the input to the output. The block propagates the active variant choice at input directly to the output and discards the blocks connected to inactive input ports during simulation.

Note Double-clicking the Manual Variant Source block does not open the block dialog box instead it toggles the output choice.

Ports

Input

`Port_1` — First variant input signal
scalar | vector

First variant input signal. The block passes this input signal to the output port when you connect the toggle switch to this port.

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

Port_n — nth variant input signal
scalar | vector

nth variant input signal. The block passes this input signal to the output port when you connect the toggle switch to this port.

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

Output

Port_1 — Output signal
scalar | vector

Output signal passed from the active variant input signal.

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

Parameters

To access the block parameter, right-click the block and select **Mask** > **Mask Parameters**.

Number of choices — Number of input choices
2 (default) | scalar

Specify the number of variant input ports.

Programmatic Use
Block Parameter: NumChoices
Type: character vector
Value: integer
Default: '2'

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Manual Variant Source and Sink Blocks | Variant Sink | Variant Source

Topics

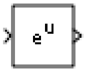
“Introduction to Variant Controls”
“Define, Configure, and Activate Variants”
“Working with Variant Choices”
“Variant Systems” (Embedded Coder)
“Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
“Variants Example Models”

Introduced in R2016b

Math Function

Perform mathematical function

Library: Simulink / 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 these 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)
square	Power 2	u^2	u.^2 (see power)
pow	Power	u^v	power
conj	Complex conjugate	$\bar{u}$	conj

Function	Description	Mathematical Expression	MATLAB Equivalent
`reciprocal`	Reciprocal	$1/u$	`1./u` (see `rdivide`)
`hypot`	Square root of sum squares	$(u^2+v^2)^{0.5}$	`hypot`
`rem`	Remainder after division	—	`rem`
`mod`	Modulus after division	—	`mod`
`transpose`	Transpose	u^T	`u.'` (see “Array vs. Matrix Operations” (MATLAB))
`hermitian`	Complex conjugate transpose	u^H	`u'` (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 these types of operations.

Function	Scalar Operations	Element-Wise Vector and Matrix Operations	Vector and Matrix Operations
`exp`	Yes	Yes	—
`log`	Yes	Yes	—
`10^u`	Yes	Yes	—
`log10`	Yes	Yes	—
`magnitude^2`	Yes	Yes	—
`square`	Yes	Yes	—
`pow`	Yes	Yes	—
`conj`	Yes	Yes	—
`reciprocal`	Yes	Yes	—

Function	Scalar Operations	Element-Wise Vector and Matrix Operations	Vector and Matrix Operations
`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

This 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	—	—	—
`log`	Yes	Yes	—	—	—

Function	single	double	boolean	built-in integer	fixed point
`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

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal specified as a scalar, vector, or matrix. All supported modes accept both real and complex inputs, except for `reciprocal`, which does not accept complex fixed-point inputs. See Description on page 1-1195 for more information.

Dependencies

Data type support for this block depends on the **Function** you select and the size of the input(s). For more information, see “Data Type Support” on page 1-1197.

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

`Port_2` — Input signal

scalar | vector | matrix

Input signal specified as a scalar, vector, or matrix. All supported modes accept both real and complex inputs, except for `reciprocal`, which does not accept complex fixed-point inputs.

Dependencies

To enable this port, set **Function** to `hypot`, `rem`, or `mod`.

Data type support for this block depends on the **Function** you select, and the size of the input(s). For more information, see “Data Type Support” on page 1-1197.

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

Output

`Port_1` — Result of the operation of the function on the input or inputs

scalar | vector | matrix

Output signal specified as a scalar, vector, or matrix. The dimensions of the block output depend on the **Function** you select and the size of the inputs. The block output is real or complex, depending on what you select for **Output signal type**. See Description on page 1-1195 for more information.

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

Parameters

Main

`Function` — Math function

`exp` (default) | `log` | `10^u` | `log10` | `magnitude^2` | `square` | `pow` | `conj` | `reciprocal` | `hypot` | `rem` | `mod` | `transpose` | `hermitian`

Specify the mathematical function. See Description on page 1-1195 for more information about the options for this parameter.

Programmatic Use
Block Parameter: `Operator`
Type: character vector
Values: `'exp' | 'log' | '10^u' | 'log10' | 'magnitude^2' | 'square' | 'pow' | 'conj' | 'reciprocal' | 'hypot' | 'rem' | 'mod' | 'transpose' | 'hermitian'`
Default: `'exp'`

`Output signal type` — Complexity of output signal

`auto` (default) | `real` | `complex`

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` `complex`	`real` `complex`	`real` `error`	`complex` `complex`
`magnitude squared`	`real` `complex`	`real` `real`	`real` `real`	`complex` `complex`
`hypot`, `rem`, `mod`	`real` `complex`	`real` `error`	`real` `error`	`complex` `error`

Programmatic Use
Block Parameter: `OutputSignalType`
Type: character vector
Values: `'auto' | 'real' | 'complex'`
Default: `'auto'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

`Output minimum` — Minimum output value for range checking

`[ ]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Same as first input` (default) | `Inherit: Inherit via internal rule` | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

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

Dependencies

To enable this parameter, set the **Function** to `magnitude^2`, `square`, or `reciprocal`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Same as first input'` | `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Same as first input'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Dependencies

To enable this parameter, set the **Function** to `magnitude^2`, `square`, or `reciprocal`.

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

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.

Dependencies

To enable this parameter, set the **Function** to `magnitude^2`, `square`, or `reciprocal`.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Choose the behavior when integer overflow occurs

`on` (default) | boolean

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 `int8` (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.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Dependencies

To enable this parameter, set the **Function** to `magnitude^2`, `square`, `conj`, `reciprocal`, or `hermitian`.

Programmatic Use

Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
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™.

The Math Function block has HDL code generation restrictions when you set the **Function** to `reciprocal`. For more information, see Math Function.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

The Math Function block only supports fixed-point conversion in certain configurations. For more information, see the Block Support Table.

See Also

Fcn | Sqrt, Signed Sqrt, Reciprocal Sqrt | Trigonometric Function

Topics

“hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)”

Introduced before R2006a

MATLAB Function

Include MATLAB code in models that generate embeddable C code

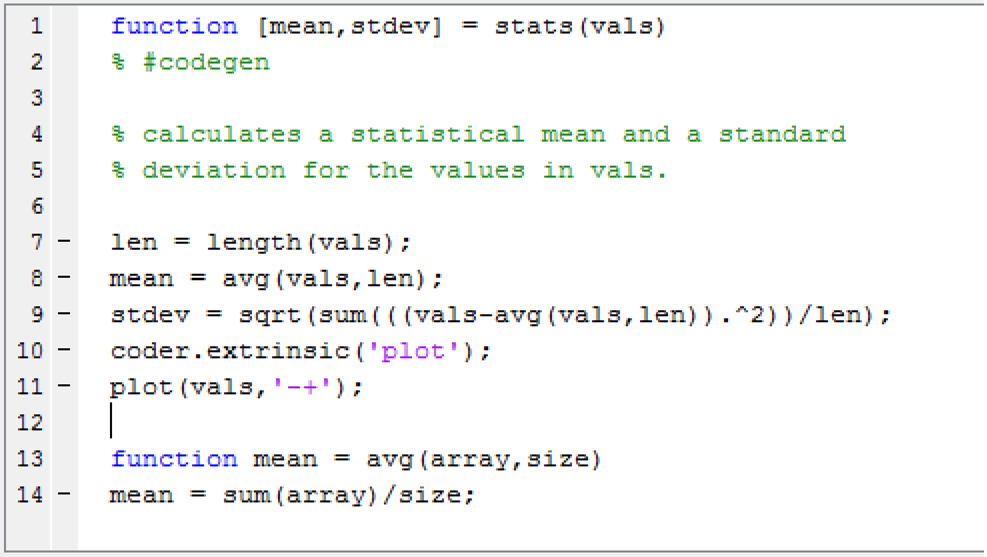
Library: Simulink / User-Defined Functions

u
fcn
y
MATLAB Function

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.

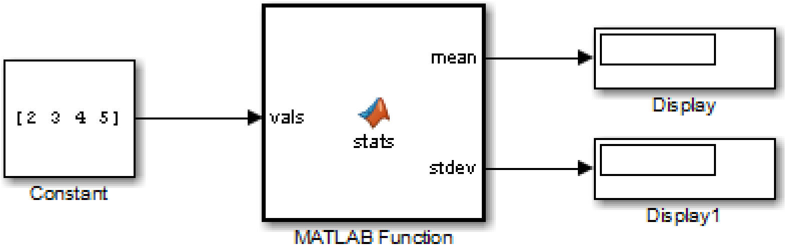
Double-clicking the MATLAB Function block opens its editor, where you write the MATLAB function, as in this example:



```
function [mean,stdev] = stats(vals)
% #codegen

% calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
coder.extrinsic('plot');
plot(vals,'-+');

function mean = avg(array,size)
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.

Ports

Input

u — Input argument u
scalar | vector | matrix

Input corresponding to the first input argument of the function inside the MATLAB Function block. If you rename the function argument in the editor, the block renames the port correspondingly.

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).

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

n — Input argument n
scalar | vector | matrix

`n`th input argument to the function in the MATLAB Function block. When you add the argument in the editor, the block adds the port correspondingly.

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).

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

Output

y — Output argument

scalar | vector | matrix

First output argument of the function inside the MATLAB Function block. If you rename the function argument in the editor, the block renames the port correspondingly.

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

n — Output argument n

scalar | vector | matrix

nth output argument from the function inside the MATLAB Function block. When you add the argument in the editor, the block adds the port correspondingly.

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

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[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™.

Actual data type or capability support depends on block implementation.

For information about HDL Code Generation, see MATLAB Function.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

See Also

Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

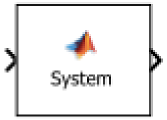
Topics

“Create Model That Uses MATLAB Function Block”
“Use Nondirect Feedthrough in a MATLAB Function Block”
“What Is a MATLAB Function Block?”
“Why Use MATLAB Function Blocks?”

Introduced in R2011a

MATLAB System

Include System object in model
Library: Simulink / 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 “Integrate System Objects Using MATLAB System Block”.

Ports

Input

`In` — Signal input to a MATLAB System block

scalar | vector | matrix

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).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Out` — Signal output of a MATLAB System block

scalar | vector | matrix

Signal output of a MATLAB System block that the System objectreturns.

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

Parameters

`System Object Name` — Name of the System object

cell array (default)

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.

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.

Programmatic Use
Block Parameter: `System`
Type: character vector
Value: name of the System object
Default: `' '`

New — Create a System object from a template

`Basic` (default) | `Advanced` | `Simulink Extension`

Select one of the options for a System object template.

`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.

`Simulate using` — Select the simulation mode

`Code generation` (default) | `Interpreted Execution`

Select the simulation mode.

`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.

Dependency — Dependency parameter for MATLAB System block

auto (default)

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

Off (default) | **On**

☑ 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.

Tips

- 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.

Dependency

This check box appears when you use the `showFiSettingsImpl` method in the System object.

Programmatic Use

Block Parameter:`SaturateOnIntegerOverflow`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Treat these inherited Simulink signal types as fi objects — Specify fi data types

`Fixed-point` (default) | `Fixed-point & Integer`

Select which inherited data types to treat fi data types,

`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.

Dependency

This check box appears when you use the `showFiSettingsImpl` method in the System object.

`MATLAB System fimath` — Specify fixed-point settings to use

Same as MATLAB (default) | **Specify Other**

Select which fixed-point math settings to use.

Same as MATLAB

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`.

Dependency

This check box appears when you use the `showFiSettingsImpl` method in the System object.

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[b][a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`Yes`[c][a]
Zero-Crossing Detection	`No`

a. Actual data type or capability support depends on block implementation.

b. See Nonvirtual Buses and MATLAB System Block for more information.

c. See Variable-Size Signals for more information.

Extended Capabilities

C/C++ Code Generation

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

Actual code generation support depends on block implementation.

HDL Code Generation

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

For more information about HDL code generation, see MATLAB System.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

MATLAB Function

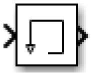
Topics

"System Identification for an FIR System Using MATLAB System Blocks"
"MATLAB System Block with Variable-Size Input and Output Signals"
"Using Buses with MATLAB System Blocks"
"Pulse Width modulation Using MATLAB System block"
"MATLAB System Block"
"What Are System Objects?" (MATLAB)

Introduced in R2013b

Memory

Output input from previous time step
Library: Simulink / 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 a scalar, vector, matrix, or N-D array. If the input is non-scalar, the block holds and delays all elements of the input 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

The Memory, Unit Delay, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different.

This table shows 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.	• `sldemo_enginewc` (Compression subsystem)
Memory on page 1-1222	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.	• `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.	• `sldemo_radar_eml` • `aero_dap3dof`

Each block has the following capabilities.

Capability	Memory	Unit Delay	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	No, because the block can only inherit sample time from the driving block or the solver used for the entire model.	Yes	Yes
Support for frame-based signals	No	Yes	Yes
Support for state logging	No	Yes	No

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 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”.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. The input can be continuous or discrete, containing real, or complex values of any data type Simulink supports.

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

Output

`Port_1` — Input delayed by one major integration time step

scalar | vector | matrix | N-D array

Output is the input from the previous time step.

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

Parameters

Main

`Initial condition` — Initial condition

0 (default) | scalar | vector | matrix | N-D array

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`.

Programmatic Use
Block Parameter: `InitialCondition`
Type: character vector

Values: scalar | vector
Default: '0'

Inherit sample time — Inherit sample time

off (default) | on

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 check box 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.

Programmatic Use
Block Parameter: InheritSampleTime
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Direct feedthrough of input during linearization — Output the input during linearization and trim

off (default) | on

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 this 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.

Programmatic Use
Block Parameter: `LinearizeMemory`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Treat as a unit delay when linearizing with discrete sample time` — Linearize to unit delay for discrete inputs

`off` (default) | `on`

Select to linearize the Memory block to a unit delay when the Memory block is driven by a signal with a discrete sample time.

Programmatic Use
Block Parameter: `LinearizeAsDelay`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: `StateName`
Type: character vector
Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, specify a value for **State name**. 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**.

Programmatic Use
Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default) | `<StorageClass.PackageName>`

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use

Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`
Default: `'Simulink.Signal'`

`Code generation storage class` — State storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `BitField (Custom)` | `Model default` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `AutoScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation.

- `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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

To enable this parameter, specify a value for **State name**.

Programmatic Use

Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto' | 'SimulinkGlobal' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

`TypeQualifier` — Storage type qualifier

`''` (default) | `const` | `volatile` | ...

Specify a storage type qualifier such as `const` or `volatile`.

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.

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 “Data Objects”.

Dependencies

To enable this parameter, set **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `Model default`. This parameter is hidden unless you previously set its value.

Programmatic Use

Block Parameter: `RTWStateStorageTypeQualifier`
Type: character vector
Values: `'' | 'const' | 'volatile' | ...`
Default: `''`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
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™.

For more information about HDL code generation, see Memory.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Unit Delay | Zero-Order Hold

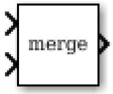
Topics

"Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder)
"Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder)
"Data Objects"

Introduced before R2006a

Merge

Combine multiple signals into single signal
Library: Simulink / Signal Routing



Description

The Merge block combines inputs into a single output. The output value at any time is equal to the most recently computed output of its driving blocks. Specify the number of inputs by setting the parameter **Number of inputs** parameter.

Use Merge blocks 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.

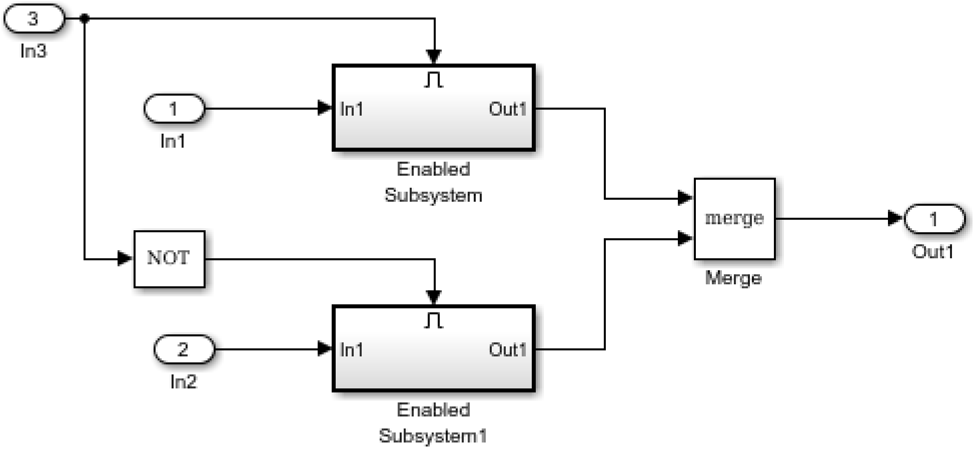
Guidelines for Using the Merge Block

When you use the Merge block, follow these guidelines:

- Always use conditionally executed subsystems to drive Merge blocks.
- Ensure that at most one of the driving conditionally executed subsystems executes at any time step.
- Ensure that all input signals have the same sample time.
- 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.
- For all conditionally executed subsystem Outport 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.

For each input of a Merge block, the topmost nonatomic and nonvirtual source must be a conditionally executed subsystem that is not an Iterator Subsystem.

The next diagram shows valid Merge block usage, merging signals from two conditionally executed subsystems.



Bus Support

The Merge block is a bus-capable block. The inputs can be virtual or nonvirtual bus signals subject to these 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).

All signals in a nonvirtual bus input to a Merge block 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.

Merging S-Function Outputs

The Merge block can merge a signal from an S-Function block only if the memory used to store the output from the S-Function 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`.

Limitations

- All signals that connect to a Merge block, are functionally the same signal. Therefore, they are 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.
- Do not set the outports of conditionally executed subsystems being merged to reset when disabled. This action can cause multiple subsystems to update the block at the same time. Specifically, the disabled subsystem updates the Merge block by resetting its output, while the enabled subsystem updates the block by computing its output.

  To prevent this behavior, set the Outport block parameter **Output when disabled** to `held` for each conditionally executed subsystem being merged.

  Note If you are using Simplified Initialization Mode, set the Outport block parameter **Output when disabled** to `held`.
- A Merge block does not accept input signals whose elements have been reordered or partially selected. In addition, do not connect input signals to the block that have been combined outside of a conditionally executed subsystem.

You can use an array of buses as an input signal to a Merge block with these limitations:

- **Allow unequal port widths** — Clear this parameter.
- **Initial condition** — You can specify this parameter using:
  - The value `0`. In this case, each 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.

Ports

Input

Port_1 — First input signal

scalar | vector

First input signal merged with the other input signals.

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

Port_n — nth input signal

scalar | vector

nth input signal merged with the other input signals.

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

Output

Port_1 — Output signal

scalar | vector

Output signal merged from the input signals.

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

Parameters

Number of inputs — Number of input signals to merge

2 (default) | integer

Specify the number of input signals to merge. The block creates a port for each input signal.

Programmatic Use
Block Parameter: `Inputs`

Type: character vector
Values: integer
Default: `'2'`

`Initial output` — Initial output value

[ ] (default) | scalar | vector

Specify the initial value of the output signal. If you do not specify an initial output value, then 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 can vary, initialization can be inconsistent for the simulation and the code generation of a model.

Programmatic Use
Block Parameter: `InitialOutput`
Type: character vector
Values: scalar | vector
Default: `'[ ]'`

`Allow unequal port widths` — Allow inputs of unequal dimensions

`off` (default) | `on`

Select this check box to allow the block to accept inputs having different numbers of elements. 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

$\mathrm{max}(w_1+o_1,\ w_2+o_2,\ ...\ w_n+o_n)$

where w_1, ... w_n are the widths of the input signals and o_1, ... o_n are the offsets for the input signals.

If you clear this check box, the Merge block accepts only inputs of equal dimensions and outputs a signal of the same dimensions as the inputs.

Programmatic Use
Block Parameter: `AllowUnequalInputPortWidths`
Type: character vector

Values: `'off'` | `'on'`
Default: `'off'`

`Input port offsets` — Offset for input signals
`[ ]` (default) | vector

Enter a vector to specify the offset of each input signal relative to the beginning of the output signal.

Programmatic Use
Block Parameter: `InputPortOffsets`
Type: character vector
Values: scalar | vector
Default: `'[ ]'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

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 does not specify any initial outputs.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Manual Switch | Switch

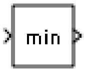
Topics

“Conditionally Executed Subsystems Overview”

Introduced before R2006a

MinMax

Output minimum or maximum input value
Library: Simulink / Math Operations



Description

The MinMax block outputs either the minimum or the maximum element or elements of the inputs. You choose whether the block outputs the minimum or maximum values by setting the **Function** parameter.

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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Provide an input signal from which the block outputs the maximum or minimum values.

- When 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.
- When 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.

Dependencies

To support matrix input, you must set the **Number of input ports** parameter to an integer greater than one. All nonscalar inputs must have the same dimensions.

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

`Port_N` — N-th input signal

scalar | vector | matrix

Provide an input signal from which the block outputs the maximum or minimum values.

When 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.

Dependencies

To provide more than one input signal, set the **Number of input ports** to an integer greater than 1.

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

Output

`Port_1` — Minimum or maximum values of inputs

scalar | vector | matrix

When the block has one input, the output is a scalar value, equal to the minimum or maximum of the input elements. When the block has multiple inputs, the output is a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

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

Parameters

Main

`Function` — Specify minimum or maximum

`min` (default) | `max`

Specify whether to apply the function `min` or `max` to the input.

Programmatic Use
Block Parameter: `Function`
Type: character vector
Values: `'min' | 'max'`
Default: `'min'`

`Number of input ports` — Specify number of input ports

`1` (default) | `positive integer`

Specify the number of inputs to the block.

Programmatic Use
Block Parameter: `Inputs`
Type: character vector
Values: `'1'`
Default: `'1'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than -1

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

`Require all inputs to have the same data type` — Inputs must have the same data type

`off` (default) | `on`

Select this check box to require that all inputs have the same data type.

Programmatic Use

Block Parameter: `Inputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Inherit via back propagation'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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	Rationale	Impact on Overflows	Example
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`

Variable-Size Signals	`Yes`
Zero-Crossing Detection	`Yes`

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™.

For more information about generating HDL code, see MinMax.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

MinMax Running Resettable

Introduced before R2006a

MinMax Running Resettable

Determine minimum or maximum of signal over time

Library: Simulink / 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 running minimum or maximum with the **Function** parameter.

The block can reset its state based on an external reset signal R. When the reset signal R is nonzero (true), the block resets the output to the value of the **Initial condition** parameter.

The input can be a scalar, vector, or matrix signal. 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.

Ports

Input

u — Input signal

scalar | vector | matrix

Input signal as a scalar, vector, or matrix. Based on what you specify for the **Function** parameter, the block outputs the minimum or maximum value of all past inputs u.

If you specify a scalar value for the **Initial condition** parameter, the block expands the parameter to have the same dimensions as nonscalar input u.

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

R — Reset signal
scalar | vector | matrix

The input port accepting the reset signal as a scalar, vector, or matrix. When the reset signal is nonscalar, it must have the same dimensions as input signal u. As long as the reset signal has a value of zero, the block outputs the running minimum or maximum value of input u. Whenever the reset signal has a nonzero value (true), the block resets the output to the value of the **Initial condition** parameter.

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

Output

y — Running minimum or maximum value
scalar | vector | matrix

Output signal, specified as a scalar, vector, or matrix, where each output element equals the running minimum or maximum value of the corresponding input elements. Output signal y has the same data type and dimensions as input signal u.

When the block receives a nonzero (true) reset signal, the block resets the output to the value of the **Initial condition** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Function — Specify minimum or maximum
min (default) | max

Specify whether the block outputs the running minimum or maximum value of the corresponding input elements.

Programmatic Use
Block Parameter: Function

Type: character vector
Values: `'min' | 'max'`
Default: `'min'`

`Initial condition` — Value to reset output to

`0.0` (default) | scalar or vector

Specify the initial condition value. When the reset input signal `R` is `true`, the block resets the output to the value you specify.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar or vector
Default: `'0.0'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

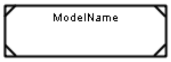
MinMax

Introduced before R2006a

Model

Include multiple model implementations as block in another model through model reference

Library: Simulink / 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 References” for more information.

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.

To set the referenced model and simulation parameters, open the **Block Parameters** dialog box and use the **Main** tab. To specify values for model arguments, use the **Arguments** tab.

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 “Model Reference Interface”.

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 “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 “Model Reference Interface”.

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 “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 model reference

`Normal` (default) | `Accelerator` | `Software-in-the-loop (SIL)` | `Processor-in-the-loop (PIL)`

Specify the simulation mode for the Model block. The simulation mode for the Model block can be different than the simulation mode of its referenced model and of other models in the model hierarchy.

- `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 of the Model block. 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.

While you can set a referenced model to rapid accelerator mode, simulation ignores the referenced model simulation mode. For information about simulation mode precedence in a model hierarchy, see “Simulate Model Hierarchies”.

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 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'`

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”.

When changing argument values, you can use a partial structure, which has fields that correspond to only the arguments whose values you want to change. Arguments not included in the partial structure retain their values. In the structure, include the model argument names and values, represented as character vectors.

Programmatic Use
Block parameter: `ParameterArgumentNames`
Type: character vector
Value: character vector in the form of `'argument1,argument2'`
Default: none

Programmatic Use
Block parameter: `ParameterArgumentValues`
Type: structure
Value: structure
Default: structure with no fields

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[a] Actual data type or capability support depends on block implementation.

Compatibility Considerations

Model blocks will no longer support model variants

Warns starting in R2017b

In a future release, Model blocks that contain variant models will convert to Variant Subsystem blocks that reference the variant models. Variant Subsystem blocks provide these advantages:

- Allow you to mix Model blocks and Subsystem blocks as variant systems
- Support flexible I/O, so that variants can have a different number of input and output ports

To convert a Model block that contains variant models to a Variant Subsystem block that contains Model blocks referencing the variant models, use one of these approaches:

- Right-click the Model block and select **Subsystems & Model Reference** > **Convert to** > **Variant Subsystem**.
- Use the `Simulink.VariantManager.convertToVariant` function. Specify the Model block path or block handle.

If you convert a Model block to a Variant Subsystem block, the Model block parameter **Generate preprocessor conditionals** behaves differently from the Variant Subsystem block parameter **Analyze all choices during update diagram and generate preprocessor conditionals**. For Model blocks that contain model variants, enabling the parameter causes simulation and update diagram to compile the active variant only. For Variant Subsystem blocks, enabling the parameter compiles all the variants, which can make simulation and updates slower.

Converting model variants to subsystem variants can require you to update scripts that use the `Variants` command-line parameter.

Extended Capabilities

C/C++ Code Generation

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

Actual code generation support depends on block implementation.

HDL Code Generation

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

For more information about HDL code generation, see Model.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

`Simulink.SubSystem.convertToModelReference` | `Simulink.VariantManager.convertToVariant` | `depview` | `find_mdlrefs`

Topics

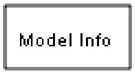
“Reference Existing Models”
“Parameterize Instances of a Reusable Referenced Model”
“Simulate Model Hierarchies”

Introduced before R2006a

Model Info

Display model properties and text in model

Library: Simulink / 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.

Parameters

`Specify Text and Properties to Display` — Content and format of the text to display

no default

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 replaces 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”.

Block Characteristics

Data Types	
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

The Model Info block is ignored during code generation.

HDL Code Generation

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

For more information about HDL code generation, see Model Info.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

DocBlock

Topics

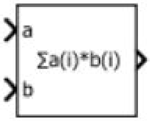
“Version Information Properties”
“Source Control in Simulink Project”

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 inputs a and b, and bias c, to compute result dataOut.

```
dataOut = sum(a.* b) + c
```

By default, the block operates in the vector mode. The inputs a and b can be scalars or vectors. The default bias value, c, is equal to zero, and the block computes the dot product of inputs a and b. You can specify a nonzero value for c using Dialog or Input port as the **Source**. The block adds this bias to the dot product of a and b. 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.

By using the **Operation Mode** setting, you can specify streaming modes of operation for the Multiply-Accumulate block. For HDL code generation, when you use the streaming operation mode, input scalar values to the block. The block has two streaming modes: Streaming - using Start and End ports and Streaming - using Number of Samples. When you select these streaming modes, you can specify the control signals to use with the mode. The control signals specify when to start and end accumulation, and when the output is valid.

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 `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`startIn` — Start of accumulation control signal

`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to start accumulation. It is recommended that you use a `boolean` data type signal as input to the port. To start obtaining the accumulated output value from the **dataOut** signal, both **startIn** and **validIn** signals must be high. The **dataOut** signal produces the accumulated result from the next clock cycle.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`validIn` — Valid input control signal

`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to indicate that the input signal is valid for accumulation. It is recommended that you use a `boolean` data type signal as input to the port. To start obtaining the accumulated output value from the **dataOut** signal, both **validIn** and **startIn** signals must be high. The **dataOut** signal produces the accumulated result from the next clock cycle. The **validIn** signal has higher priority than **startIn** and **endIn** signals.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` or `Streaming - using Number of Samples`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`endIn` — End of accumulation control signal

`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to indicate end of accumulation. You can use the **startIn** and **endIn** signals with the **validIn** signal to indicate a frame that contains the accumulated output.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` and then select **End input and output ports**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

dataOut — Output signal

scalar | vector | matrix | array | bus

Port that generates the output data from the multiply-accumulate operation. By default, the block uses the Vector mode of operation and computes the dot product of the input signals, and adds the bias to produce the result. If you specify a streaming mode of operation as **Operation Mode**, the value of the **dataOut** signal depends on the control signals that you provide. The data type of the output signal is same as that of the accumulator.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

startOut — Start of accumulation output control signal

scalar | vector | matrix | array | bus

Port that generates output control signal to indicate the start of accumulation. When both **validIn** and **startIn** are high, the **startOut** signal becomes high in the next clock cycle. The clock cycle at which **startOut** becomes high indicates the start of a frame and that the **dataOut** signal has started producing valid accumulated output.

Dependencies

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports and then select **Start output port**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

validOut — Valid output control signal

scalar | vector | matrix | array | bus

Port that generates the output control signal to indicate that the **dataOut** signal is valid. When the **validIn** signal becomes high , the **validOut** signal becomes high in the next clock cycle and indicates that the **dataOut** is valid.

Dependencies

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports and then select **Valid output port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`endOut` — End of accumulation output control signal

`scalar` | `vector` | `matrix` | `array` | `bus`

Port that generates the output control signal to indicate the end of accumulation. You can use the clock cycles between when the **startOut** signal becomes high and when the **endOut** signal becomes high to indicate a valid frame that contains the accumulated output.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` and then select **End input and output ports**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`countOut` — Count output control signal

`scalar` | `vector` | `matrix` | `array` | `bus`

Port that generates the output control signal to indicate number of samples to accumulate. The value of this signal increases from `1` to the value that you specify for **Number of Samples**. As long as the **validIn** signal is high, the **countOut** increments by `1` every clock cycle.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples` and then select **Count output port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Operation Mode` — Mode of accumulation of inputs

`'Vector'` (default) | `'Streaming - using Start and End Ports'` | `'Streaming - using Number of Samples'`

You can specify the **Operation Mode** as:

- `Vector`: You can use scalars or vectors as inputs. The block performs the dot product of the inputs `u1` and `u2` and adds bias `k` to produce the result.
- `Streaming - using Start and End Ports`: Use scalar inputs for HDL code generation. In this mode, you can use the **startIn** and **endIn** control signals to determine when to start and stop accumulation. The output data is valid when **validIn** is high.
- `Streaming - using Number of Samples`: Use scalar inputs for HDL code generation. In this mode, you can specify the **Number of Samples** and use the **countIn** control signal to determine when to start and stop accumulation. The output data is valid when **validIn** is high.

Programmatic Use

Block parameter: `opMode`
Type: character vector
Value: `'Vector'` | `'Streaming - using Start and End Ports'` | `'Streaming - using Number of Samples'`
Default: `'Vector'`

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.

Programmatic Use

Block parameter: `initValueSetting`
Type: character vector
Value: `'Dialog'` | `'Input port'`
Default: `'Dialog'`

If you set **Source** as `Dialog`, you can specify the initial value by using the `initValue2` setting.
Block parameter: `initValue2`
Type: character vector
Value: An integer greater than or equal to zero
Default: `'0.0'`

`Number of Samples` — Number of samples of valid accumulated output signal

`{'2'}` (default)

You can specify the **Number of Samples** to specify a frame containing the number of samples of valid accumulated output **dataOut**.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples`.

Programmatic Use
Block parameter: `num_samples`
Type: character vector
Value: An integer greater than or equal to zero
Default: `'2'`

`Output data type` — Data type of the block output

`Inherit: Inherit via back propagation` (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)`

The streaming modes do not suuport `Inherit: Inherit via internal rule`. 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.

`Valid output port` — Control generation of validOut output port

off (default) | on

Control generation of the **validOut** output port. This port indicates whether **dataOut** is valid.

□ off

Does not display the **validOut** output port.

☑ on

Display the **validOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples` or `Streaming - using Start and End Ports`.

Programmatic Use
Block parameter: `validOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`End input and output ports` — Control generation of endIn input port and endOut output port

off (default) | on

Control generation of the **endIn** input port and the **endOut** output port. The ports indicate the end of a frame containing valid accumulation output.

□ off

Does not display the **endIn** input port and the **endOut** output port.

☑ on

Display the **endIn** input port and the **endOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports`.

Programmatic Use
Block parameter: `endInandOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Start output port` — Control generation of startOut output port

off (default) | on

Control generation of the **startOut** output port. This port generates the **startOut** signal that indicates the start of a frame containing valid accumulated output.

□ off

Does not display the **startOut** output port.

☑ on

Display the **startOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports`.

Programmatic Use

Block parameter: `startOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Count output port` — Control generation of countOut output port

off (default) | on

Control generation of the **countOut** output port. This port generates the counter that indicates a frame containing valid samples.

□ off

Does not display the **countOut** output port.

☑ on

Display the **countOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples`.

Programmatic Use
Block parameter: `countOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

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`

Algorithms

Streaming Mode Using Start and End Ports

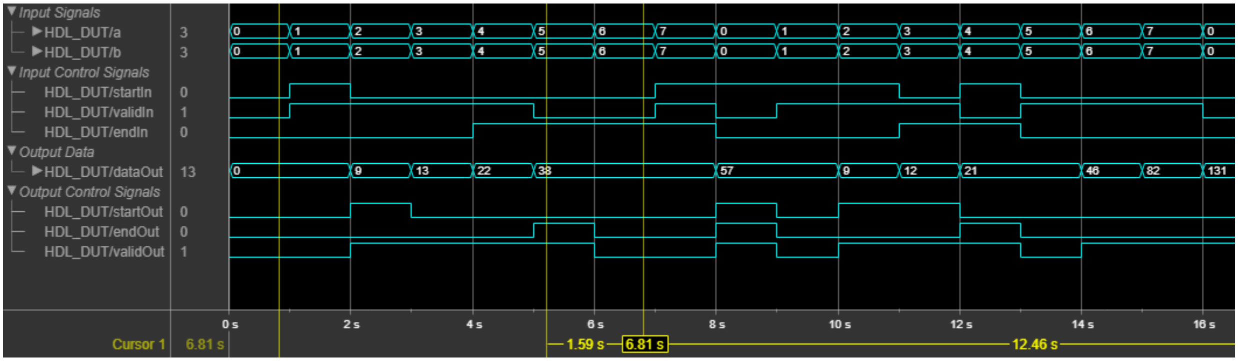
You can use the **Operation Mode** setting for the block to specify a streaming mode of operation. When you select `Streaming - using Start and End Ports`, you see three additional settings enabled by default. The settings include:

- **Valid output port**
- **End input and output ports**
- **Start output port**

It is recommended that you leave these settings enabled. When you apply the settings, three additional input ports and three additional output ports appear:

Input Ports	Output Ports
startIn	**startOut**
validIn	**validOut**
endIn	**endOut**

This figure illustrates the streaming mode of operation using the start and end ports. In this example, the bias value is 8.



Initially, when **validIn** is low, **dataOut** is zero. At time 1s, both **startIn** and **validIn** become high. Therefore, **validOut** becomes high in the next clock cycle and **dataOut** starts producing valid accumulation output. During accumulation, **dataOut** takes the values of **a** and **b** from the previous clock cycle. For example, at time t = 2s, **dataOut** = 1*1 + 8 = 9.

To continue accumulation, make **startIn** low at the next clock cycle and keep **validIn** high. **dataOut** continues accumulating the inputs until **validIn** becomes low. At each time step, **dataOut** computes the product of the inputs from the previous clock cycle and sums the result with the **dataOut** value from the previous clock cycle. For example, at time t = 3s, **dataOut** = 2*2 + 9 = 13.

When **validIn** becomes low, **dataOut** holds the output value as seen at time t = 5s. At t = 5s, **endIn** and **validIn** are high. Therefore, **endOut** becomes high in the next clock cycle, which indicates end of frame. Therefore the frame between t = 2s (when **startOut** is high) and t = 6s (when **endOut** is high) indicates a frame containing valid output.

If **startIn**, **validIn**, and **endIn** are both high at the same time, only the **dataOut** corresponding to those inputs are accumulated as seen at t = 8s. If **startIn** is high for multiple clock cycles, and if **validIn** is high, the accumulator is reset at each clock cycle as seen at t = 10s and t = 11s. The accumulation continues at t = 12s.

Streaming Mode Using Number of Samples

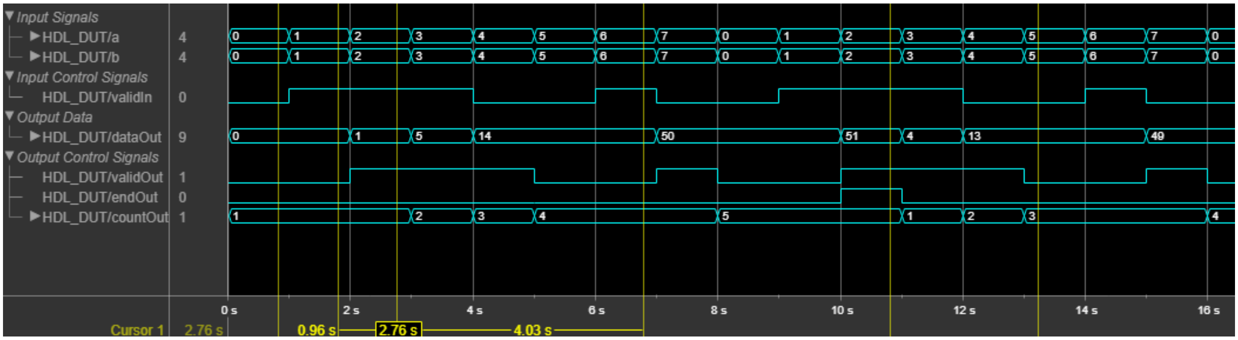
You can use the **Operation Mode** setting for the block to specify a streaming mode of operation. When you select `Streaming - using Number of Samples`, you see two additional settings enabled by default. The settings include:

- **Valid output port**
- **Count output port**

It is recommended that you leave these settings enabled. When you apply the settings, you have an additional input port **validIn** and three additional output ports appear:

- **endOut**
- **validOut**
- **countOut**

This figure illustrates the streaming mode of operation using the number of samples. In this example, the bias value is 8 and the **Number of Samples** is 5.



Initially, when **validIn** is low, **dataOut** is `0` and **countOut** is `1`. At time `1s`, **validIn** becomes high. Therefore, **validOut** becomes high in the next clock cycle, and **dataOut** starts producing valid accumulation output. During accumulation, **dataOut** takes the values of **a** and **b** from the previous clock cycle. For example, at time t = `2s`, **dataOut** = 1*1 = 1. **countOut** increments by `1` at the next clock cycle, that is, at t = `3s`, **countOut** becomes `2`.

To continue accumulation, keep **validIn** high. **dataOut** continues accumulating the inputs until **validIn** becomes low. When five valid outputs are obtained from **dataOut**, **countOut** becomes `5` and **endOut** becomes high, which indicates the end of the frame. Therefore, the time between when **countOut** is `1` and when **countOut** is five indicates a frame containing valid output.

The accumulator counter is now reset and **countOut** starts from `1`. When **validIn** becomes high again, **dataOut** starts accumulating a new set of values and **countOut** starts incrementing for each valid **dataOut**.

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

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

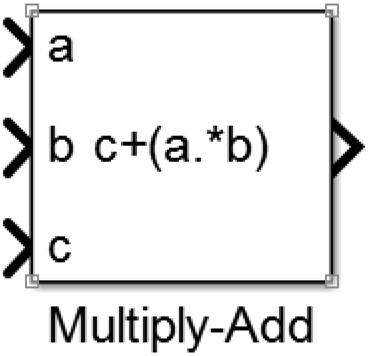
See Also

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.

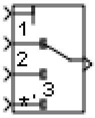
Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

See Also

Introduced in R2015b

Multiport Switch

Choose between multiple block inputs
Library: Simulink / Signal Routing



Description

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.

Control Input	Truncation	Setting for Data Port Order	Block Behavior During Simulation	
			Indexing to Select Data Input	Out-of-Range Condition
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 1-1285.

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.

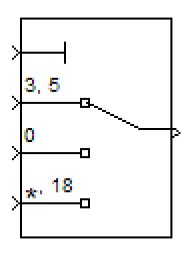
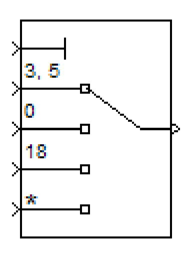
To configure a Multiport Switch block to work as an Index Vector block, set **Number of data ports** to `1` and **Data port order** to `Zero-based contiguous`.

How the Block Handles an Out-of-Range Control Input

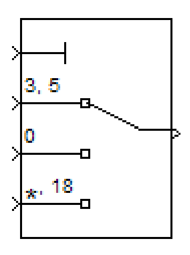
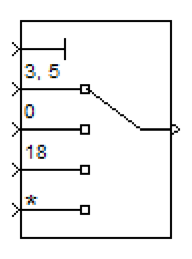
For an input with an integer value less than `intmax('int32')`, the input is out of range when the value does not match any data port indices. For a control input that is not an integer value, the input is out of range when the *truncated* value does not match any data port indices. In both cases, the 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
`Last data port` 	Use the last data port and do not report any warning or error.	Use the last data port and report a warning.	Report an error and stop simulation.
`Additional data port` 	Use the additional data port with a * label and do not report any warning or error.	Use the additional data port with a * label and report a warning.	Report an error and stop simulation.

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 (Results in variable-size output signal)” on page 1-0 . 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 1-1285.
- 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.	• Set **Data port order** to `Specify indices`. • Set **Data port indices** to use this value for the last data port. • Set **Data port for default case** to `Last data port`.	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.	• Set **Data port for default case** to `Additional data port`.	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.	• Set **Data port for default case** to `Last data port`. • Set **Diagnostic for default case** to `None`.	This block configuration avoids unnecessary diagnostic action.
The block does not have a data input port for every value of the enumerated type.	• Set **Data port for default case** to `Additional data port`.	This block configuration handles enumerated values that do not have a data input port, along with invalid values.

Ports

Input

`Port_1` — Control signal

scalar | vector | matrix | N-D array

The control signal can be of any data type that Simulink supports, including fixed-point and enumerated types. When the control input is not an integer value, the block truncates the value to an integer by rounding to zero.

For information on control signals of enumerated type, see “Guidelines on Setting Parameters for Enumerated Control Port” on page 1-1288.

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 1-1285.

Limitations

- If the control signal is numeric, the control signal 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.

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

1 — First data input

scalar | vector | matrix | N-D array

First data input, specified as a scalar, vector, matrix, or N-D array. All input data signals can be of any data type that Simulink supports.

- If all the data inputs are scalar, the output is scalar
- 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 any two nonscalar signals have a different number of dimensions or different dimension lengths, select the **Allow different data input sizes** check box. For more information, see “Use Data Inputs That Have Different Dimensions” on page 1-1287
- If any data signal is of an enumerated type, all others must be of the same enumerated type.

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

2 — Second data input

scalar | vector | matrix | N-D array

Second data input, specified as a scalar, vector, matrix, or N-D array. All input data signals can be of any data type that Simulink supports.

- If all the data inputs are scalar, the output is scalar
- 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 any two nonscalar signals have a different number of dimensions or different dimension lengths, select the **Allow different data input sizes** check box. For more information, see “Use Data Inputs That Have Different Dimensions” on page 1-1287
- If any data signal is of an enumerated type, all others must be of the same enumerated type.

Dependencies

To enable this port, set **Number of data ports** to an integer greater than 1.

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

`N` — Nth data input

scalar | vector | matrix | N-D array

Nth data input, specified as a scalar, vector, matrix, or N-D array. All input data signals can be of any data type that Simulink supports.

- If all the data inputs are scalar, the output is scalar
- 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 any two nonscalar signals have a different number of dimensions or different dimension lengths, select the **Allow different data input sizes** check box. For more information, see "Use Data Inputs That Have Different Dimensions" on page 1-1287
- If any data signal is of an enumerated type, all others must be of the same enumerated type.

Dependencies

To enable the **N**th input port, set **Number of data ports** to an integer value greater than or equal to `N`.

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

* — Data port for out-of-range inputs

scalar | vector | matrix | N-D array

Input data port for out-of-range control signal inputs, specified as a scalar, vector, matrix, or N-D array. All input 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. If any two signals have a different number of dimensions or different dimension lengths, select the **Allow different data input sizes** check box. For more information, see "Use Data Inputs That Have Different Dimensions" on page 1-1287.

Dependencies

To create an additional data port for out-of-range control signal inputs, set **Data port for default case** to `Additional data port`. When you set **Data port for default case** to `Last data port`, the block uses the last data port for output when the control signal value does not match any data port indices.

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

Output

`Port_1` — Selected data input, based on control signal value

scalar | vector | matrix | N-D array

The block outputs one of the data inputs, selected according to the control signal value. The output has the same dimensions as the corresponding data input. When you select the **Allow different data input sizes** check box, the output of the block is a variable size signal.

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

Parameters

Main

`Data port order` — Type of ordering for data input ports

`One-based contiguous` | `Zero-based contiguous` | `Specify indices`

Specify the type of ordering for your data input ports.

- `Zero-based contiguous` — Block uses zero-based indexing for ordering contiguous data ports. This is the default value of the Index Vector block.
- `One-based contiguous` — Block uses one-based indexing for ordering contiguous data ports. This is the default value of the Multiport Switch block
- `Specify indices` — Block uses noncontiguous indexing for ordering data ports.

Tips

- 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:	The Block Produces:
`Zero-based contiguous` or `One-based contiguous`	A warning
`Specify indices`	An error

Dependencies

Selecting `Zero-based contiguous` or `One-based contiguous` enables the **Number of data ports** parameter.

Selecting `Specify indices` enables the **Data port indices** parameter.

Programmatic Use
Block Parameter: `DataPortOrder`
Type: character vector
Values: `'Zero-based contiguous' | 'One-based contiguous' | 'Specify indices'`
Default: `'One-based contiguous'` (Multiport Switch) `'Zero-based contiguous'` (Index Vector)

`Number of data ports` — Number of data input ports

`1` | `3` | integer between 1 and 65536

Specify the number of data input ports to the block. The total number of input ports is the number you specify, plus one for the control signal input port, and plus one more if you set **Data port for default case** to `Additional data port`.

Dependencies

To enable this parameter, set **Data port order** to `Zero-based contiguous` or `One-based contiguous`.

Programmatic Use
Block Parameter: `Inputs`
Type: character vector
Values: integer between 1 and 65536

Default: `'3'` (Multiport Switch) `'1'` (Index Vector)

`Data port indices` — Array of indices for data ports

`{1,2,3}` (default) | array of indices

Specify an array of indices for your data ports. The block icon changes to match the data port indices you specify.

Tips

- 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.

Dependencies

To enable this parameter, set **Data port order** to `Specify indices`.

Programmatic Use
Block Parameter: `DataPortIndices`
Type: character vector

Values: array of indices
Default: `'{1,2,3}'`

`Data port for default case` — Port to use for out-of-range inputs

`Last data port` (default) | `Additional data port`

Specify whether to use the last data port for out-of-range inputs, or to use an additional port. An asterisk (*) next to the port name indicates the port the block uses when the control port value does not match any data port indices.

- `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.

Tip

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.

Programmatic Use
Block Parameter: `DataPortForDefault`
Type: character vector
Values: `'Last data port' | 'Additional data port'`
Default: `'Last data port'`

`Diagnostic for default case` — Diagnostic action when control port value does not match data port indices

`Error` (default) | `Warning` | `None`

Specify the diagnostic action to take when the control port value does not match any data port indices.

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display 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 1-1285.

Programmatic Use
Block Parameter: `DiagnosticForDefault`
Type: character vector
Values: `'None' | 'Warning' | 'Error'`
Default: `'Error'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

`Require all data port inputs to have the same data type` — Require all inputs to have the same data type

`off` (default) | `on`

Select this check box to require that all data input ports have the same data type. When you clear this check box, the block allows data port inputs to have different data types.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

When you select an inherited option, the block behaves as follows:

- `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.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'`, `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Integer rounding mode` — Specify the rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Choose one of these rounding modes.

`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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

See Also

For more information, see “Rounding” (Fixed-Point Designer).

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Allow different data input sizes (Results in variable-size output signal)` — Allow input signals with different sizes
`off` (default) | `on`

Select this check box to allow input signals with different sizes.

- `On` — Allows input signals with different sizes, and propagate the input signal size to the output signal. In this mode, the block produces a variable-size output signal.
- `Off` — Requires that all nonscalar data input signals be the same size.

Programmatic Use
Parameter: `AllowDiffInputSizes`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`

Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Multiport Switch.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Index Vector | Switch

Topics

"Variable-Size Signal Basics"

Introduced before R2006a

MultiStateImage

Display image reflecting input value
Library: Simulink / Dashboard

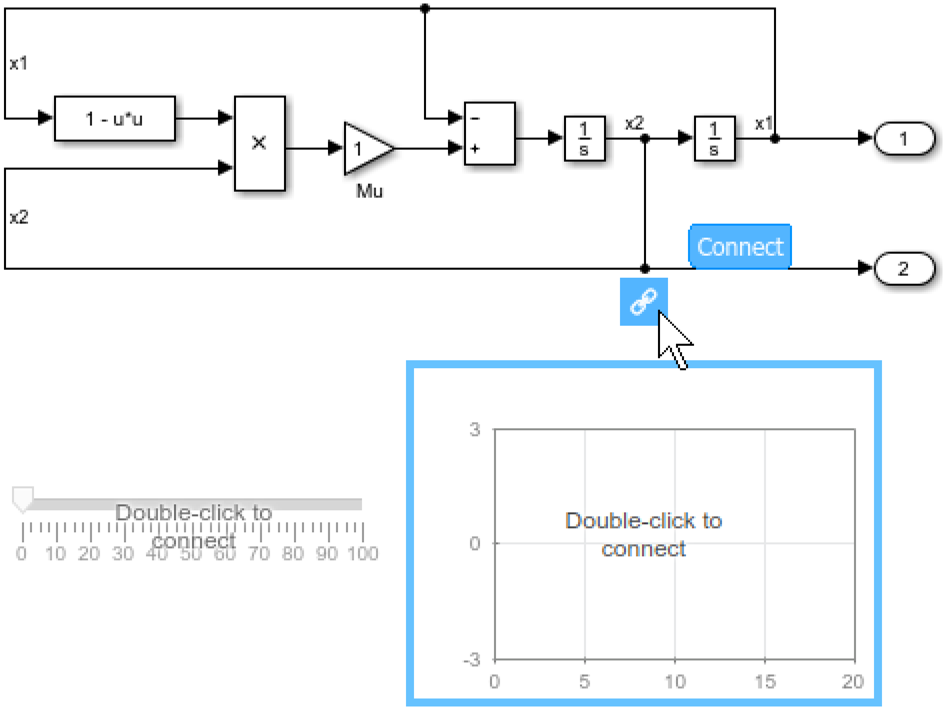
Description

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.

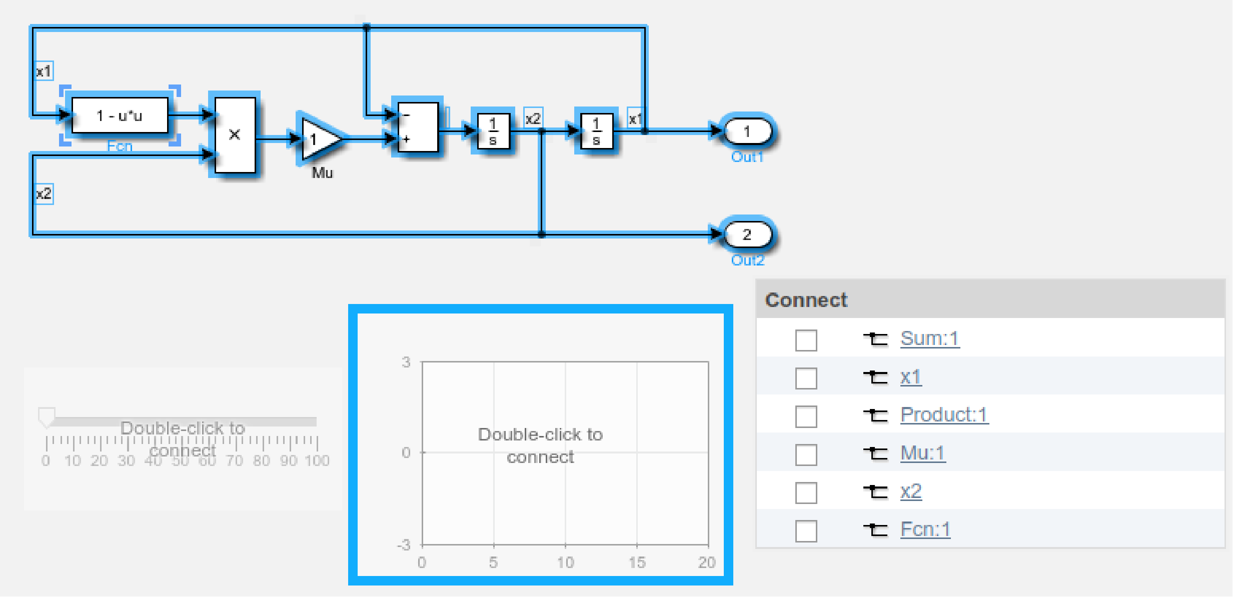
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

- □ Sum:1
- □ x1
- □ Product:1
- □ Mu:1
- □ x2
- □ Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced 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.

Parameters

`Connection` — Select a signal to connect and display

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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.

See Also

Lamp

Topics

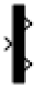
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

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	Specifies the number of inputs to the Mux block. When you use this format, the block accepts scalar or vector signals of any size. Simulink assigns each input the name `signalN`, where `N` is the input port number.
Vector	The length of the vector specifies the number of inputs. Each element specifies the size of the corresponding input. A positive value specifies that the corresponding port can accept only vectors of that size. For example, `[2 3]` 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 `-1` specifies that the corresponding port can accept scalars or vectors of any size.

Format	Block Behavior
Cell array	The length of the cell array specifies the number of inputs. The value of each cell specifies the size of the corresponding input. A scalar value `N` specifies a vector of size `N`. A value of `-1` means that the corresponding port can accept scalar or vector signals of any size.
Signal name list	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 `position,velocity`, the Mux block has two inputs, named `position` and `velocity`.

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'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`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™.

This block has a single, default HDL architecture. See Mux.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point 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 🗑.

To reduce the number of bus element signals displayed in the Block Parameters dialog box, use the **Filter** box. The **Filter** box supports regular expressions. To use a regular expression character as a literal, include an escape character (`\`). For example, to use a question mark: `sig\?1`.

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`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`

Zero-Crossing Detection	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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

See Also

Blocks

Bus Creator | In Bus Element | Outport

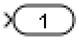
Topics

"Simplify Subsystem Bus Interfaces"
"Composite Signal Techniques"
"Select a Composite Signal Technique"
"Getting Started with Buses"

Introduced in R2017a

Outport

Create output port for subsystem or external output
Library: Simulink / Commonly Used Blocks
Simulink / Ports & Subsystems
Simulink / Sinks



Description

Outport blocks link signals from a system to a destination outside of the system. They can connect signals flowing from a subsystem to other parts of the model. They can also supply external outputs at the top level of a model hierarchy.

Simulink software assigns Outport block port numbers according to these rules:

- It automatically numbers the Outport blocks within a root-level system or subsystem sequentially, starting with 1.
- If you add an Outport block, it is assigned the next available number.
- If you delete an Outport block, other port numbers are automatically renumbered to ensure that the Outport blocks are in sequence and that no numbers are omitted.

Outport Blocks in a Subsystem

Outport blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Outport block in a subsystem flows out of the associated output port on that Subsystem block. The Outport block associated with an output port is the block whose **Port number** parameter matches the relative position of the output port on the Subsystem block. For example, the Outport 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. The block continues to send the signal to the same block outside the subsystem.

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. 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.

Top-Level Outport Block in a Model Hierarchy

Outport blocks at the top level of a model hierarchy have two uses. They can supply external outputs to the base MATLAB workspace, and they provide a means for the `linmod` and `trim` 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 Outport 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 Outport block. The column order matches the order of the port numbers for the Outport blocks.

If you specify more than one variable name after the second (state) argument, data from each Outport block is written to a different variable. For example, if the system has two Outport blocks, to save data from Outport block 1 to `speed` and the data from Outport block 2 to `dist`, specify this command:

```
[t,x,speed,dist] = sim(...);
```

Connecting Buses to Root-Level Outports

A root-level Outport of a model can accept a virtual bus only if all elements of the bus have the same data type. The Outport 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 Outport of a model to accept a bus signal that contains mixed types, set Outport block **Data type** to `Bus: <object name>` . If the bus signal is virtual, it is converted to nonvirtual, as described in “Bus Conversion”.

Associate Root-Level Outport Block with `Simulink.Signal` Object

To associate a root-level Outport block with a `Simulink.Signal` object, use the Model Data Editor. See “For Signals”.

Ports

Input

`Port_1` — Outport signal

scalar | vector

Input signal that flows through the outport to an external subsystem or model.

An Outport block can 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 its input.

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

Parameters

Main

`Port number` — Port number of block

1 (default)

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

Programmatic Use
Block Parameter: `Port`
Type: character vector
Values: real integer
Default: `'1'`

`Signal name` — Signal name

`' '` (default) | character vector

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.

Programmatic Use
Block Parameter: `SignalName`
Type: character vector
Values: character vector
Default: `' '`

`Icon display` — Icon display

`Port number` (default) | `Signal name` | `Port number and signal name`

Specify the information to be displayed on the icon of this port.

Programmatic Use
Block Parameter: `IconDisplay`
Type: character vector
Values: `'Signal name'` | `'Port number'` | `'Port number and signal name'`
Default: `'Port number'`

`Specify output when source is unconnected` — Specify unconnected output

`off` (default) | `on`

Specify a constant output value to be displayed when source is not connected.

Dependency

Enabled when a non-driven Outport block is in a Variant Subsystem block.

Programmatic Use
Block Parameter: `OutputWhenUnconnected`
Type: character vector
Values: `'on' | 'off'`
Default: `'off'`

`Constant value` — Specify unconnected output

`0` (default) | scalar | vector

Specify a constant output value to be displayed when source is not connected.

Dependency

Enabled in a Variant Subsystem block on a non-driven Output block.

Programmatic Use
Block Parameter: `OutputWhenUnconnectedValue`
Type: character vector
Values: | `real integer`
Default: `'1`

`Interpret vector parameters as 1-D` — Treat vectors as 1-D

`off` (default) | `on`

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.

Dependency

Enabled in a Variant Subsystem block on a non-driven Output block when you select the **Specify output when source is unconnected** parameter.

Programmatic Use
Block Parameter: `VectorParams1DForOutWhenUnconnected`
Type: character vector
Values: `'on' | 'off'`
Default: `'off'`

`Ensure outport is virtual` — Ensure that outport is virtual

`off` (default) | `on`

Clear this check box to specify that Simulinkuses a signal buffer on output port. This buffer ensures consistent initialization of the Outport block signal.

If you select this check box, Simulink tries to remove the signal buffer.

- If the signal buffer is not needed, Simulink removes 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.

For examples with conditional writes and partial writes, see “Ensure Outport is Virtual”.

Dependencies

This parameter applies to these blocks:

- Conditional subsystem
- Assignment
- Merge
- Model with root Outport block

`Source of initial output value` — Source of initial output value

`Dialog` (default) | `Input signal`

Select the source of the initial output value of the block. Select `Dialog` to specify that the initial output value is the value of the **Initial output** parameter. Select `Input signal` to specify that the initial output value is inherited from the input signal. See “Conditional Subsystem Initial Output Values”.

Tips

- If you are using classic initialization mode, selecting `Input signal` causes 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”.

Dependencies

This parameter is enabled when the Outport resides in a Conditional Subsystem.

Selecting `Dialog` enables the following parameters:

- **Output when disabled**
- **Initial output**

Programmatic Use
Block Parameter: `SourceOfInitialOutputValue`
Type: character vector
Values: `'Dialog' | 'Input signal'`
Default: `'Dialog'`

`Output when disabled` — Output when disabled

`held` (default) | `reset`

Specify what happens to the block output when the subsystem is disabled. Select `held` to indicate that the output is held when the subsystem is disabled. Select `reset` to indicate that the output is reset to the value given by **Initial output** when the subsystem is disabled.

Dependencies

To enable this parameter, select `Dialog` in **Source of initial output value** when the outport resides in a conditional sybsystem with valid enabling and disabling semantics.For example, this parameter is disabled when the Outport is placed inside a Triggered Subsystem but is enabled when the Outport is placed inside an Enabled Subsystem.

If an Outport 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).

When connecting the output of a conditional subsystem to a Merge block, set this parameter to `held`. Setting it to `reset` returns an error.

Programmatic Use
Block Parameter: `OutputWhenDisabled`
Type: character vector
Values: `'held' | 'reset'`
Default: `'held'`

`Initial output` — Initial output for conditionally executed subsystems

`[ ]` (default) | scalar

For conditionally executed subsystems, specify the block output before the subsystem executes and while it is disabled. 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”.

Tips

If the conditional subsystem is driving a Merge block, you do not need to specify an Initial Condition (IC) for the Outport block. For more information, see “Underspecified initialization detection”.

Dependencies

To enable this parameter, set **Source of initial output value** to `Dialog` when this block resides in a conditionally executed subsystem.

Limitations

- This block does not allow an initial output of `inf` or `NaN`.
- When the input is a virtual bus, an **Initial output** value `[]` is treated as double(0).
- When the input contains a nonvirtual bus, **Initial output** does not support nonzero scalar values.

Programmatic Use

Block Parameter: `InitialOutput`
Type: character vector
Values: `'[]'` | scalar
Default: `'[]'`

Signal Attributes

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Minimum` — Minimum output value

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

Maximum — Maximum output value

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Data type` — Output data type

`Inherit: auto` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class name>` | `Bus: <object name>` | `<data type expression>`

Specify the output data type of the external input. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. Do not specify a bus object as the expression.

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Output as nonvirtual bus in parent model` — Outport as nonvirtual bus in parent model

`off` (default) | `on`

Specify the outport bus to be nonvirtual in the 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**.

Clear this parameter if you want the bus emerging in the parent model to be virtual.

Tips

- 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 Outport 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.

Dependency

To enable this parameter, select **Data type** > `Bus: <object name>`.

Programmatic Use

Block Parameter: `BusOutputAsStruct`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Unit (e.g., m, m/s^2, N*m)` — Physical unit of the input signal to the block

`inherit` (default) | `<Enter unit>`

Specify the physical unit of the input signal to the block. 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.

Programmatic Use
Block Parameter: `Unit`
Type: character vector
Values: `'inherit'` | `'<Enter unit>'`
Default: `'inherit'`

`Port dimensions (-1 for inherited)` — Port dimensions

`-1` (default) | integer | [integer, integer]

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

`-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.

Programmatic Use
Block Parameter: `PortDimensions`
Type: character vector
Values: `'-1'` | integer | [integer, integer]
Default: `'-1'`

`Variable-size signal` — Allow variable-size signals

`Inherit` (default) | `No` | `Yes`

Specify the type of signals allowed out of this port. To allow variable-size and fixed-size signals, select `Inherit`. To allow only variable-size signals, select `Yes`. Not to allow variable-size signals, select `No`.

Dependencies

When the signal at this port is a variable-size signal, the **Port dimensions** parameter specifies the maximum dimensions of the signal.

Command-Line Information
Parameter: `VarSizeSig`
Type: character vector
Value: `'Inherit`'| `'No'` | `'Yes'`
Default: `'Inherit'`

`Sample time (-1 for inherited)` — Specify sample time

`-1` (default) | scalar

Specify the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

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”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

`Signal type` — Output signal type

`auto` (default) | `real` | `complex`

Specify the numeric type of the signal output. To choose the numeric type of the signal that is connected to its input, select `auto`. Otherwise, choose a real or complex signal type.

Programmatic Use
Block Parameter: `SignalType`
Type: character vector
Values: `'auto'` | `'real'` | `'complex'`
Default: `'auto'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For information about HDL Code Generation, see Inport.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Inport | Out Bus Element

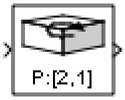
Topics

“Simplify Subsystem Bus Interfaces”
“Ensure Outport is Virtual”
“Specify Data Types Using Data Type Assistant”

Introduced before R2006a

Permute Dimensions

Rearrange dimensions of multidimensional array dimensions
Library: Simulink / Math Operations



Description

The Permute Dimensions block reorders the elements of the input signal by permuting its dimensions. You specify the permutation to be applied to the input signal using the **Order** parameter.

For example, to transpose a 3-by-5 input signal, specify the permutation vector `[2 1]` for the **Order** parameter. When you do, the block reorders the elements of the input signal and outputs a 3-by-5 matrix.

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”.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix | N-D array

This port accepts scalar, vector, matrix, and N-dimensional signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types.

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

Output

`Port_2` — Permutation of input signal
scalar | vector | matrix | N-D array

The block outputs the permutation of the input signal, according to the value of the **Order** parameter. The output has the same data type as the input.

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

Parameters

`Order` — Permutation vector
`[2,1]` (default) | `N`-element vector, where `N` is the number of dimensions of the input signal

Specify the permutation order to apply to the dimensions of the input signal. The value of this parameter must be an `N`-element vector where `N` is the number of dimensions of the input signal. The elements of the permutation vector must be a rearrangement of the values from 1 to `N`.

For example, the permutation vector `[2 1]` applied to a 5-by-3 input signal results in a 3-by-5 output signal, in other words, the transpose of the input signal.

Programmatic Use
Block Parameter: `Order`
Type: character vector
Value: N-element vector
Default: `'[2 1]'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`No`

Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Math Function | `permute`

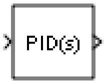
Topics

“Multidimensional Arrays” (MATLAB)

Introduced in R2007a

PID Controller

Continuous-time or discrete-time PID controller
Library: Simulink / Continuous



Description

The PID Controller block implements a PID controller (PID, PI, PD, P only, or I only). The block is identical to the Discrete PID Controller block with the **Time domain** parameter set to `Continuous-time`.

The 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.

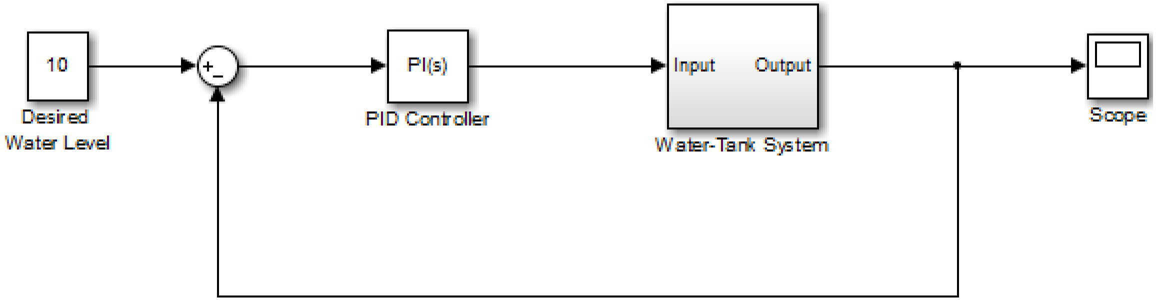
The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, PD, P only, or I only) — See the **Controller** parameter.
- Controller form (Parallel or Ideal) — See the **Form** parameter.
- Time domain (continuous or discrete) — See the **Time domain** parameter.
- Initial conditions and reset trigger — See the **Source** and **External reset** parameters.
- Output saturation limits and built-in anti-windup mechanism — See the **Limit output** parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the **Enable tracking mode** parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (See “Variant Subsystems”.) To examine the internal structure of the block and its variant subsystems, right-click the block and select **Mask > Look Under Mask**.

Control Configuration

In one common implementation, the PID Controller block operates in the feedforward path of a 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 PID Controller (2DOF).

PID Gain Tuning

The PID controller coefficients are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software. For more information about automatic tuning, see the **Select tuning method** parameter.

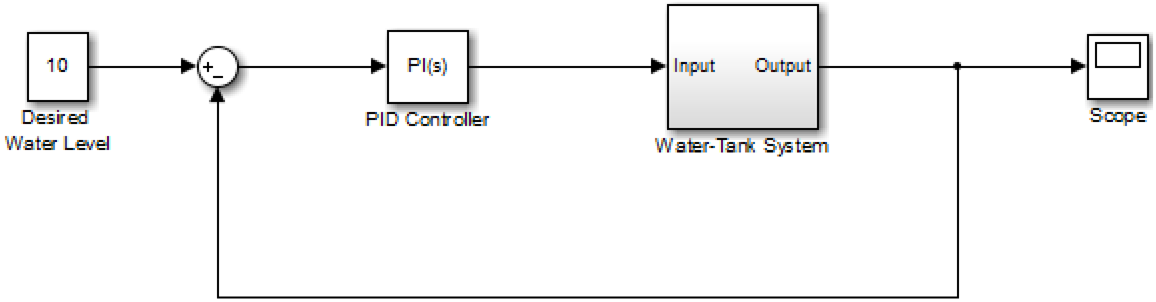
Ports

Input

`Port_1( u )` — Error signal input

scalar | vector

Difference between a reference signal and the output of the system under control, as shown.



When the error signal is a vector, the block acts separately on each signal, vectorizing the PID coefficients and producing a vector output signal of the same dimensions. You can specify the PID coefficients and some other parameters as vectors of the same dimensions as the input signal. Doing so is equivalent to specifying a separate PID controller for each entry in the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

P — Proportional gain

scalar | vector

Proportional gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

I — Integral gain

scalar | vector

Integral gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`D` — Derivative gain

scalar | vector

Derivative gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`N` — Filter coefficient

scalar | vector

Derivative filter coefficient, provided from a source external to the block. External coefficient 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

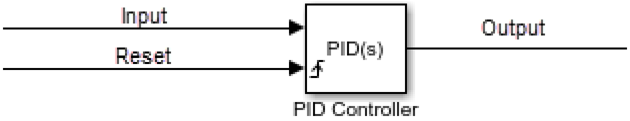
To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has a filtered derivative.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Reset` — External reset trigger

scalar

Trigger to reset the integrator and filter to their initial conditions. The value of the **External reset** parameter determines whether reset occurs on a rising signal, a falling signal, or a level signal. The port icon indicates the selected trigger type. For example, the following illustration shows a continuous-time PID block with **External reset** set to `rising`.



When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the **Integrator Initial condition** and **Filter Initial condition** parameters or the $\mathbf{I_0}$ and $\mathbf{D_0}$ ports.

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.

Dependencies

To enable this port, set **External reset** to any value other than `none`.

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

$\mathtt{I_0}$ — Integrator initial condition

scalar | vector

Integrator initial condition, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

D_0 — Filter initial condition

scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

TR — Tracking signal

scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input. Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the **Enable tracking mode** parameter.

Dependencies

To enable this port, select the **Enable tracking mode** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1( y )` — Controller output

scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the **Controller** parameter. The base controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the **Upper Limit** and **Lower Limit** saturation parameters.

The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as *N* independent PID controllers, where *N* is the number of signals in the input vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Controller` — Controller type

`PID` (default) | `PI` | `PD` | `P` | `I`

Specify which of the proportional, integral, and derivative terms are in the controller.

`PID`

Proportional, integral, and derivative action.

`PI`

Proportional and integral action only.

`PD`

Proportional and derivative action only.

`P`

Proportional action only.

`I`

Integral action only.

Tip The controller transfer function for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use

Block Parameter: `Controller`
Type: string, character vector
Values: `"PID"`, `"PI"`, `"PD"`, `"P"`, `"I"`
Default: `"PID"`

`Form` — Controller structure

`Parallel` (default) | `Ideal`

Specify whether the controller structure is parallel or ideal.

`Parallel`

The controller output is the sum of the proportional, integral, and derivative actions, weighted independently by **P**, **I**, and **D**, respectively. For example, for a continuous-time parallel-form PID controller, the transfer function is:

$$C_{par}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right).$$

For a discrete-time parallel-form controller, the transfer function is:

$$C_{par}(z) = P + I\alpha(z) + D\left[\frac{N}{1+N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

`Ideal`

The proportional gain **P** acts on the sum of all actions. For example, for a continuous-time ideal-form 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-form controller, the transfer function is:

$$C_{id}(z) = P\left[1 + I\alpha(z) + D\frac{N}{1+N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $a(z)$ and $b(z)$, respectively.

Tip The controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use
Block Parameter: `Controller`
Type: string, character vector
Values: `"Parallel"`, `"Ideal"`
Default: `"Parallel"`

`Time domain` — Specify continuous-time or discrete-time controller

`Continuous-time` (default) | `Discrete-time`

When you select `Discrete-time`, it is recommended that you specify an explicit sample time for the block. See the **Sample time (-1 for inherited)** parameter. Selecting `Discrete-time` also enables the **Integrator method**, and **Filter method** parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select `Continuous-time`.

Note The PID Controller and Discrete PID Controller blocks are identical except for the default value of this parameter.

Programmatic Use
Block Parameter: `TimeDomain`
Type: string, character vector
Values: `"Continuous-time"`, `"Discrete-time"`
Default: `"Continuous-time"`

`Sample time (-1 for inherited)` — Discrete interval between samples

–1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of –1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the

controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.

To implement a continuous-time controller, set **Time domain** to `Continuous-time`.

Tip If you want to run the block with an externally specified or variable sample time, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use

Block Parameter: `SampleTime`
Type: scalar
Values: `-1`, positive scalar
Default: `-1`

`Integrator method` — Method for computing integral in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the integral term of the controller transfer function is $I\alpha(z)$, where $\alpha(z)$ depends on the integrator method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z) = \frac{T_s}{z-1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z) = \frac{T_s z}{z-1}.$$

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`

Bilinear approximation,

$$\alpha(z) = \frac{T_s}{2}\frac{z+1}{z-1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

Note For the `BackwardEuler` or `Trapezoidal` methods, you cannot generate HDL code for the block if either:

- **Limit output** is selected and **Anti-Windup Method** is anything other than `none`.
- **Enable tracking mode** is selected.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with integral action.

Programmatic Use

Block Parameter: `IntegratorMethod`

Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

`Filter method` — Method for computing derivative in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the derivative term of the controller transfer function is:

$$D\left[\frac{N}{1+N\alpha(z)}\right],$$

where $\alpha(z)$ depends on the filter method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z)=\frac{T_s}{z-1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z)=\frac{T_s z}{z-1}.$$

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`

Bilinear approximation,

$$\alpha(z)=\frac{T_s}{2}\frac{z+1}{z-1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with derivative action.

Programmatic Use

Block Parameter: `FilterMethod`
Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

Main

Source — Source for controller gains and filter coefficient

internal (default) | external

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.

`internal`

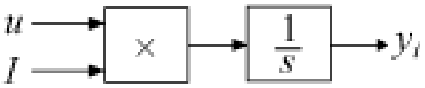
Specify the controller gains and filter coefficient using the block parameters **P**, **I**, **D**, and **N**.

`external`

Specify the PID gains and filter coefficient externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

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 gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. 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 input signal *u* 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.

Programmatic Use
Block Parameter: `ControllerParametersSource`
Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Proportional (P)` — Proportional gain

1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- `Parallel` — Proportional action is independent of the integral and derivative actions. For instance, for a continuous-time parallel PID controller, the transfer function is:

$$C_{par}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right).$$

For a discrete-time parallel-form controller, the transfer function is:

$$C_{par}(z) = P + I\alpha(z) + D\left[\frac{N}{1+N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

- `Ideal` — The proportional gain multiples the integral and derivative terms. For instance, 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-form controller, the transfer function is:

  $$C_{id}(z) = P\left[1 + I\alpha(z) + D\frac{N}{1+N\beta(z)}\right],$$

  where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to `PID`, `PD`, `PI`, or `P`.

Programmatic Use

Block Parameter: `P`
Type: scalar, vector
Default: 1

`Integral (I)` — Integral gain

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `I`
Type: scalar, vector
Default: 1

`Derivative (D)` — Derivative gain

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to `PID` or `PD`.

Programmatic Use

Block Parameter: `D`
Type: scalar, vector
Default: 0

`Use filtered derivative` — Apply filter to derivative term

`on` (default) | `off`

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller transfer function becomes:

$$D\frac{z-1}{zT_s}.$$

For continuous-time PID controllers, the derivative term is always filtered.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`, and set **Controller** to a type that has derivative action.

Programmatic Use

Block Parameter: `UseFilter`
Type: string, character vector
Values: `"on"`, `"off"`
Default: `"on"`

`Filter coefficient (N)` — Derivative filter coefficient

100 (default) | scalar | vector

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the **Time domain** parameter.

- When **Time domain** is `Continuous-time`, the pole location is `s = -N`.
- When **Time domain** is `Discrete-time`, the pole location depends on the **Filter method** parameter.

Filter Method	Location of Filter Pole
`Forward Euler`	$z_{pole} = 1 - NT_s$
`Backward Euler`	$z_{pole} = \frac{1}{1 + NT_s}$
`Trapezoidal`	$z_{pole} = \frac{1 - NT_s / 2}{1 + NT_s / 2}$

The block does not support `N = Inf` (ideal unfiltered derivative). When the **Time domain** is `Discrete-time`, you can clear **Use filtered derivative** to remove the derivative filter.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to `PID` or `PD`.

Programmatic Use
Block Parameter: `N`
Type: scalar, vector
Default: `100`

`Select tuning method` — Tool for automatic tuning of controller coefficients

`Transfer Function Based (PID Tuner App)` (default) | `Frequency Response Based`

If you have Simulink Control Design software, you can automatically tune the PID coefficients. To do so, use this parameter to select a tuning tool, and click **Tune**.

`Transfer Function Based (PID Tuner App)`

> Use **PID Tuner**, which lets you interactively tune PID coefficients 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 coefficients 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`

> Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to zero. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

`Enable zero-crossing detection` — Detect zero crossings on reset and on entering or leaving a saturation state

`on` (default) | `off`

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 **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

Programmatic Use
Block Parameter: `ZeroCross`
Type: string, character vector
Values: `"on"`, `"off"`
Default: `"on"`

Initialization

`Source` — Source for integrator and derivative initial conditions

`internal` (default) | `external`

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event. (See the **External reset** parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.

`internal`

Specify the initial conditions using the **Integrator Initial condition** and **Filter Initial condition** parameters. If **Use filtered derivative** is not selected, use the **Differentiator** parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

`external`

Specify the initial conditions externally using block inputs. Additional input ports $\mathbf{I_o}$ and $\mathbf{D_o}$ appear on the block. If **Use filtered derivative** is not selected, supply the initial condition for the unfiltered differentiator at $\mathbf{D_o}$ instead of a filter initial condition.

Programmatic Use
Block Parameter: `InitialConditionSource`
Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Integrator` — Integrator initial condition

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `InitialConditionForIntegrator`
Type: scalar, vector
Default: 0

`Filter` — Filter initial condition

0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and use a controller that has a derivative filter.

Programmatic Use

Block Parameter: `InitialConditionForFilter`
Type: scalar, vector
Default: 0

`Differentiator` — Initial condition for unfiltered derivative

0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see **External**

reset). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, set **Time domain** to `Discrete-time`, clear the **Use filtered derivative** check box, and in the **Initialization** tab, set **Source** to `internal`.

Programmatic Use

Block Parameter: `DifferentiatorICPrevScaledInput`
Type: scalar, vector
Default: 0

`Initial condition setting` — Location at which initial condition is applied

`State (most efficient)` (default) | `Output`

Use this parameter to specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. You can change this parameter at the command line only, using `set_param` to set the `InitialConditionSetting` parameter of the block.

`State (most efficient)`

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** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

Programmatic Use

Block Parameter: `InitialConditionSetting`
Type: string, character vector
Values: `"state"`, `"output"`
Default: `"state"`

`External reset` — Trigger for resetting integrator and filter values

`none` (default) | `rising` | `falling` | `either` | `level`

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than `none` enables the **Reset** port on the block for the external reset signal.

`none`

The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

`rising`

Reset the outputs when the reset signal has a rising edge.

`falling`

Reset the outputs when the reset signal has a falling edge.

`either`

Reset the outputs when the reset signal either rises or falls.

`level`

Reset the outputs when the reset signal either:

- Is nonzero at the current time step
- Changes from nonzero at the previous time step to zero at the current time step

This option holds the outputs to the initial conditions while the reset signal is nonzero.

Dependencies

To enable this parameter, set **Controller** to a type that has derivative or integral action.

Programmatic Use

Block Parameter: `ExternalReset`
Type: string, character vector
Values: `"none"`, `"rising"`, `"falling"`, `"either"`,`"level"`
Default: `"none"`

`Ignore reset when linearizing` — Force linearization to ignore reset

`off` (default) | `on`

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the **External reset** parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

Programmatic Use

Block Parameter: `IgnoreLimit`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

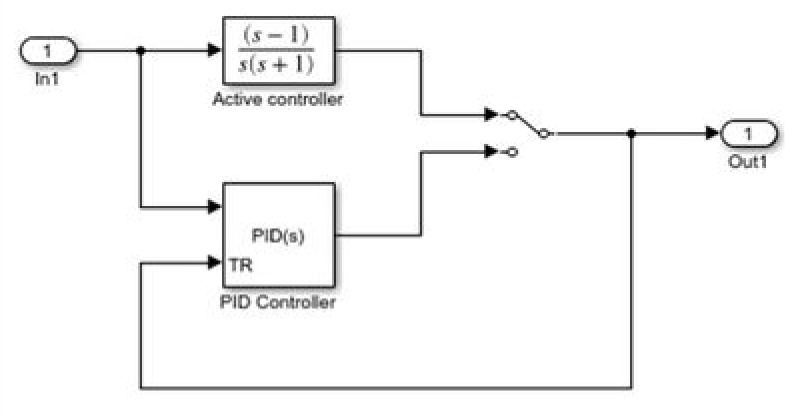
`Enable tracking mode` — Activate signal tracking

`off` (default) | `on`

Signal tracking lets the block output follow a tracking signal that you provide at the **TR** port. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input with a gain `Kt`, specified by the **Tracking gain (Kt)** parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.

Bumpless control transfer

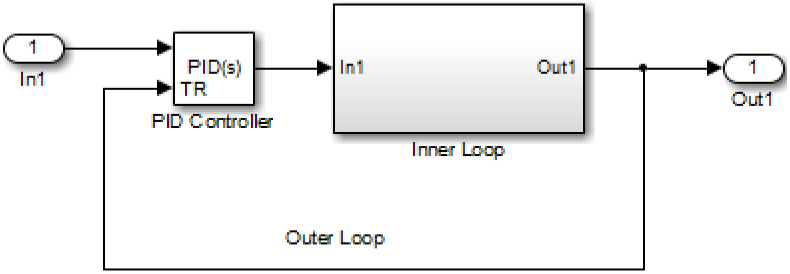
Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and another controller. To do so, connecting the controller output to the **TR** input as shown in the following illustration.



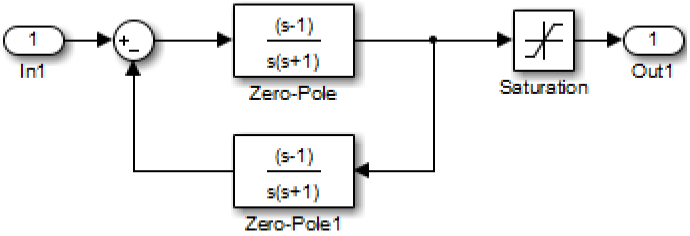
For more information, see "Bumpless Control Transfer" on page 14-111.

Multiloop control

Use signal tracking to prevent block windup in multiloop control approaches, as in the following model.



The Inner Loop subsystem contains the blocks shown in the following diagram.



Because the PID controller tracks the output of the inner loop, its output never exceeds the saturated inner-loop output. For more details, see “Prevent Block Windup in Multiloop Control” on page 14-110.

Dependencies

To enable this parameter, set **Controller** to a type that has integral action.

Programmatic Use
Block Parameter: `TrackingMode`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Tracking coefficient (Kt)` — Gain of signal-tracking feedback loop
1 (default) | scalar

When you select **Enable tracking mode**, the difference between the signal **TR** and the block output is fed back to the integrator input with a gain `Kt`. Use this parameter to specify the gain in that feedback loop.

Dependencies

To enable this parameter, select **Enable tracking mode**.

Programmatic Use

Block Parameter: `Kt`
Type: scalar
Default: 1

Output saturation

`Limit Output` — Limit block output to specified saturation values

`off` (default) | `on`

Activating this option limits the block output internally to the block, so that you do not need a separate Saturation on page 1-1607 block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the **Anti-windup method** parameter). Specify the saturation limits using the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use

Block Parameter: `LimitOutput`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Upper limit` — Upper saturation limit for block output

`Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use

Block Parameter: `UpperSaturationLimit`
Type: scalar
Default: `Inf`

`Lower limit` — Lower saturation limit for block output

`-Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use

Block Parameter: `LowerSaturationLimit`
Type: scalar
Default: `-Inf`

`Ignore saturation when linearizing` — Force linearization to ignore output limits

`off` (default) | `on`

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the **Upper limit** and **Lower limit** parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use

Block Parameter: `LinearizeAsGain`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Anti-windup method` — Integrator anti-windup method

`none` (default) | `back-calculation` | `clamping`

When you select **Limit output** 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 windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, 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 unwind the integrator and return the weighted sum within the block saturation limit.

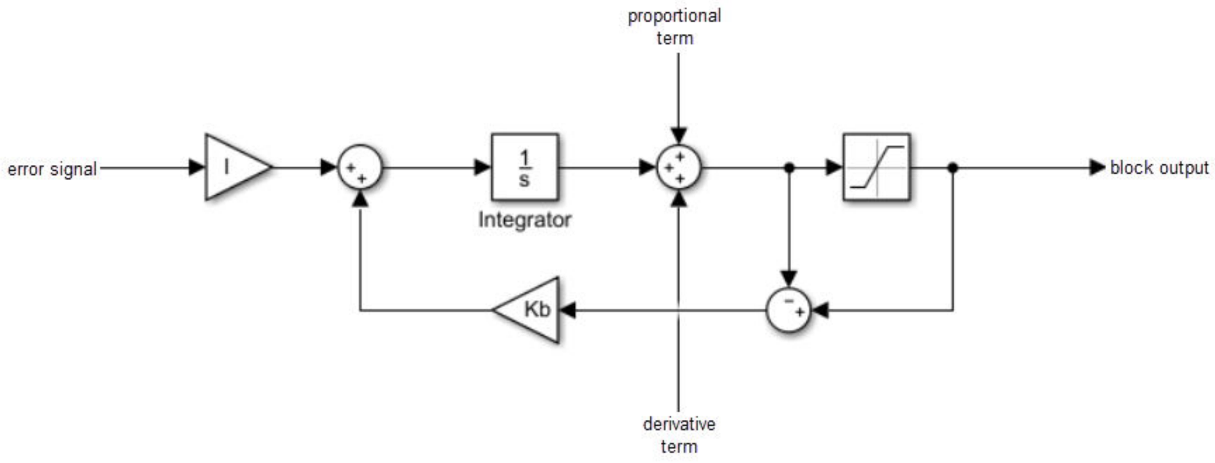
In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.

`none`

Do not use an anti-windup mechanism.

`back-calculation`

Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select **Mask > Look Under Mask**.



Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set `Kb = I`, or for controllers with derivative action, `Kb = sqrt(I*D)`. Back-calculation can be effective for plants with relatively large dead time [1].

`clamping`

Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use
Block Parameter: `AntiWindupMode`
Type: string, character vector
Values: `"none"`, `"back-calculation"`,`"clamping"`
Default: `"none"`

`Back-calculation coefficient (Kb)` — Gain coefficient of anti-windup feedback loop

1 (default) | scalar

The `back-calculation` anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. For more information, see the **Anti-windup method** parameter.

Dependencies

To enable this parameter, select the **Limit output** parameter, and set the **Anti-windup method** parameter to `back-calculation`.

Programmatic Use
Block Parameter: `Kb`
Type: scalar

Default: 1

Data Types

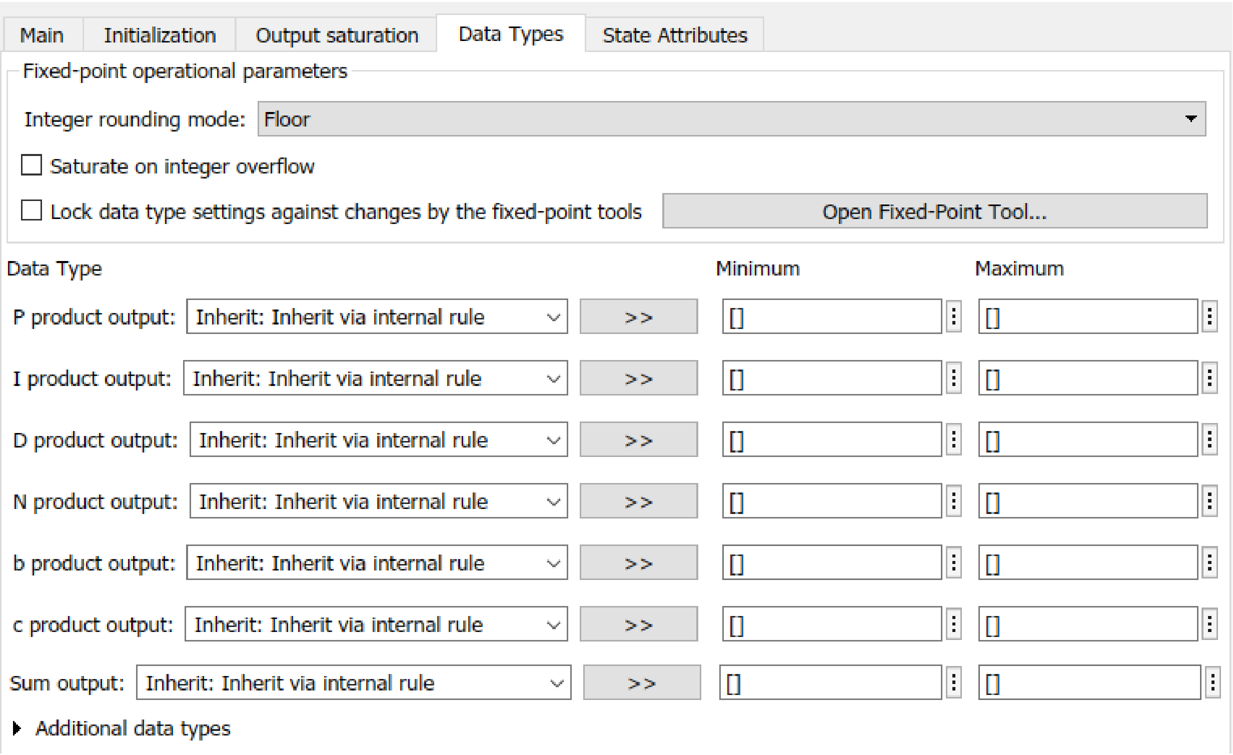
The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click **Open Fixed-Point Tool** and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see "Autoscaling Data Objects Using the Fixed-Point Tool" (Fixed-Point Designer).

After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.

For assistance in selecting appropriate values, click >> to open the Data Type Assistant for the corresponding quantity. For more information, see "Specify Data Types Using Data Type Assistant".



Fixed-point operational parameters

Integer rounding mode: Floor

□ Saturate on integer overflow

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

Data Type		Minimum	Maximum
P product output:	Inherit: Inherit via internal rule	[]	[]
I product output:	Inherit: Inherit via internal rule	[]	[]
D product output:	Inherit: Inherit via internal rule	[]	[]
N product output:	Inherit: Inherit via internal rule	[]	[]
b product output:	Inherit: Inherit via internal rule	[]	[]
c product output:	Inherit: Inherit via internal rule	[]	[]
Sum output:	Inherit: Inherit via internal rule	[]	[]

▸ Additional data types

The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- Product output — Stores the result of a multiplication carried out under the block mask. For example, **P product output** stores the output of the gain block that multiplies the block input with the proportional gain **P**.
- Parameter — Stores the value of a numeric block parameter, such as **P**, **I**, or **D**.
- Block output — Stores the output of a block that resides under the PID controller block mask. For example, use **Integrator output** to specify the data type of the output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.
- Accumulator — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block

SumI2. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as `SumI2`. (See “Search and Edit Using Model Explorer”.)

Matching Input and Internal Data Types

By default, all data types in the block are set to `Inherit: Inherit via internal rule`. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type `double`. If the block input signal is a type that cannot be converted to `double`, such as `uint16`, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to `double`, ensuring that the signals reaching the continuous-time integrators are of type `double`.

In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

Fixed-Point Operational Parameters

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector

Values: `'off' | 'on'`
Default: `'off'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

The parameters in this tab are primarily of use in code generation.

`State name (e.g., 'position')` — Name for continuous-time filter and integrator states

`''` (default) | character vector

Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the **State name** parameter.) The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

Dependencies

To enable this parameter, set **Time domain** to `Continuous-time`.

Programmatic Use
Parameter: `IntegratorContinuousStateAttributes`, `FilterContinuousStateAttributes`
Type: character vector
Default: `''`

`State name` — Names for discrete-time filter and integrator states

empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the **State name (e.g., 'position')** parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use
Parameter: `IntegratorStateIdentifier`, `FilterStateIdentifier`
Type: string, character vector
Default: `""`

`State name must resolve to Simulink signal object` — Require that state name resolve to a signal object

`off` (default) | `on`

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class** for the corresponding integrator or filter state.

Programmatic Use
Block Parameter: `IntegratorStateMustResolveToSignalObject`, `FilterStateMustResolveToSignalObject`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Code generation storage class` — Storage class for code generation

`Auto` (default) | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer`

Select state storage class for code generation. If you do not need to interface to external code, select `Auto`.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Programmatic Use
Block Parameter: `IntegratorRTWStateStorageClass`, `FilterRTWStateStorageClass`
Type: string, character vector
Values: `"Auto"`, `"ExportedGlobal"`, `"ImportedExtern"` | `"ImportedExternPointer"`
Default: `"Auto"`

`Code generation storage type qualifier` — Storage type qualifier

empty string (default) | character vector | `"const"` | `"volatile"` | ...

Specify a storage type qualifier such as `const` or `volatile`.

Note This parameter 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.

Dependencies

To enable this parameter, set **Code generation storage class** to any value other than `Auto`.

Programmatic Use
Block Parameter: `IntegratorRTWStateStorageTypeQualifier`,`FilterRTWStateStorageTypeQualifier`
Type: string, character vector
Values:`""`,`"const"`,`"volatile"`
Default: `""`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

References

[1] Visioli, A., "Modified Anti-Windup Scheme for PID Controllers," *IEE Proceedings - Control Theory and Applications*, Vol. 150, Number 1, January 2003

Extended Capabilities

C/C++ Code Generation

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

For continuous-time PID controllers (**Time domain** set to `Continuous-time`):

- Consider using "Model Discretizer" to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, from your model, select **Analysis** > **Control Design** > **Analysis** > **Model Discretizer**.
- Not recommended for production code.

For discrete-time PID controllers (**Time domain** set to `Discrete-time`):

- Depends on absolute time when placed inside a triggered subsystem hierarchy.
- Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

HDL Code Generation

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

HDL code generation is supported for discrete-time PID controllers only (**Time domain** set to `Discrete-time`).

If the **Integrator method** is set to `BackwardEuler` or `Trapezoidal`, you cannot generate HDL code for the block under either of the following conditions:

- **Limit output** is selected and the **Anti-Windup Method** is anything other than `none`.
- **Enable tracking mode** is selected.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (**Time domain** set to `Discrete-time`).

See Also

Derivative | Discrete PID Controller | Gain | Integrator | PID Controller (2DOF)

Introduced in R2009b

PID Controller (2DOF)

Continuous-time or discrete-time two-degree-of-freedom PID controller
Library: Simulink / Continuous

Description

The PID Controller (2DOF) block implements a two-degree-of-freedom PID controller (PID, PI, or PD). The block is identical to the Discrete PID Controller (2DOF) block with the **Time domain** parameter set to `Continuous-time`.

The 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 the proportional and derivative actions according to the setpoint weights (**b** and **c**) that 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 **P**, **I**, and **D**. A first-order pole filters the derivative action.

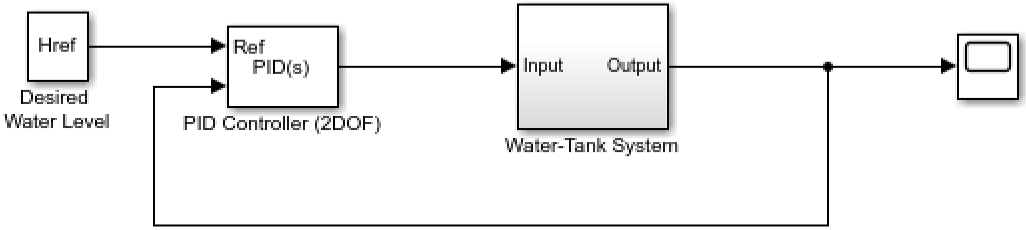
The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, or PD) — See the **Controller** parameter.
- Controller form (Parallel or Ideal) — See the **Form** parameter.
- Time domain (continuous or discrete) — See the **Time domain** parameter.
- Initial conditions and reset trigger — See the **Source** and **External reset** parameters.
- Output saturation limits and built-in anti-windup mechanism — See the **Limit output** parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the **Enable tracking mode** parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (See “Variant Subsystems”.) To examine the internal structure of the block and its variant subsystems, right-click the block and select **Mask > Look Under Mask**.

Control Configuration

In one common implementation, the PID Controller block operates in the feedforward path of a feedback loop.



For a single-input block that accepts an error signal (a difference between a setpoint and a system output), see PID Controller.

PID Gain Tuning

The PID controller coefficients and the setpoint weights are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software. For more information about automatic tuning, see the **Select tuning method** parameter.

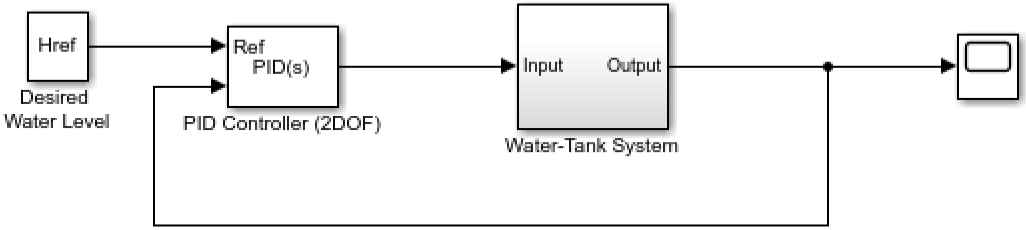
Ports

Input

`Ref` — Reference signal

scalar | vector

Reference signal for plant to follow, as shown.



When the reference signal is a vector, the block acts separately on each signal, vectorizing the PID coefficients and producing a vector output signal of the same dimensions. You can specify the PID coefficients and some other parameters as vectors of the same dimensions as the input signal. Doing so is equivalent to specifying a separate PID controller for each entry in the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_1( y )` — Measured system output

scalar | vector

Feedback signal for the controller, from the plant output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

P — Proportional gain

scalar | vector

Proportional gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`I` — Integral gain

scalar | vector

Integral gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`D` — Derivative gain

scalar | vector

Derivative gain, provided from a source external to the block. 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 gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`N` — Filter coefficient

scalar | vector

Derivative filter coefficient, provided from a source external to the block. External coefficient 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has a filtered derivative.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

b — Proportional setpoint weight

scalar | vector

Proportional setpoint weight, provided from a source external to the block. External 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set **Controller parameters Source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

c — Derivative setpoint weight

scalar | vector

Derivative setpoint weight, provided from a source external to the block. External 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 the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

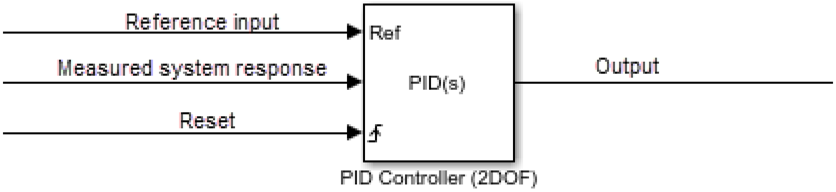
To enable this port, set **Controller parameters Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Reset — External reset trigger

scalar

Trigger to reset the integrator and filter to their initial conditions. Use the **External reset** parameter to specify what kind of signal triggers a reset. The port icon indicates the trigger type specified in that parameter. For example, the following illustration shows a continuous-time PID Controller (2DOF) block with **External reset** set to `rising`.



When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the **Integrator Initial condition** and **Filter Initial condition** parameters or the $\mathbf{I_0}$ and $\mathbf{D_0}$ ports.

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.

Dependencies

To enable this port, set **External reset** to any value other than `none`.

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

I_0 — Integrator initial condition

scalar | vector

Integrator initial condition, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has integral action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

D_0 — Filter initial condition

scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to `external`, and set **Controller** to a controller type that has derivative action.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

TR — Tracking signal

scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input. Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the **Enable tracking mode** parameter.

Dependencies

To enable this port, select the **Enable tracking mode** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1( u )` — Controller output

scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the setpoint weights and by the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the **Controller** parameter. The base controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the **Upper Limit** and **Lower Limit** saturation parameters.

The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as *N* independent PID controllers, where *N* is the number of signals in the input vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Controller` — Controller type

`PID` (default) | `PI` | `PD`

Specify which of the proportional, integral, and derivative terms are in the controller.

`PID`

Proportional, integral, and derivative action.

`PI`

Proportional and integral action only.

`PD`

Proportional and derivative action only.

Tip The controller output for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use
Block Parameter: `Controller`
Type: string, character vector
Values: `"PID"`, `"PI"`, `"PD"`
Default: `"PID"`

`Form` — Controller structure

`Parallel` (default) | `Ideal`

Specify whether the controller structure is parallel or ideal.

`Parallel`

The proportional, integral, and derivative gains **P**, **I**, and **D**, are applied independently. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output *u* is:

$$u = P\left(br - y\right) + I\frac{1}{s}\left(r - y\right) + D\frac{N}{1 + N\frac{1}{s}}\left(cr - y\right),$$

where *r* is the reference signal, *y* is the measured plant output signal, and *b* and *c* are the setpoint weights.

For a discrete-time 2-DOF controller in parallel form, the controller output is:

$$u = P\left(br - y\right) + I\alpha\left(z\right)\left(r - y\right) + D\frac{N}{1 + N\beta\left(z\right)}\left(cr - y\right),$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

`Ideal`

The proportional gain **P** acts on the sum of all actions. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$u = P\left[\left(br - y\right) + I\frac{1}{s}\left(r - y\right) + D\frac{N}{1 + N\frac{1}{s}}\left(cr - y\right)\right].$$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:

$$u = P\left[\left(br - y\right) + I\alpha\left(z\right)\left(r - y\right) + D\frac{N}{1 + N\beta\left(z\right)}\left(cr - y\right)\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Tip The controller output for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask.

Programmatic Use
Block Parameter: `Controller`
Type: string, character vector
Values: `"Parallel"`, `"Ideal"`
Default: `"Parallel"`

`Time domain` — Specify continuous-time or discrete-time controller

`Continuous-time` (default) | `Discrete-time`

When you select `Discrete-time`, it is recommended that you specify an explicit sample time for the block. See the **Sample time (-1 for inherited)** parameter. Selecting `Discrete-time` also enables the **Integrator method**, and **Filter method** parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select `Continuous-time`.

Note The PID Controller (2DOF) and Discrete PID Controller (2DOF) blocks are identical except for the default value of this parameter.

Programmatic Use
Block Parameter: `TimeDomain`
Type: string, character vector
Values: `"Continuous-time"`, `"Discrete-time"`
Default: `"Continuous-time"`

`Sample time (-1 for inherited)` — Discrete interval between samples

–1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of –1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the

controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.

To implement a continuous-time controller, set **Time domain** to `Continuous-time`.

Tip If you want to run the block with an externally specified or variable sample time, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use
Block Parameter: `SampleTime`
Type: scalar
Values: `-1`, positive scalar
Default: `-1`

`Integrator method` — Method for computing integral in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the integral term of the controller transfer function is *Ia*(*z*), where *a*(*z*) depends on the integrator method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z) = \frac{T_s}{z-1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z) = \frac{T_s z}{z-1}.$$

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`

Bilinear approximation,

$$\alpha(z) = \frac{T_s}{2}\frac{z+1}{z-1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with integral action.

Programmatic Use

Block Parameter: `IntegratorMethod`
Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

`Filter method` — Method for computing derivative in discrete-time controller

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

In discrete time, the derivative term of the controller transfer function is:

$$D\left[\frac{N}{1+N\alpha(z)}\right],$$

where $\alpha(z)$ depends on the filter method you specify with this parameter.

`Forward Euler`

Forward rectangular (left-hand) approximation,

$$\alpha(z)=\frac{T_s}{z-1}.$$

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`

Backward rectangular (right-hand) approximation,

$$\alpha(z)=\frac{T_s z}{z-1}.$$

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`

Bilinear approximation,

$$\alpha(z)=\frac{T_s}{2}\frac{z+1}{z-1}.$$

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.

Tip The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

Dependencies

To enable this parameter, set **Time Domain** to `Discrete-time` and set **Controller** to a controller type with derivative action.

Programmatic Use
Block Parameter: `FilterMethod`
Type: string, character vector
Values: `"Forward Euler"`, `"Backward Euler"`, `"Trapezoidal"`
Default: `"Forward Euler"`

Main

`Source` — Source for controller gains and filter coefficient

internal (default) | external

`internal`

Specify the controller gains, filter coefficient, and setpoint weights using the block parameters **P**, **I**, **D**, **N**, **b**, and **c** respectively.

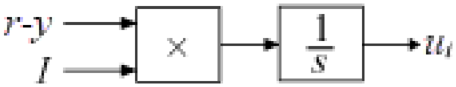
`external`

Specify the PID gains, filter coefficient, and setpoint weights externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

Enabling external inputs for the parameters allows you to compute their values externally to the block and provide them to the block as signal inputs.

External 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 gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. The derivative setpoint weight *c* is also differentiated. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. 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 input signal *u* is multiplied by the externally supplied integrator gain, *I*, before integration. This implementation yields:

$$u_i = \int (r-y) I \, dt.$$

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* and the derivative setpoint weight *c* to be differentiated.

Programmatic Use
Block Parameter: `ControllerParametersSource`
Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Proportional (P)` — Proportional gain

1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- `Parallel` — Proportional action is independent of the integral and derivative actions. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output *u* is:

  $$u = P(br-y) + I\frac{1}{s}(r-y) + D\frac{N}{1+N\frac{1}{s}}(cr-y),$$

  where *r* is the reference signal, *y* is the measured plant output signal, and *b* and *c* are the setpoint weights.

  For a discrete-time 2-DOF controller in parallel form, the controller output is:

  $$u = P(br-y) + I\alpha(z)(r-y) + D\frac{N}{1+N\beta(z)}(cr-y),$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

- `Ideal` — The proportional gain multiples the integral and derivative terms. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$u = P\left[(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y)\right].$$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:

$$u = P\left[(br - y) + I\alpha(z)(r - y) + D\frac{N}{1 + N\beta(z)}(cr - y)\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Tunable: Yes

Dependencies

To enable this parameter, set the Controller parameters **Source** to `internal`.

Programmatic Use

Block Parameter: `P`
Type: scalar, vector
Default: 1

`Integral (I)` — Integral gain

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `I`
Type: scalar, vector
Default: 1

`Derivative (D)` — Derivative gain

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`, and set **Controller** to `PID` or `PD`.

Programmatic Use

Block Parameter: `D`
Type: scalar, vector
Default: 0

`Use filtered derivative` — Apply filter to derivative term

`on` (default) | `off`

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller output becomes:

$$D\frac{z-1}{zT_s}(cr-y).$$

For continuous-time PID controllers, the derivative term is always filtered.

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`, and set **Controller** to a type that has a derivative term.

Programmatic Use

Block Parameter: `UseFilter`
Type: string, character vector
Values: `"on"`, `"off"`

Default: `"on"`

`Filter coefficient (N)` — Derivative filter coefficient

100 (default) | scalar | vector

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the **Time domain** parameter.

- When **Time domain** is `Continuous-time`, the pole location is `s = -N`.
- When **Time domain** is `Discrete-time`, the pole location depends on the **Filter method** parameter.

Filter Method	Location of Filter Pole
`Forward Euler`	$z_{pole} = 1 - NT_s$
`Backward Euler`	$z_{pole} = \frac{1}{1 + NT_s}$
`Trapezoidal`	$z_{pole} = \frac{1 - NT_s / 2}{1 + NT_s / 2}$

The block does not support `N = Inf` (ideal unfiltered derivative). When the **Time domain** is `Discrete-time`, you can clear **Use filtered derivative** to remove the derivative filter.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to `PID` or `PD`.

Programmatic Use

Block Parameter: `N`
Type: scalar, vector
Default: 100

`Setpoint weight (b)` — Proportional setpoint weight

1 (default) | scalar | vector

Setpoint weight on the proportional term of the controller. The proportional term of a 2-DOF controller output is *P*(*br*–*y*), where *r* is the reference signal and *y* is the measured plant output. Setting *b* to 0 eliminates proportional action on the reference signal, which can reduce overshoot in the system response to step changes in the setpoint. Changing the relative values of *b* and *c* changes the balance between disturbance rejection and setpoint tracking.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`.

Programmatic Use

Block Parameter: `b`
Type: scalar, vector
Default: 1

`Setpoint weight (c)` — Derivative setpoint weight

1 (default) | scalar | vector

Setpoint weight on the derivative term of the controller. The derivative term of a 2-DOF controller acts on *cr*–*y*, where *r* is the reference signal and *y* is the measured plant output. Thus, setting *c* to 0 eliminates derivative action on the reference signal, which can reduce transient response to step changes in the setpoint. Setting *c* to 0 can yield a controller that achieves both effective disturbance rejection and smooth setpoint tracking without excessive transient response. Changing the relative values of *b* and *c* changes the balance between disturbance rejection and setpoint tracking.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal` and set **Controller** to a type that has derivative action.

Programmatic Use

Block Parameter: `c`
Type: scalar, vector
Default: 1

`Select tuning method` — Tool for automatic tuning of controller coefficients

`Transfer Function Based (PID Tuner App)` (default) | `Frequency Response Based`

If you have Simulink Control Design software, you can automatically tune the PID coefficients when they are internal to the block. To do so, use this parameter to select a tuning tool, and click **Tune**.

`Transfer Function Based (PID Tuner App)`

Use **PID Tuner**, which lets you interactively tune PID coefficients while examining relevant system responses to validate performance. **PID Tuner** can tune all the coefficients **P**, **I**, **D**, and **N**, and the setpoint coefficients **b** and **c**. By default, **PID Tuner** works with a linearization of your plant model. For models that cannot be linearized, you can tune PID coefficients against a plant model estimated from simulated or measured response data. For more information, see “Design Two-Degree-of-Freedom PID Controllers” (Simulink Control Design).

`Frequency Response Based`

Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to zero. **Frequency Response Based PID Tuner** tunes the coefficients **P**, **I**, **D**, and **N**, but does not tune the setpoint coefficients **b** and **c**. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to `internal`.

`Enable zero-crossing detection` — Detect zero crossings on reset and on entering or leaving a saturation state

on (default) | `off`

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 **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

Programmatic Use
Block Parameter: `ZeroCross`
Type: string, character vector
Values: `"on"`, `"off"`
Default: `"on"`

Initialization

`Source` — Source for integrator and derivative initial conditions

`internal` (default) | `external`

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event. (See the **External reset** parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.

`internal`

Specify the initial conditions using the **Integrator Initial condition** and **Filter Initial condition** parameters. If **Use filtered derivative** is not selected, use the **Differentiator** parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

`external`

Specify the initial conditions externally using block inputs. Additional input ports $\mathbf{I_o}$ and $\mathbf{D_o}$ appear on the block. If **Use filtered derivative** is not selected, supply the initial condition for the unfiltered differentiator at $\mathbf{D_o}$ instead of a filter initial condition.

Programmatic Use
Block Parameter: `InitialConditionSource`

Type: string, character vector
Values: `"internal"`, `"external"`
Default: `"internal"`

`Integrator` — Integrator initial condition

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `InitialConditionForIntegrator`
Type: scalar, vector
Default: 0

`Filter` — Filter initial condition

0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, in the **Initialization** tab, set **Source** to `internal`, and use a controller that has a derivative filter.

Programmatic Use

Block Parameter: `InitialConditionForFilter`
Type: scalar, vector
Default: 0

`Differentiator` — Initial condition for unfiltered derivative

0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be `NaN` or `Inf`.

Dependencies

To use this parameter, set **Time domain** to `Discrete-time`, clear the **Use filtered derivative** check box, and in the **Initialization** tab, set **Source** to `internal`.

Programmatic Use

Block Parameter: `DifferentiatorICPrevScaledInput`
Type: scalar, vector
Default: 0

`Initial condition setting` — Location at which initial condition is applied

`State (most efficient)` (default) | `Output`

Use this parameter to specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. You can change this parameter at the command line only, using `set_param` to set the `InitialConditionSetting` parameter of the block.

`State (most efficient)`

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** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

Programmatic Use

Block Parameter: `InitialConditionSetting`

Type: string, character vector
Values: `"state"`, `"output"`
Default: `"state"`

`External reset` — Trigger for resetting integrator and filter values

`none` (default) | `rising` | `falling` | `either` | `level`

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than `none` enables the **Reset** port on the block for the external reset signal.

`none`

The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

`rising`

Reset the outputs when the reset signal has a rising edge.

`falling`

Reset the outputs when the reset signal has a falling edge.

`either`

Reset the outputs when the reset signal either rises or falls.

`level`

Reset the outputs when the reset signal either:

- Is nonzero at the current time step
- Changes from nonzero at the previous time step to zero at the current time step

This option holds the outputs to the initial conditions while the reset signal is nonzero.

Dependencies

To enable this parameter, set **Controller** to a type that has derivative or integral action.

Programmatic Use

Block Parameter: `ExternalReset`
Type: string, character vector
Values: `"none"`, `"rising"`, `"falling"`, `"either"`,`"level"`
Default: `"none"`

`Ignore reset when linearizing` — Force linearization to ignore reset

`off` (default) | `on`

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the **External reset** parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

Programmatic Use

Block Parameter: `IgnoreLimit`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

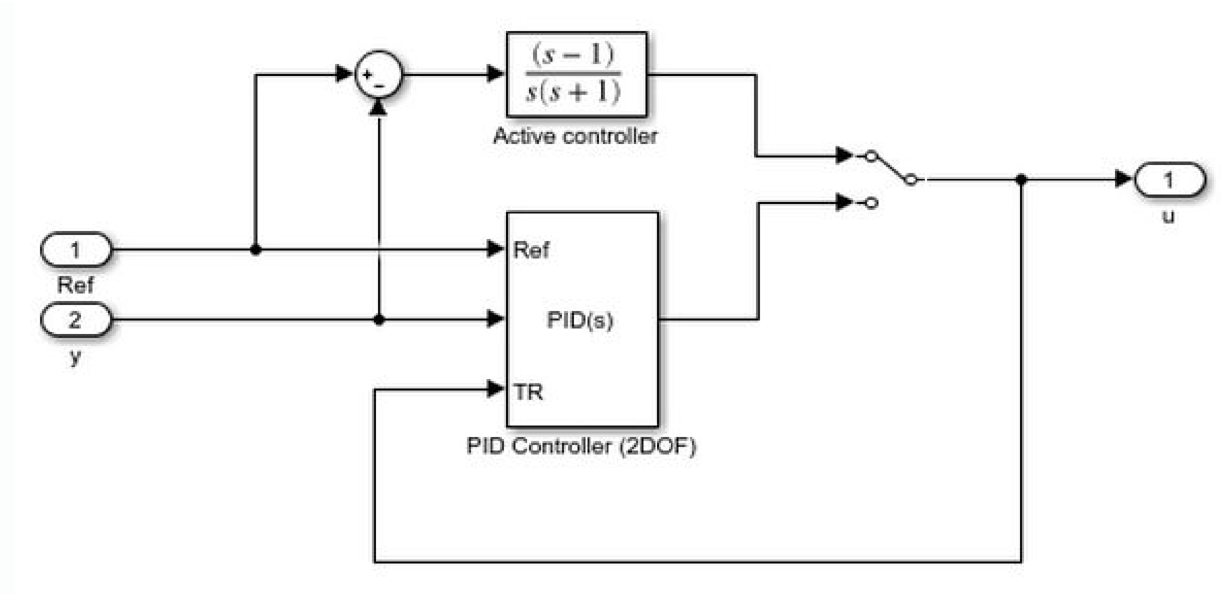
`Enable tracking mode` — Activate signal tracking

`off` (default) | `on`

Signal tracking lets the block output follow a tracking signal that you provide at the **TR** port. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input with a gain `Kt`, specified by the **Tracking gain (Kt)** parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.

Bumpless control transfer

Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and another controller. To do so, connecting the controller output to the **TR** input as shown in the following illustration.



For more information, see “Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller” on page 14-112.

Multiloop control

Use signal tracking to prevent block windup in multiloop control approaches. For an example illustrating this approach with a 1DOF PID controller, see “Prevent Block Windup in Multiloop Control” on page 14-110.

Dependencies

To enable this parameter, set **Controller** to a type that has integral action.

Programmatic Use

Block Parameter: `TrackingMode`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Tracking coefficient (Kt)` — Gain of signal-tracking feedback loop

1 (default) | scalar

When you select **Enable tracking mode**, the difference between the signal **TR** and the block output is fed back to the integrator input with a gain `Kt`. Use this parameter to specify the gain in that feedback loop.

Dependencies

To enable this parameter, select **Enable tracking mode**.

Programmatic Use
Block Parameter: `Kt`
Type: scalar
Default: 1

Output saturation

`Limit Output` — Limit block output to specified saturation values

`off` (default) | `on`

Activating this option limits the block output internally to the block, so that you do not need a separate Saturation on page 1-1607 block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the **Anti-windup method** parameter). Specify the saturation limits using the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use
Block Parameter: `LimitOutput`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Upper limit` — Upper saturation limit for block output

`Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use
Block Parameter: `UpperSaturationLimit`

Type: scalar
Default: `Inf`

`Lower limit` — Lower saturation limit for block output

`-Inf` (default) | scalar

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.

Dependencies

To enable this parameter, select **Limit output**.

Programmatic Use

Block Parameter: `LowerSaturationLimit`
Type: scalar
Default: `-Inf`

`Ignore saturation when linearizing` — Force linearization to ignore output limits

`off` (default) | `on`

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the **Upper limit** and **Lower limit** parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use

Block Parameter: `LinearizeAsGain`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Anti-windup method` — Integrator anti-windup method

`none` (default) | `back-calculation` | `clamping`

When you select **Limit output** 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 windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, two results are possible:

- If the sign of the signal entering the integrator 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 signal entering the integrator changes once the weighted sum has grown beyond the output limits, it can take a long time to unwind the integrator and return the weighted sum within the block saturation limit.

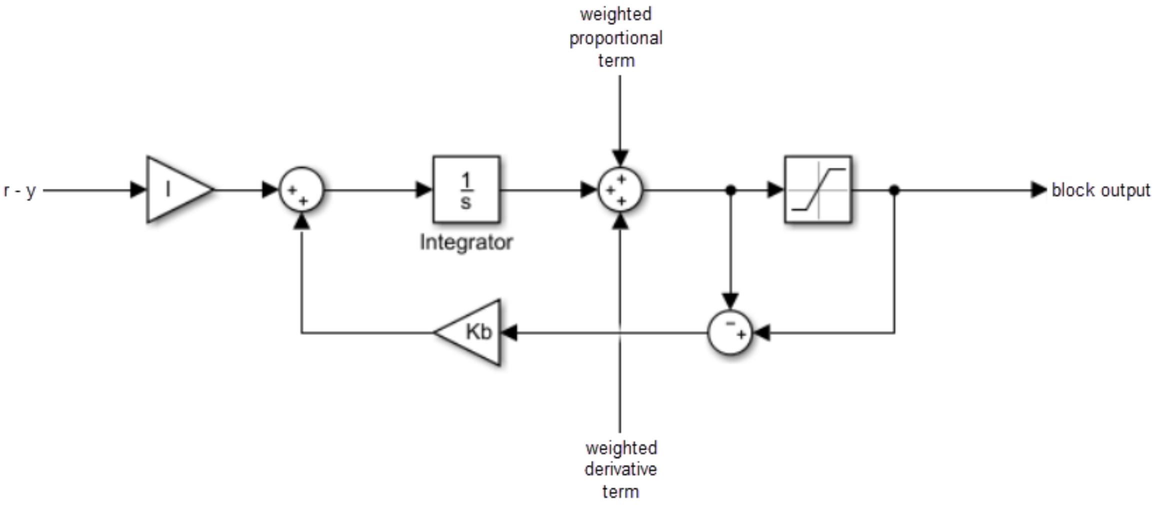
In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.

`none`

Do not use an anti-windup mechanism.

`back-calculation`

Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select **Mask > Look Under Mask**.



Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set `Kb = I`, or for controllers with derivative action, `Kb = sqrt(I*D)`. Back-calculation can be effective for plants with relatively large dead time [1].

`clamping`

Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].

Dependencies

To enable this parameter, select the **Limit output** parameter.

Programmatic Use

Block Parameter: `AntiWindupMode`
Type: string, character vector
Values: `"none"`, `"back-calculation"`,`"clamping"`

Default: "none"

Back-calculation coefficient (Kb) — Gain coefficient of anti-windup feedback loop

1 (default) | scalar

The back-calculation anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. For more information, see the **Anti-windup method** parameter.

Dependencies

To enable this parameter, select the **Limit output** parameter, and set the **Anti-windup method** parameter to back-calculation.

Programmatic Use

Block Parameter: Kb
Type: scalar
Default: 1

Data Types

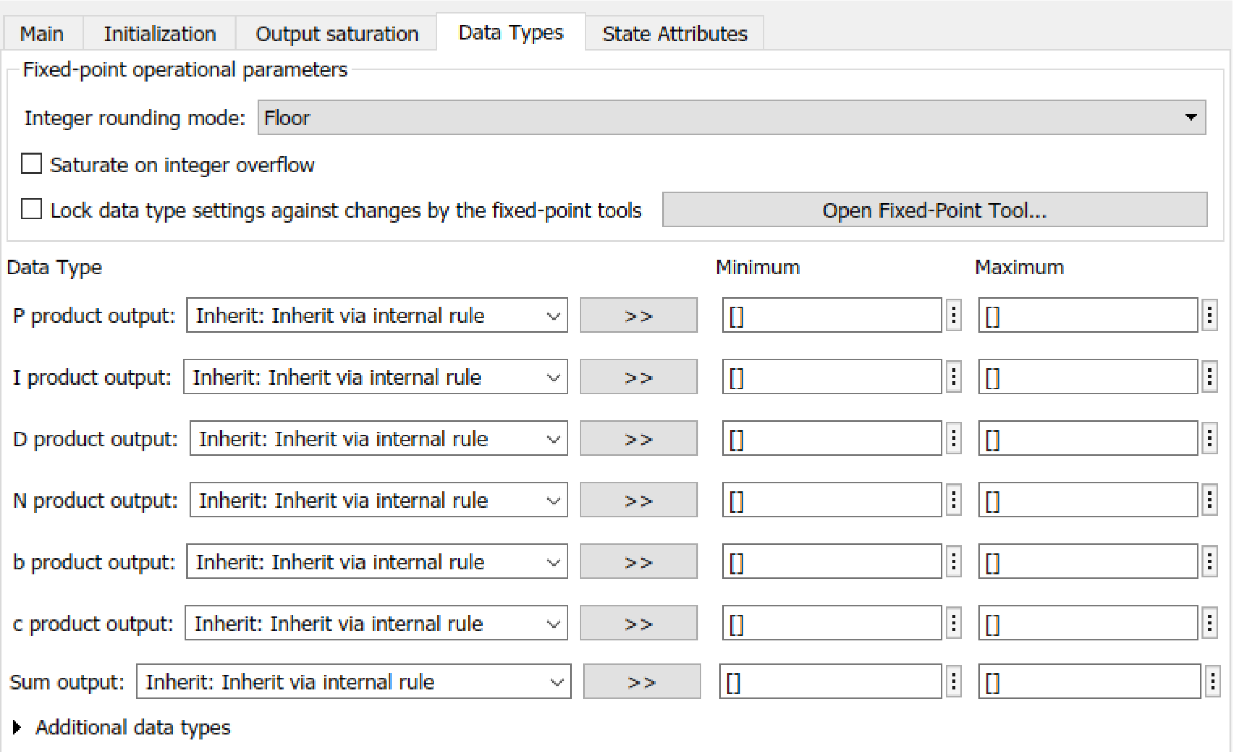
The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click **Open Fixed-Point Tool** and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).

After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.

For assistance in selecting appropriate values, click >> to open the Data Type Assistant for the corresponding quantity. For more information, see “Specify Data Types Using Data Type Assistant”.



Main | Initialization | Output saturation | Data Types | State Attributes

Fixed-point operational parameters

Integer rounding mode: Floor

□ Saturate on integer overflow

□ Lock data type settings against changes by the fixed-point tools [Open Fixed-Point Tool...]

Data Type			Minimum	Maximum
P product output:	Inherit: Inherit via internal rule	>>	[]	[]
I product output:	Inherit: Inherit via internal rule	>>	[]	[]
D product output:	Inherit: Inherit via internal rule	>>	[]	[]
N product output:	Inherit: Inherit via internal rule	>>	[]	[]
b product output:	Inherit: Inherit via internal rule	>>	[]	[]
c product output:	Inherit: Inherit via internal rule	>>	[]	[]
Sum output:	Inherit: Inherit via internal rule	>>	[]	[]

▸ Additional data types

The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- Product output — Stores the result of a multiplication carried out under the block mask. For example, **P product output** stores the output of the gain block that multiplies the block input with the proportional gain **P**.
- Parameter — Stores the value of a numeric block parameter, such as **P**, **I**, or **D**.
- Block output — Stores the output of a block that resides under the PID controller block mask. For example, use **Integrator output** to specify the data type of the

output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.

- Accumulator — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block SumI2. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as `SumI2`. (See "Search and Edit Using Model Explorer".)

Matching Input and Internal Data Types

By default, all data types in the block are set to `Inherit: Inherit via internal rule`. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type `double`. If the block input signal is a type that cannot be converted to `double`, such as `uint16`, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to `double`, ensuring that the signals reaching the continuous-time integrators are of type `double`.

In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

Fixed-Point Operational Parameters

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see "Lock the Output Data Type Setting" (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

State Attributes

The parameters in this tab are primarily of use in code generation.

`State name (e.g., 'position')` — Name for continuous-time filter and integrator states

`''` (default) | character vector

Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the **State name** parameter.) The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

Dependencies

To enable this parameter, set **Time domain** to `Continuous-time`.

Programmatic Use

Parameter: `IntegratorContinuousStateAttributes`, `FilterContinuousStateAttributes`
Type: character vector
Default: `''`

`State name` — Names for discrete-time filter and integrator states

empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the **State name (e.g., 'position')** parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Dependencies

To enable this parameter, set **Time domain** to `Discrete-time`.

Programmatic Use

Parameter: `IntegratorStateIdentifier`, `FilterStateIdentifier`
Type: string, character vector
Default: `""`

`State name must resolve to Simulink signal object` — Require that state name resolve to a signal object

`off` (default) | `on`

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class** for the corresponding integrator or filter state.

Programmatic Use
Block Parameter: `IntegratorStateMustResolveToSignalObject`, `FilterStateMustResolveToSignalObject`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

`Code generation storage class` — Storage class for code generation

`Auto` (default) | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer`

Select state storage class for code generation. If you do not need to interface to external code, select `Auto`.

For more information, see "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder) and "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder).

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to `Discrete-time`.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than `None`.

Programmatic Use
Block Parameter: `IntegratorRTWStateStorageClass`, `FilterRTWStateStorageClass`
Type: string, character vector
Values: `"Auto"`, `"ExportedGlobal"`, `"ImportedExtern"` | `"ImportedExternPointer"`
Default: `"Auto"`

`Code generation storage type qualifier` — Storage type qualifier

empty string (default) | character vector | `"const"` | `"volatile"` | ...

Specify a storage type qualifier such as `const` or `volatile`.

Note This parameter 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.

Dependencies

To enable this parameter, set **Code generation storage class** to any value other than `Auto`.

Programmatic Use

Block Parameter:
`IntegratorRTWStateStorageTypeQualifier`,`FilterRTWStateStorageTypeQualifier`
Type: string, character vector
Values:`""`,`"const"`,`"volatile"`
Default: `""`

Block Characteristics

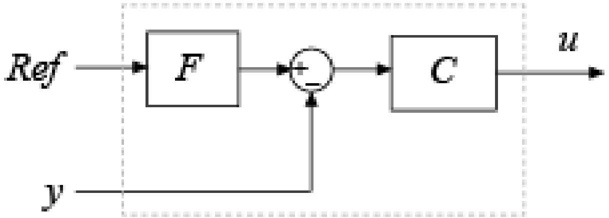
Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Definitions

Decomposition of 2-DOF PID Controllers

A 2-DOF PID controller can be interpreted as a PID controller with a prefilter, or a PID controller with a feedforward element.

In parallel form, a two-degree-of-freedom PID controller can be equivalently modeled by the following block diagram, where *C* is a single degree-of-freedom PID controller and *F* is a prefilter on the reference signal.



Ref is the reference signal, *y* is the feedback from the measured system output, and *u* is the controller output. For a continuous-time 2-DOF PID controller in parallel form, the transfer functions for *F* and *C* 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 weights.

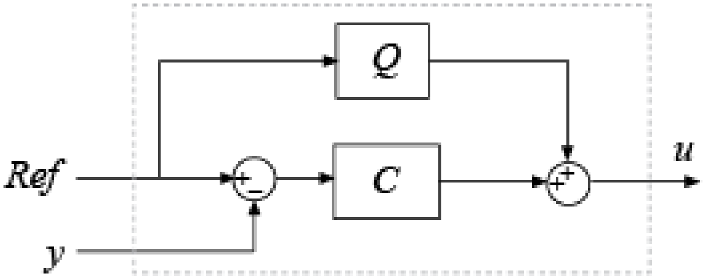
For a 2-DOF PID controller in ideal form, the transfer functions are

$$F_{id}(s) = \frac{(b+cDN)s^2 + (bN+I)s + IN}{(1+DN)s^2 + (N+I)s + IN},$$

$$C_{id}(s) = P\frac{(1+DN)s^2 + (N+I)s + IN}{s(s+N)}.$$

A similar decomposition applies for a discrete-time 2-DOF controller.

Alternatively, the parallel two-degree-of-freedom PID controller can be modeled by the following block diagram.



In this realization, *Q* acts as feed-forward conditioning on the reference signal. For a continuous-time 2-DOF PID controller in parallel form, the transfer function for *Q* is

$$Q_{par}(s) = \frac{\left((b-1)P + (c-1)DN\right)s + (b-1)PN}{s+N}.$$

For a 2-DOF PID controller in ideal form, the transfer function is

$$Q_{id}(s) = P\frac{\left((b-1) + (c-1)DN\right)s + (b-1)N}{s+N}.$$

The transfer functions for *C* are the same as in the filter decomposition.

A similar decomposition applies for a discrete-time 2-DOF controller.

References

[1] Visioli, A., "Modified Anti-Windup Scheme for PID Controllers," *IEE Proceedings - Control Theory and Applications*, Vol. 150, Number 1, January 2003

Extended Capabilities

C/C++ Code Generation

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

For continuous-time PID controllers (**Time domain** set to `Continuous-time`):

- Consider using “Model Discretizer” to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, from your model, select **Analysis** > **Control Design** > **Analysis** > **Model Discretizer**.
- Not recommended for production code.

For discrete-time PID controllers (**Time domain** set to `Discrete-time`):

- Depends on absolute time when placed inside a triggered subsystem hierarchy.
- Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (**Time domain** set to `Discrete-time`).

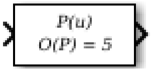
See Also

Derivative | Discrete PID Controller (2DOF) | Gain | Integrator | PID Controller

Introduced in R2009b

Polynomial

Perform evaluation of polynomial coefficients on input values
Library: Simulink / Math Operations



Description

The Polynomial block evaluates `P(u)` at each time step for the input *u*. You define a set of polynomial coefficients in the form that the MATLAB `polyval` command accepts.

Ports

Input

`Port_1` — Input signal
real scalar or vector

Value at which to evaluate the polynomial *P(u)*.

Data Types: `single` | `double`

Output

`Port_1` — Evaluated polynomial value
real scalar or vector

Value of the polynomial *P(u)* evaluated at the input signal.

Data Types: `single` | `double`

Parameters

Polynomial coefficients — Coefficients of polynomial to be evaluated
[+2.081618890e-019, -1.441693666e-014, +4.719686976e-010, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001] (default) | real array

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.

Programmatic Use
Block Parameter: coefs
Type: real array
Default: [+2.081618890e-019, -1.441693666e-014, +4.719686976e-010, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001]

Block Characteristics

Data Types	double | single
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

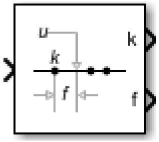
Topics

`polyval`

Introduced before R2006a

Prelookup

Compute index and fraction for Interpolation Using Prelookup block
Library: Simulink / 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 must specify a set of breakpoint values. You choose whether to specify the breakpoint values directly on the dialog box or by feeding the values to a `bp` input port by setting the **Source** parameter to `Dialog` or `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 ≤ 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 **Extrapolation method** block parameter for more information.

Ports

Input

`Port_1` — Input signal, `u`

scalar | vector | matrix

The Prelookup block accepts real-valued signals of any numeric data type that Simulink supports, except Boolean. The Prelookup block supports fixed-point data types for signals and breakpoint data.

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

Output

k — Index of the interval containing the input, `u`

scalar | vector | matrix

The zero-based index, *k*, is a real-valued integer that specifies the interval containing the input, `u`.

Dependencies

To enable this port, set the **Output selection** to `Index and fraction` or `Index only`.

Data Types: `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

f — Fraction representing the normalized position of the input, `u`, within the interval, *k*

scalar | vector | matrix

Fraction, *f*, represents the normalized position of the input, `u`, within the interval *k*.

Dependencies

To enable this port, set the **Output selection** to `Index and fraction`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_2` — Bus containing index, *k*, and fraction, *f*
bus

Outputting the index, *k*, and fraction *f*, as a bus object can help simplify the model.

Dependencies

To enable this port, set the **Output selection** to `Index and fraction as bus`.

Data Types: `bus`

Parameters

Main

Breakpoints data

`Specification` — Choose how to enter breakpoint data
`Explicit values` (default) | `Even spacing` | `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.

Programmatic Use
Block Parameter: `BreakpointsSpecification`
Type: character vector
Values: `'Explicit values' | 'Even spacing' | 'Breakpoint object'`
Default: `'Explicit values'`

`Source` — Specify source of breakpoint data
`Dialog` (default) | `Input port`

If you set **Source** to:

- `Dialog`, specify breakpoint data 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.

Dependencies

To enable this parameter, set **Specification** to `Explicit values`.

Programmatic Use
Block Parameter: `BreakpointsDataSource`
Type: character vector
Values: `'Dialog' | 'Input port'`
Default: `'Dialog'`

Value — Breakpoint data values

`[10:10:110]` (default)

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 additional breakpoint attributes on the **Data Types** pane.

To open the Lookup Table Editor, click **Edit** (see “Edit Lookup Tables”).

Note When you set **Specification** to `Explicit values` and **Source** to `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 (including data type) from the `bp` input port.

Dependencies

To enable this parameter, set **Specification** to `Explicit values` and **Source** to `Dialog`.

Programmatic Use
Block Parameter: `BreakpointsData`

Type: character vector
Values: `'[10:10:110]'`
Default: `'[10:10:110]'`

`First point` — First point in evenly spaced breakpoint data

`10` (default) | `real-valued scalar`

Dependencies

To enable this parameter, set **Specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsFirstPoint`
Type: character vector
Values: `'10'`
Default: `'10'`

`Spacing` — Spacing between evenly spaced breakpoints

`10` (default) | `real-valued, positive scalar`

Dependencies

To enable this parameter, set **Specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsSpacing`
Type: character vector
Values: `'10'`
Default: `'10'`

`Number of points` — Number of evenly spaced points

`11` (default) | `real-valued, positive scalar`

Dependencies

To enable this parameter, set **Specification** to `Even spacing`.

Programmatic Use

Block Parameter: `BreakpointsNumPoints`
Type: character vector
Values: `'11'`
Default: `'11'`

`Name` — Name of an existing `Simulink.Breakpoint` object

`no default` | `Simulink.Breakpoint`

Dependencies

To enable this parameter, set **Specification** to `Breakpoint object`.

Programmatic Use
Block Parameter: `BreakpointObject`
Type: character vector
Values: `Simulink.Breakpoint` object
Default: `''`

Algorithm

`Output selection` — Specify the signals the block outputs

`Index and fraction` (default) | `Index and fraction as bus` | `Index only`

If you want the block to output the index and interval fraction, you can specify whether the block outputs 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.

Programmatic Use
Block Parameter: `OutputSelection`
Values: `'Index and fraction' | 'Index and fraction as bus' | 'Index only'`
Type: character vector
Default: `'Index and fraction'`

`Index search method` — Method for searching breakpoint data

`Evenly spaced points` (default) | `Linear search` | `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`.

`Begin index search using previous index result` — Start search using the index found at the previous time step

`off` (default) | `on`

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.

Programmatic Use
Block Parameter: `IndexSearchMethod`
Values: `'Binary search' | 'Evenly spaced points' | 'Linear search'`
Type: character vector
Default: `'Binary search'`

Extrapolation method — Method for handling out-of-range input values

`Clip` (default) | `Linear`

Options include:

- `Clip`

Block Input	Block Outputs
Less than the first breakpoint	• Index of the first breakpoint (for example, 0) • Interval fraction of 0
Greater than the last breakpoint	• 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	• 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	• 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 the following conditions are true:

- The input u, breakpoint data, and fraction output use floating-point data types.
- The index uses a built-in integer data type.

Programmatic Use
Block Parameter: ExtrapMethod
Type: character vector
Values: 'Clip' | 'Linear'
Default: 'Clip'

Use last breakpoint for input at or above upper limit — Method of handling inputs at or above upper limit

off (default) | on

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 (on)	• Index of the last element in the breakpoint data set • Interval fraction of 0
Cleared (off)	• Index of the next-to-last breakpoint • Interval fraction of 1

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.

Dependencies

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.

Programmatic Use
Block Parameter: `UseLastBreakpoint`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Diagnostic for out-of-range input` — Block action when input is out of range

`None` (default) | `Warning` | `Error`

Options include:

- `None` — Produce no response.
- `Warning` — Display a warning and continue the simulation.
- `Error` — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: `DiagnosticForOutOfRangeInput`
Type: character vector
Values: `'None' | 'Warning' | 'Error'`
Default: `'None'`

Code generation

`Remove protection against out-of-range input in generated code` — Remove code that checks for out-of-range breakpoint inputs

`Off` (default) | `On`

Check Box	Result	When to Use
`On`	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
`Off`	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”.

Programmatic Use
Block Parameter: `RemoveProtectionInput`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Data Types

`Breakpoint` — Breakpoint data type

`Inherit: Same as input` (default) | `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)` | `Enum: <class name>` | `<data type expression>`

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 class, for example, an enumerated data type class
- 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 data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

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`
- Enumerated data:
  - Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.
  - If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
  - For enumerated data, **Extrapolation method** must be `Clip`.
  - Because the fraction is `1` or `0`, select **Output selection** > **Index only**.

    If you are using the index only output selection setting with the Interpolation Using Prelookup block, consider using the **Number of sub-table selection dimensions** parameter.

These are limitations for using enumerated data with this block:

- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.
- When breakpoints data source is set to `Inport port`, the enumeration data type must have `0` as the default value. For example, for this enumeration class, the default value of `GEAR1` must be `0`.

```
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(32),
        NEUTRAL(0)
    end
end
```

Dependencies

To enable this parameter, set the breakpoints data **Source** to `Dialog`.

Note When you set **Source** to `Input port`, the block inherits all breakpoint attributes (data type, minimum, and maximum) from the `bp` input port.

Programmatic Use

Block Parameter: `BreakpointDataTypeStr`
Type: character vector
Values: `'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)'|'<data type expression>'`
Default: `'Inherit: Same as input'`

Index — Index data type

`uint32` (default) | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `<data type expression>`

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

Programmatic Use

Block Parameter: `IndexDataTypeStr`
Type: character vector
Values:
`'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | '<data type expression>'`
Default: `'uint32'`

`Fraction` — Fraction data type

`Inherit: Inherit via internal rule` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

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

Dependencies

This parameter displays only when you set **Output selection** on the **Main** tab to `Index and fraction`.

Programmatic Use
Block Parameter: `FractionDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Output` — Output data type

`Inherit: auto` (default) | `bus: <object name>`

To output a virtual bus, use the `Inherit: auto` setting. The resulting virtual bus contains two elements, the index and the fraction signals.

To output and specify a nonvirtual bus, use the `Bus: <object name>` template. 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 Use the Fixed-Point Tool data type override option to override 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, use the **Fix** button to redefine the original bus object and 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:

```
myBus.Elements(1).DataType = 'fixdt(''int32'',''DataTypeOverride'',''Off'')'
```

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Dependencies

This parameter displays only when you set **Output selection** on the **Main** tab to `Index and fraction as bus`.

Programmatic Use

Block Parameter: `OutputBusDataTypeStr`
Type: character vector
Values: `'Inherit: auto'` | `'Bus: <object name>'` | `'<data type expression>'`
Default: `'Inherit: auto'`

`Breakpoint Minimum` — Minimum value breakpoint data can have

`[]` (default) | scalar

Specify the minimum value that the breakpoint data can have. The default value is `[]` (unspecified).

Dependencies

To enable this parameter, set the breakpoints data **Source** to `Dialog` on the **Main** tab.

Programmatic Use

Block Parameter: `BreakpointMin`
Type: character vector
Value: scalar
Default: `'[]'`

`Breakpoint Maximum` — Maximum value breakpoint data can have

`[]` (default) | scalar

Specify the maximum value that the breakpoint data can have. The default value is `[]` (unspecified).

Dependencies

To enable this parameter, set the breakpoints data **Source** to `Dialog` on the **Main** tab.

Programmatic Use

Block Parameter: `BreakpointMax`
Type: character vector
Value: scalar
Default: `'[]'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see "Rounding" (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Block Characteristics

Data Types	double | single | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Definitions

Enumerated Values in Prelookup

Simulate a Prelookup block with enumerated values.

Suppose that you have a Prelookup block with an enumerated class like this defined:

```
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
        NEUTRAL(0)
    end
end
```

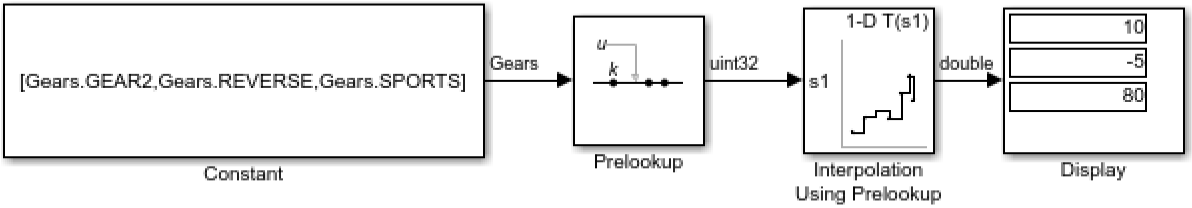
Prelookup block has these settings:

- **Breakpoints data** value is `enumeration('Gears')`.
- **Output selection** is `Index only`.
- For an unordered search, set **Index search method** to `Linear search` and clear the **Begin index search using previous index result** check box.
- **Extrapolation method** is `Clip`.

Interpolation using Prelookup block has these settings:

- **Number of dimensions** to `1`.
- **Table data** value is `[5 10 20 40 80 -5 0]`.
- **Interpolation method** is `Flat`.
- **Number of sub-table selection dimensions** is `1`.

Simulation produces a vector `[10 -5 80]`, which correspond to `GEAR2`, `REVERSE`, and `SPORTS`.



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™.

For more information about HDL Code Generation, see Prelookup.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Simulink PLC Coder has limited support for lookup table blocks. The coder does not support:

- Number of dimensions greater than 2
- Cubic spline interpolation method
- Begin index search using a previous index mode
- Cubic spline extrapolation method

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Interpolation Using Prelookup | `Simulink.Breakpoint` | n-D Lookup Table

Topics

“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”

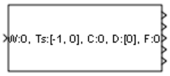
“Guidelines for Choosing a Lookup Table”

Introduced in R2006b

Probe

Output signal attributes, including width, dimensionality, sample time, and complex signal flag

Library: Simulink / Signal Attributes



Description

The Probe block outputs selected information about the signal on its input. The block can output the following attributes of the input signal: 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, with an independent data type control. During simulation, the block icon displays the probed data.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix | N-D array

Input signal to probe, specified as a scalar, vector, matrix, or N-D array. The block accepts real or complex-valued signals of any built-in data type.

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”.

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

Output

`Port_1 (W)` — Signal width

scalar

Width, or number of elements, in the input signal, specified as a scalar. The width is also displayed on the block icon with the notation `W:`.

Dependencies

To enable this port, select **Probe width**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_2 (Ts)` — Sample time

vector

Sample time of the input signal, as a two-element vector that specifies the period and offset of the sample time, respectively. The sample time is also displayed on the block icon with the notation `Ts:`. See “Specify Sample Time” for more information.

Dependencies

To enable this port, select **Probe sample time**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_3 (C)` — Signal complexity

scalar

Indication of input signal complexity:

- When the input signal is complex, the block outputs `1`.
- When the input signal is real-valued, the block outputs `0`.

The indication of signal complexity is also displayed on the block icon with the notation `C:`.

Dependencies

To enable this port, select **Detect complex signal**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_4 (D)` — Signal dimensions

scalar | vector

Dimensions of the input signal, output as a scalar or vector. The signal dimensions are also displayed on the block icon with the notation `D:`.

Dependencies

To enable this port, select **Probe signal dimensions**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Main

`Probe width` — Output width of the input signal

`on` (default) | `off`

Select to output the width, or number of elements, of the probed signal.

Programmatic Use

Block Parameter: `ProbeWidth`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Probe sample time` — Output sample time of input signal

`on` (default) | `off`

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.

Programmatic Use

Block Parameter: `ProbeSampleTime`
Type: character vector

Values: 'off' | 'on'
Default: 'on'

Detect complex signal — Indicate the complexity of input signal

on (default) | off

Select to output 1 if the probed signal is complex; otherwise, 0.

Programmatic Use
Block Parameter: ProbeComplexSignal
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Probe signal dimensions — Output dimensions of input signal

on (default) | off

Select to output the dimensions of the probed signal.

Programmatic Use
Block Parameter: ProbeSignalDimensions
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Signal Attributes

Data type for width — Data type of signal width output

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | Same as input

Select the output data type for the signal width.

Programmatic Use
Block Parameter: ProbeWidthDataType
Type: character vector
Values: 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'
Default: 'double'

Data type for sample time — Data type of sample time output

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | Same as input

Select the output data type for the sample time information.

Programmatic Use
Block Parameter: `ProbeSampleTimeDataType`
Type: character vector
Values: `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'Same as input'`
Default: `'double'`

`Data type for signal complexity` — Data type of complexity output

`double` (default) | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `Same as input`

Select the output data type for the complexity information.

Programmatic Use
Block Parameter: `ProbeComplexityDataType`
Type: character vector
Values: `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'Same as input'`
Default: `'double'`

`Data type for signal dimensions` — Data type for signal dimension output

`double` (default) | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `Same as input`

Select the output data type for the signal dimension output.

Programmatic Use
Block Parameter: `ProbeDimensionsDataType`
Type: character vector
Values: `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'Same as input'`
Default: `'double'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`

Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
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™.

For information on HDL code generation support, see Probe.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Clock | Digital Clock | Weighted Sample Time Math

Topics

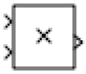
"Specify Sample Time"

Introduced before R2006a

Product

Multiply and divide scalars and nonscalars or multiply and invert matrices

Library: Simulink / Commonly Used Blocks
Simulink / Math Operations

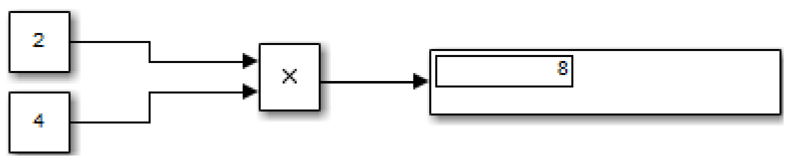


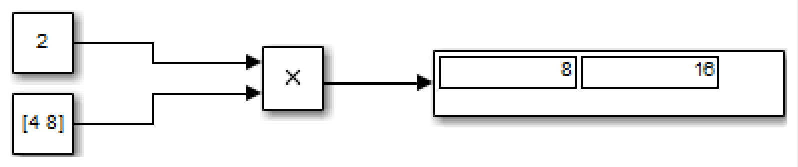
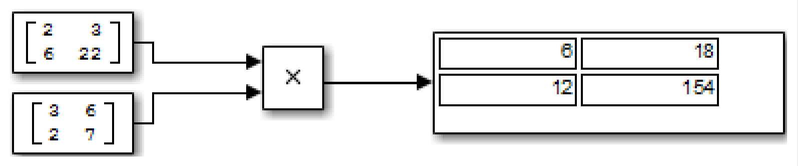
Description

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

This table shows the output of the Product block for example inputs using default block parameter values.

Inputs and Behavior	Example
Scalar X Scalar Output the product of the two inputs.	

Inputs and Behavior	Example
Scalar X Nonscalar 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.	
Nonscalar X Nonscalar Output a nonscalar having the same dimensions as the inputs. Each element of the output is the product of corresponding elements of the inputs.	

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 (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.

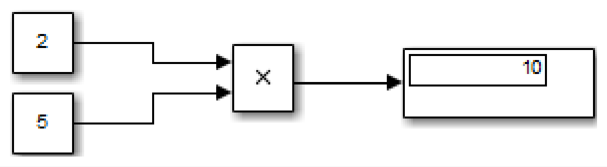
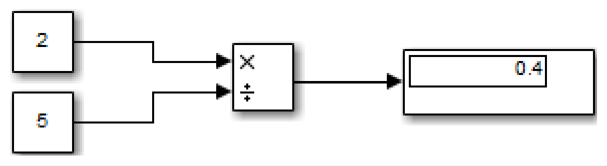
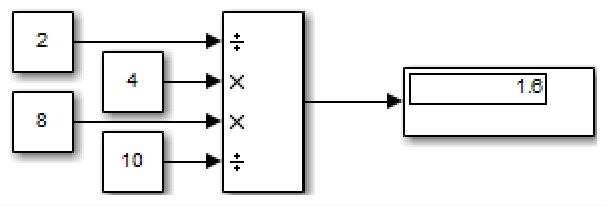
The Product block has two modes: *Element-wise mode*, which processes nonscalar inputs element by element, and *Matrix mode*, which processes nonscalar inputs as matrices.

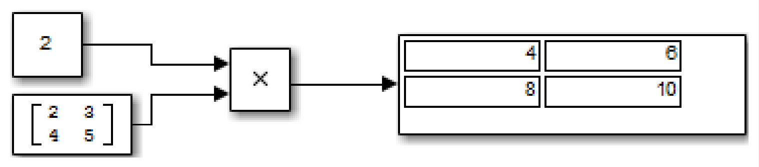
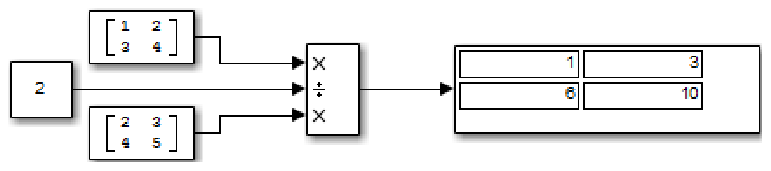
Element-Wise Mode

When you set **Multiplication** to `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 is in element-wise mode and 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	
Number of inputs: */	
Number of inputs: /**/	

Parameter Values	Examples
Number of inputs:**	
Number of inputs: */*	 1 2 3 4 2 2 3 4 5 × ÷ × 1 3 6 10

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 could include extreme values, you must 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.

Ports

Input

Port_1 — First input to multiply or divide

scalar | vector | matrix | N-D array

First input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_N — Nth input to multiply or divide

scalar | vector | matrix | N-D array

Nth input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

X — Input signal to multiply

scalar | vector | matrix | N-D array

Input signal to be multiplied with other inputs.

Dependencies

To enable one or more **X** ports, specify one or more * characters for the **Number of inputs** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

÷ — Input signal to divide or invert

scalar | vector | matrix | N-D array

Input signal for division or inversion operations.

Dependencies

To enable one or more ÷ ports, specify one or more `/` characters for the **Number of inputs** parameter.

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

Output

`Port_1` — Output computed by multiplying, dividing, or inverting inputs

scalar | vector | matrix | N-D array

Output computed by multiplying, dividing, or inverting inputs.

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

Parameters

Main

`Number of inputs` — Control number of inputs and type of operation

2 (default) | scalar | * or / for each input port

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

When you specify:

- **1 or * or /**

  The block has one input port. In element-wise mode, the block 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 1-1448 and “Matrix Mode” on page 1-1449 for more information.

- **Integer value > 1**

  The block 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 1-1448 and “Matrix Mode” on page 1-1449 for more information.
- **Unquoted string of two or more * and / characters**

  The block 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 1-1448 and “Matrix Mode” on page 1-1449 for more information.

Programmatic Use

Block Parameter: `Inputs`
Type: character vector
Values: `'2' | '**' | '*/' | '*/*' | ...`
Default: `'2'`

`Multiplication` — Element-wise (.*) or Matrix (*) multiplication

`Element-wise(.*)` (default) | `Matrix(*)`

Specify whether the block performs `Element-wise(.*)` or `Matrix(*)` multiplication.

Programmatic Use

Block Parameter: `Multiplication`
Type: character vector
Values: `'Element-wise(.*)' | 'Matrix(*)'`
Default: `'Element-wise(.*)'`

`Multiply over` — All dimensions or specified dimension

`All dimensions` (default) | `Specified dimension`

Specify the dimension to multiply over as `All dimensions`, or `Specified dimension`. When you select `Specified dimension`, you can specify the **Dimension** as `1` or `2`.

Dependencies

To enable this parameter, set **Number of inputs** to `*` and **Multiplication** to `Element-wise (.*)`.

Programmatic Use

Block Parameter: `CollapseMode`
Type: character vector
Values: `'All dimensions' | 'Specified dimension'`
Default: `'All dimensions'`

`Dimension` — Dimension to multiply over

`1` (default) | `2` | `...` | `N`

Specify the dimension to multiply over as an integer less than or equal to the number of dimensions of the input signal.

Dependencies

To enable this parameter, set:

- **Number of inputs** to `*`
- **Multiplication** to `Element-wise (.*)`
- **Multiply over** to `Specified dimension`

Programmatic Use

Block Parameter: `CollapseDim`
Type: character vector
Values: `'1' | '2' | ...`
Default: `'1'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

`Require all inputs to have the same data type` — Require that all inputs have the same data type

`off` (default) | `on`

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar

Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `Inherit: Same as first input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- `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)`, in other words, 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 first input` — Use data type of first input signal.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Same as first input'` | `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type
`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations. You can select:

`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.

For more information, see “Rounding” (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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	Rationale	Impact on Overflows	Example
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Check for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`

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™.

For more information on HDL code generation, see Product.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

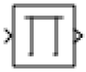
See Also

Divide | Dot Product | Product of Elements

Introduced before R2006a

Product of Elements

Copy or invert one scalar input, or collapse one nonscalar input
Library: Simulink / Math Operations



Description

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 using one of these options:
  - Multiply together the elements of each row or column
  - Take 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 nondefault 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.

Ports

Input

Port_1 — First input to multiply or divide

scalar | vector | matrix | N-D array

First input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_N — Nth input to multiply or divide

scalar | vector | matrix | N-D array

Nth input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

X — Input signal to multiply

scalar | vector | matrix | N-D array

Input signal to be multiplied with other inputs.

Dependencies

To enable one or more **X** ports, specify one or more * characters for the **Number of inputs** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

÷ — Input signal to divide or invert

scalar | vector | matrix | N-D array

Input signal for division or inversion operations.

Dependencies

To enable one or more ÷ ports, specify one or more / characters for the **Number of inputs** parameter.

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

Output

`Port_1` — Output computed by multiplying, dividing, or inverting inputs
scalar | vector | matrix | N-D array

Output computed by multiplying, dividing, or inverting inputs.

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

Parameters

Main

`Number of inputs` — Control number of inputs and type of operation
`*` (default) | positive integer scalar | `*` or `/` for each input port

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

When you specify:

- **1 or * or /**

  The block has one input port. In element-wise mode, the block 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 1-1448 and “Matrix Mode” on page 1-1449 for more information.
- **Integer value > 1**

  The block 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 1-1448 and "Matrix Mode" on page 1-1449 for more information.

- **Unquoted string of two or more * and / characters**

  The block 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 1-1448 and "Matrix Mode" on page 1-1449 for more information.

Programmatic Use

Block Parameter: `Inputs`
Type: character vector
Values: `'2' | '*' | '**' | '*/' | '*/*' | ...`
Default: `'*'`

`Multiplication` — Element-wise (.*) or Matrix (*) multiplication

`Element-wise(.*)` (default) | `Matrix(*)`

Specify whether the block performs `Element-wise(.*)` or `Matrix(*)` multiplication.

Programmatic Use

Block Parameter: `Multiplication`
Type: character vector
Values: `'Element-wise(.*)' | 'Matrix(*)'`
Default: `'Element-wise(.*)'`

`Multiply over` — All dimensions or specified dimension

`All dimensions` (default) | `Specified dimension`

Specify the dimension to multiply over as `All dimensions`, or `Specified dimension`.

When you select `All dimensions` and select configuration parameter **Use algorithms optimized for row-major array layout**, Simulink enables row-major algorithms for simulation. To generate row-major code, set configuration parameter **Array layout** (Simulink Coder) to `Row-major` in addition to selecting **Use algorithms optimized for row-major array layout**. The column-major and row-major algorithms differ only in the multiplication order. In some cases, due to different operation order on the same data set, you might experience minor numeric differences in the outputs of column-major and row-major algorithms.

When you select `Specified dimension`, you can specify the **Dimension** as `1` or `2`.

Dependencies

To enable this parameter, set **Number of inputs** to `*` and **Multiplication** to `Element-wise (.*)`.

Programmatic Use

Block Parameter: `CollapseMode`
Type: character vector
Values: `'All dimensions' | 'Specified dimension'`
Default: `'All dimensions'`

`Dimension` — Dimension to multiply over

`1` (default) | `2` | `...` | `N`

Specify the dimension to multiply over as an integer less than or equal to the number of dimensions of the input signal.

Dependencies

To enable this parameter, set:

- **Number of inputs** to `*`
- **Multiplication** to `Element-wise (.*)`
- **Multiply over** to `Specified dimension`

Programmatic Use

Block Parameter: `CollapseDim`
Type: character vector
Values: `'1' | '2' | ...`
Default: `'1'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

`Require all inputs to have the same data type` — Require that all inputs have the same data type

`off` (default) | `on`

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[]` (default) | scalar

Upper value of the output range that Simulink checks.

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" (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `Inherit: Same as first input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- `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)`, in other words, 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 first input` — Use data type of first input signal.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Same as first input'` | `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` |

'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see "Use Lock Output Data Type Setting" (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations. You can select:

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.

For more information, see “Rounding” (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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	Rationale	Impact on Overflows	Example
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`

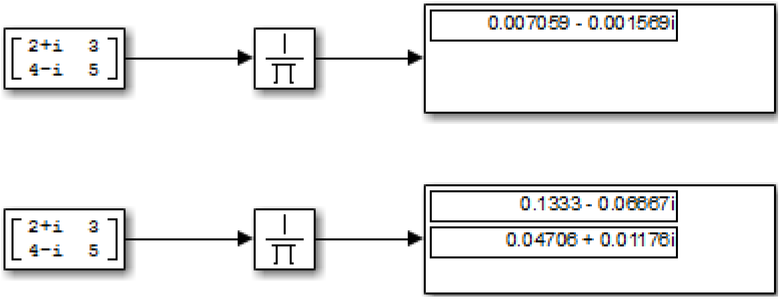
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Algorithms

The Product of Elements block uses these 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 `u1, u2, u3, ..., uN`	Multiplication	`y = u1*u2*u3*...*uN`
	Division	`y = ((((1/u1)/u2)/u3).../uN)`
Complex scalar, `u`	Multiplication	`y = u`
	Division	`y = 1/u`
Complex vector or matrix with elements `u1, u2, u3, ..., uN`	Multiplication	`y = u1*u2*u3*...*uN`
	Division	`y = ((((1/u1)/u2)/u3).../uN)`

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. Consider this 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)`

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™.

For more information about HDL code generation, see Product of Elements.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

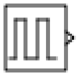
See Also

Divide | Dot Product | Product

Introduced before R2006a

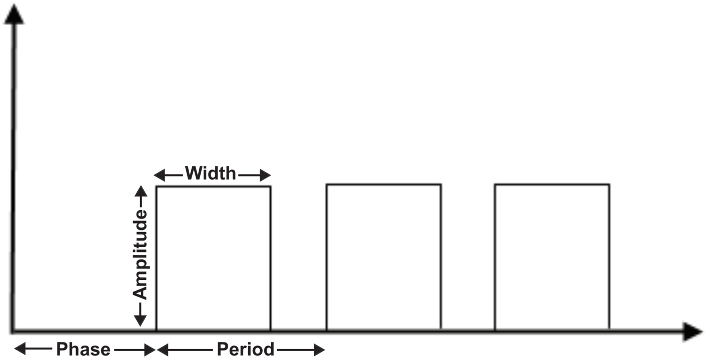
Pulse Generator

Generate square wave pulses at regular intervals
Library: Simulink / 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 block 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 (for example, `0.25`). To ensure this setting, select `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.

This block supports reset semantics in sample-based mode. For example, if a Pulse Generator block is in a resettable subsystem that hits a reset trigger, the block output resets to its initial condition.

Ports

Output

`Port_1` — Output signal
scalar | vector | matrix

Generated square wave pulse signal specified by the parameters.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Pulse type` — Type of pulse
`Time based` (default) | `Sample based`

Specify the type of square 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.

Programmatic Use
Block Parameter: `PulseType`
Type: character vector
Values: `'Time based'` | `'Sample based'`

Default: `'Time based'`

`Time (t)` — Source of time variable

`Use simulation time` (default) | `Use external signal`

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.

Programmatic Use

Block Parameter: `TimeSource`
Type: character vector
Values: `'Use simulation time'` | `'Use external signal'`
Default: `'Use simulation time'`

`Amplitude` — Signal amplitude

`1` (default) | `scalar`

Specify the amplitude of the signal.

Programmatic Use

Block Parameter: `Amplitude`
Type: character vector
Value: scalar
Default: `'1'`

`Period (secs)` — Pulse period

`10` (default) | `scalar`

Pulse period specified in seconds if the pulse type is time-based. If the pulse type is sample-based, then the period is specified as the number of sample times.

Programmatic Use

Block Parameter: `Period`
Type: character vector
Value: scalar

Default: `'10'`

`Pulse width` — Duty cycle

`5` (default) | `scalar in the range [0,100]`

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.

Programmatic Use

Block Parameter: `PulseWidth`
Type: character vector
Value: scalar
Default: `'5'`

`Phase delay (secs)` — Delay before pulse

`0` (default) | `scalar`

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.

Programmatic Use

Block Parameter: `PhaseDelay`
Type: character vector
Value: scalar
Default: `'0'`

`Sample time` — Length of sample time

`0` (default) | `scalar`

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”.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Value: scalar
Default: `'0'`

`Interpret vector parameters as 1-D` — Treat vectors as 1-D

`on` (default) | `off`

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.

- When you select this check box, the block 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`.
- When you clear this check box, the block does not output a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector.

Programmatic Use
Block Parameter: `VectorParams1D`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Cannot be used inside a triggered subsystem hierarchy.

These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Signal Generator | Waveform Generator

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.

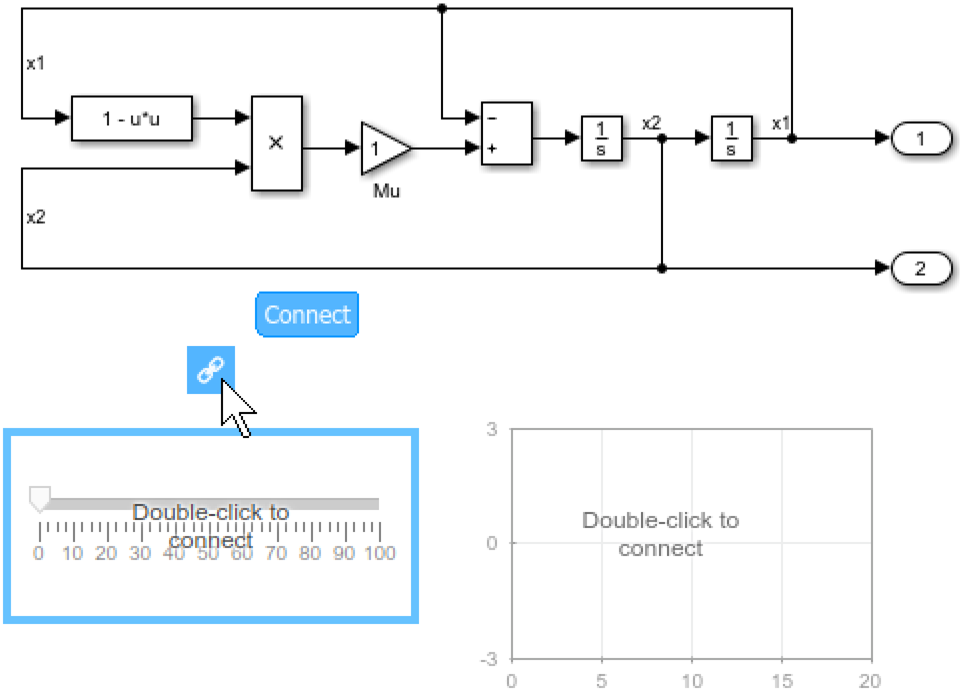
Double-clicking the Push Button block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

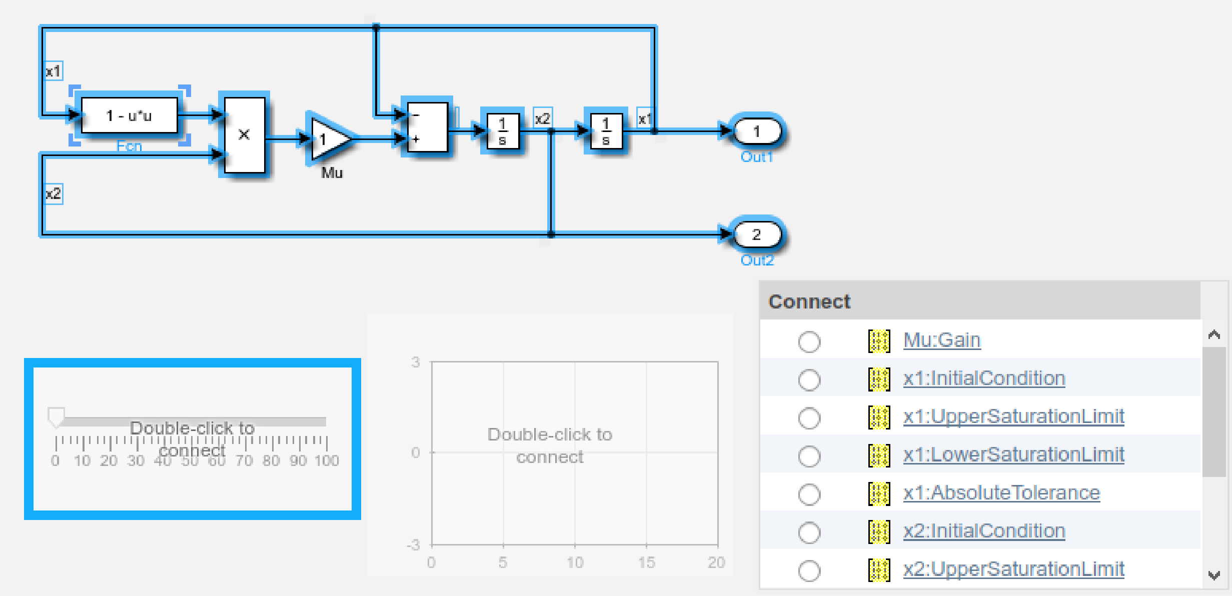
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○	Mu:Gain
○	x1:InitialCondition
○	x1:UpperSaturationLimit
○	x1:LowerSaturationLimit
○	x1:AbsoluteTolerance
○	x2:InitialCondition
○	x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

See Also

Rocker Switch | Slider Switch | Toggle Switch

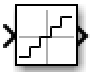
Topics

“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a

Quantizer

Discrete input at given interval
Library: Simulink / Discontinuities



Description

The Quantizer block passes the input signal through a stair-step function. Many neighboring points on the input axis are mapped to one point on the output axis. The effect quantizes a smooth signal into a stair-step output. The block uses a round-to-nearest method to produce an output that is symmetric about zero.

```
y = q * round(u/q)
```

where `y` is the output, `u` the input, and `q` the **Quantization interval** parameter.

Ports

Input

`Input 1` — Input signal to quantize

scalar | vector

The input signal to the quantization algorithm.

Data Types: `single` | `double`

Output

`Output 1` — Quantized output signal

real or complex scalar | real or complex vector

Output signal as quantized discrete values.

```
y = q * round(u/q)
```

where y is the output, u the input, and q the **Quantization interval** parameter.

Data Types: single | double

Parameters

Quantization interval — Interval around which the block quantizes the output

0.5 (default) | scalar | vector

Specify the quantization interval used in the algorithm. Permissible output values for the Quantizer block are n*q, where n is an integer and q the **Quantization interval**.

Programmatic Use
Block Parameter: QuantizationInterval
Type: character vector
Value: Any real or complex value
Default: '0.5'

Treat as gain when linearizing — Specify the gain value

On (default) | boolean

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. Clear the box to have the commands treat the gain as 0.

Programmatic Use
Block Parameter: LinearizeAsGain
Type: character vector
Value: 'off' | 'on'
Default: 'on'

Sample time — Specify sample time as a value other than -1

-1 (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

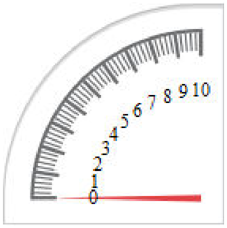
See Also

Rate Limiter | Relay

Introduced before R2006a

Quarter Gauge

Display input value on quadrant scale
Library: Simulink / Dashboard



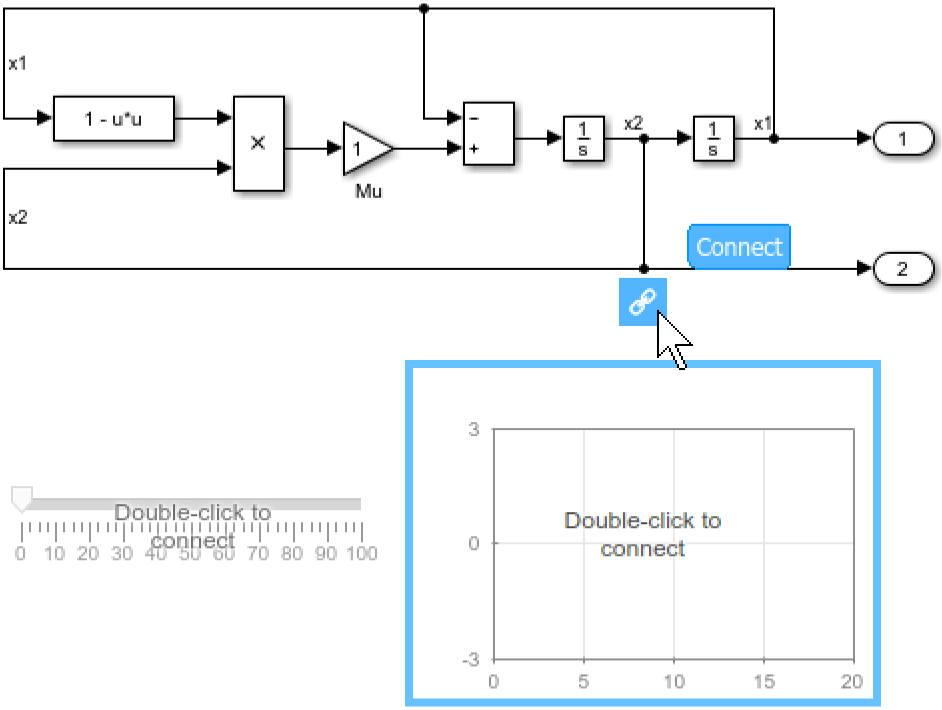
Description

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.

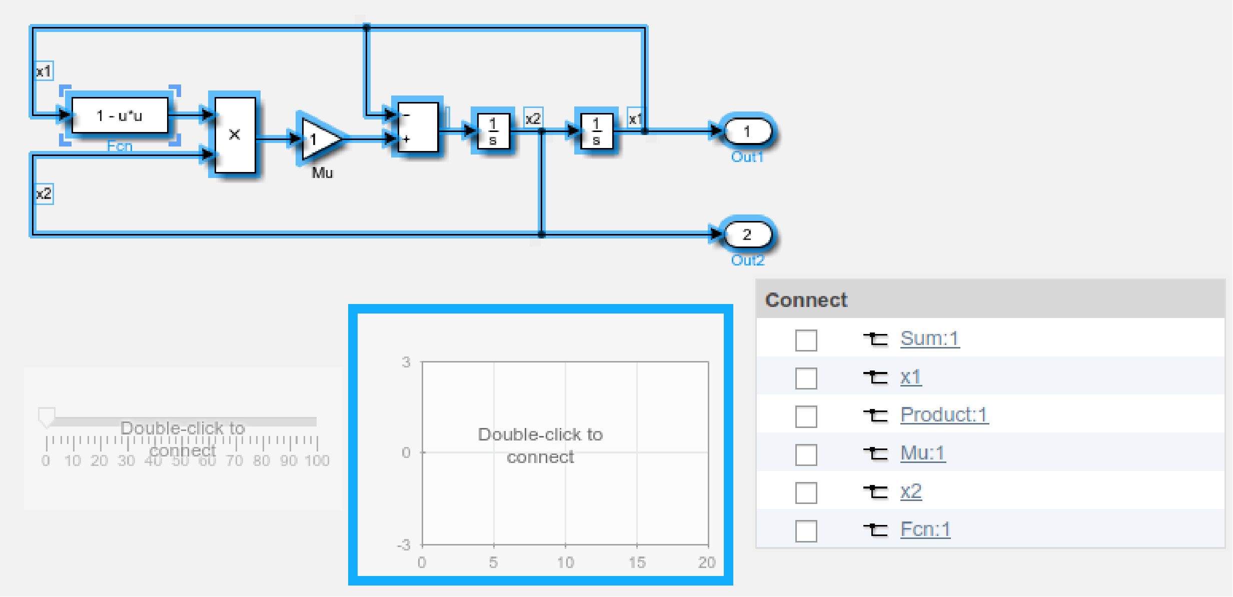
Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect	
□	Sum:1
□	x1
□	Product:1
□	Mu:1
□	x2
□	Fcn:1

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced 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.

Parameters

`Connection` — Select a signal to connect and display

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button 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 arc. 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 arc. 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 arc. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

`Scale Colors` — Color indications on gauge arc

colors for arc ranges

Color specifications for ranges on the arc. 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.

See Also

Custom Gauge | Gauge | Half Gauge | Linear Gauge

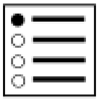
Topics

“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.

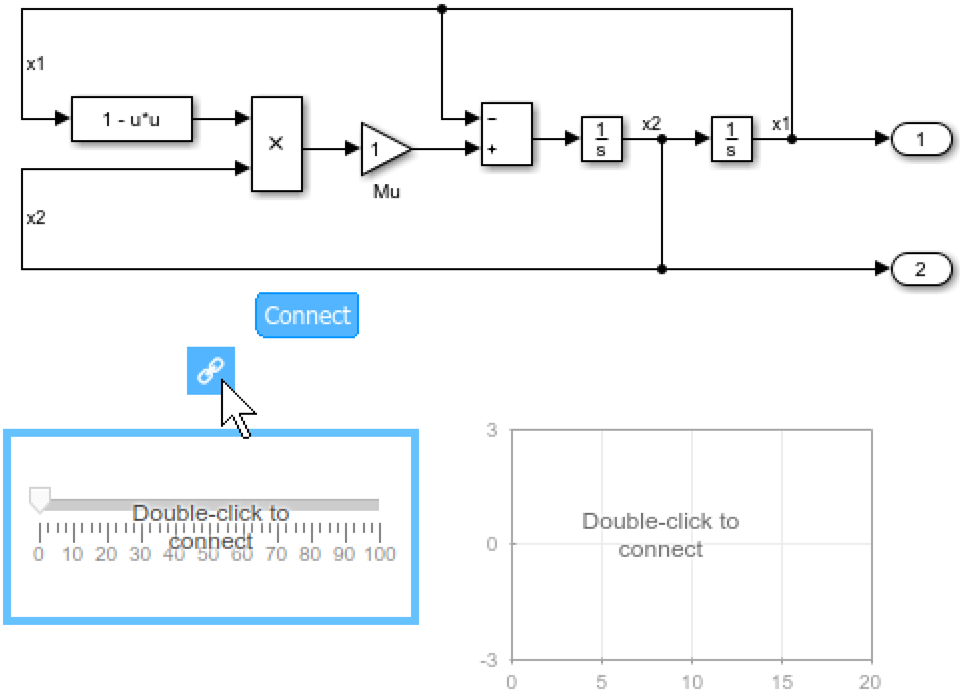
Double-clicking the Radio Button block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

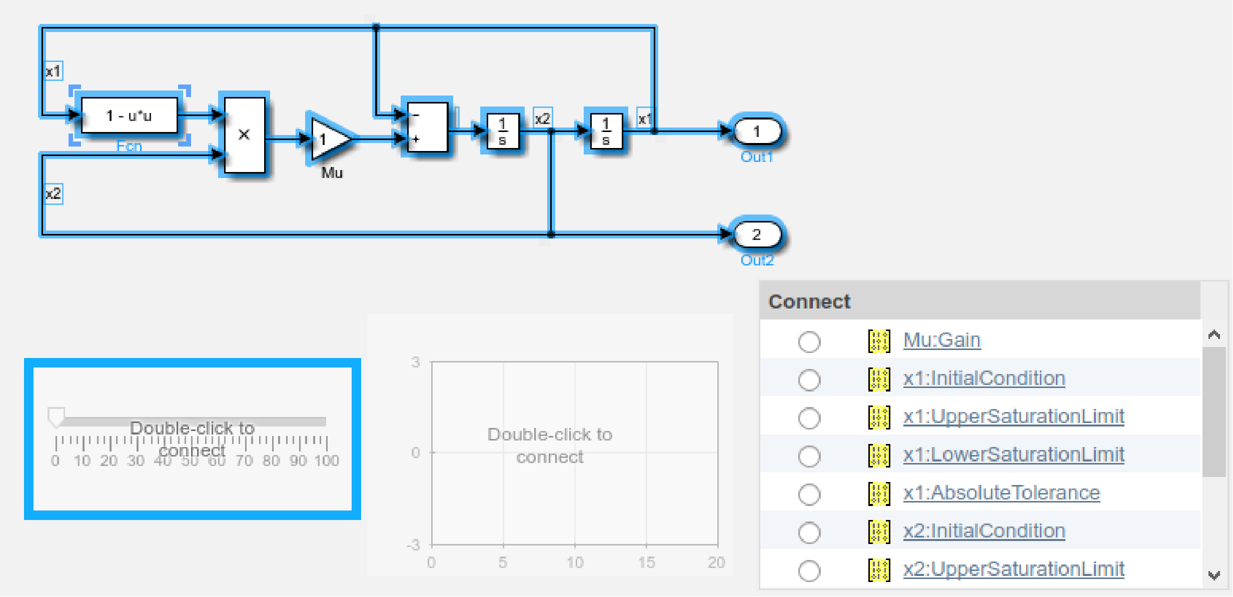
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○ Mu:Gain
○ x1:InitialCondition
○ x1:UpperSaturationLimit
○ x1:LowerSaturationLimit
○ x1:AbsoluteTolerance
○ x2:InitialCondition
○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

See Also

Check Box | Combo Box | Rotary Switch

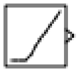
Topics

"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: Simulink / 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.

Ports

Output

`Port_1` — Output signal
scalar | vector | matrix

Generated output ramp signal characterized by the **Slope**, **Start time, and Initial output** parameters.

Data Types: `double`

Parameters

`Slope` — Slope of signal
1 (default) | scalar | vector | matrix

Specify the rate of change of the generated signal.

Programmatic Use

Block Parameter: `slope`
Type: character vector
Values: scalar
Default: `'1'`

`Start time` — Time output begins

0 (default) | scalar

Specify the time at which the block begins generating the signal.

Programmatic Use

Block Parameter: `start`
Type: character vector
Values: scalar
Default: `'0'`

`Initial output` — Initial value of output signal

0 (default) | scalar | vector | matrix

Specify the initial value of the output signal.

Programmatic Use

Block Parameter: `InitialOutput`
Type: character vector
Values: scalar
Default: `'0'`

`Interpret vector parameters as 1-D` — Treat vectors as 1-D

`on` (default) | `off`

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.

- When you select this check box, the block 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`.
- When you clear this check box, the block does not output a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector.

Programmatic Use

Block Parameter: `VectorParams1D`

Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

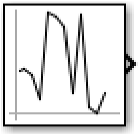
See Also

Pulse Generator | Repeating Sequence

Introduced before R2006a

Random Number

Generate normally distributed random numbers
Library: Simulink / 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.

Ports

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal that is the generated random numbers falling within a normal Gaussian distribution. The output is repeatable for a given seed.

Data Types: `double`

Parameters

`Mean` — Mean of random numbers

`0` (default) | scalar

Specify the mean of the random numbers generated.

Programmatic Use
Block Parameter: `Mean`
Type: character vector
Values: scalar
Default: `'0'`

`Variance` — Variance of random numbers

`1` (default) | scalar

Specify the variance of the random numbers.

Programmatic Use
Block Parameter: `Variance`
Type: character vector
Values: scalar
Default: `'1'`

Seed — Starting seed

`0` (default) | positive integer

Specify the starting seed for the random number generator.

The output of number generated is repeatable for a given seed.

Programmatic Use
Block Parameter: `Seed`
Type: character vector
Values: scalar
Default: `'0'`

`Sample time` — Time between intervals

`0.1` (default) | `integer`

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.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'0.1'`

`Interpret vector parameters as 1-D` — Treat vectors as 1-D

`on` (default) | `off`

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.

- When you select this check box, the block 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`.
- When you clear this check box, the block does not output a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector.

Programmatic Use

Block Parameter: `VectorParams1D`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`

Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

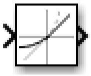
See Also

Band-Limited White Noise | Uniform Random Number

Introduced before R2006a

Rate Limiter

Limit rate of change of signal
Library: Simulink / 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:

$$rate = \frac{u(i) - y(i-1)}{t(i) - t(i-1)}$$

where *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:

$$y(-1) = Ic$$

where *Ic* 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:

$$rate = \frac{u(0) - y(-1)}{sampletime}$$

Limitations

- You cannot use a Rate Limiter block inside a Triggered Subsystem. Use the Rate Limiter Dynamic block instead.

Ports

Input

`Port_1` — Input signal
scalar

The input signal to the rate limiter algorithm.

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

Output

`Port_1` — Output signal
scalar

Output signal from the rate limiter algorithm.

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

Parameters

`Rising slew rate` — Limit of derivative for increasing input

`1` (default) | real number

Specify the limit of the rising rate of the input signal. This parameter is tunable for fixed-point inputs.

Programmatic Use
Block Parameter: `RisingSlewLimit`
Type: character vector
Values: real number
Default: `'1'`

`Falling slew rate` — Limit of derivative for decreasing input

`-1` (default) | real number

Specify the lower limit on the falling rate of the input signal. This parameter is tunable for fixed-point inputs.

Programmatic Use
Block Parameter: `FallingSlewLimit`
Type: character vector
Values: real number
Default: `'-1'`

`Sample time mode` — Sample time mode

`inherited` (default) | `continuous`

Specify the sample time mode, `continuous` or `inherited` from the driving block.

Programmatic Use
Block Parameter: `SampleTimeMode`
Type: character vector
Values: `'inherited'` | `'continuous'` |
Default: `'inherited'`

`Initial condition` — Initial output

`0` (default) | scalar

Set the initial output of the simulation. Simulink does not allow you to set the initial condition of this block to `inf` or `NaN`.

Programmatic Use
Block Parameter: `InitialCondition`
Type: character vector
Values: `scalar`
Default: `'0'`

`Treat as gain when linearizing` — Specify the gain value

`On` (default) | Boolean

Select this check box to cause the commands to treat the gain as 1. The linearization commands in Simulink software treat this block as a gain in state space. Clear the box to have the commands treat the gain as 0.

Programmatic Use
Block Parameter: `LinearizeAsGain`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Cannot be used inside a triggered subsystem hierarchy.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

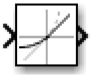
See Also

Rate Limiter Dynamic | Saturation

Introduced before R2006a

Rate Limiter Dynamic

Limit rate of change of signal
Library: Simulink / 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.

Ports

Input

u — Input signal
scalar

Input signal to the rate limiter algorithm.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

lo — Limit of derivative for decreasing input

scalar

Dynamic value providing the limit of the falling rate of the input signal. Make the signal data type of lothe same data type of the input signal u.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

up — Limit of derivative for increasing input

scalar

Dynamic value providing the limit of the rising rate of the input signal. Make the signal data type of upthe same data type of the input signal u.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Y — Output signal

scalar

Output signal from the rate limiter algorithm.

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

Block Characteristics

Data Types	double | single | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No

Zero-Crossing Detection	No

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.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

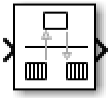
Rate Limiter

Introduced before R2006a

Rate Transition

Handle transfer of data between blocks operating at different rates

Library: Simulink / Signal Attributes



Description

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).

Transition Handling Options

Transition Handling Options	Block Parameter Settings
• Data integrity • Deterministic data transfer • Maximum latency	Select: • **Ensure data integrity during data transfer** • **Ensure deterministic data transfer**
• Data integrity • Nondeterministic data transfer • Minimum latency • Higher memory requirements	Select: • **Ensure data integrity during data transfer** Clear: • **Ensure deterministic data transfer**

Transition Handling Options	Block Parameter Settings
• Potential loss of data integrity • Nondeterministic data transfer • Minimum latency • Lower memory requirements	Clear: • **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 1-1517 and “Effects of Asynchronous Sample Times” on page 1-1519)
- 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)

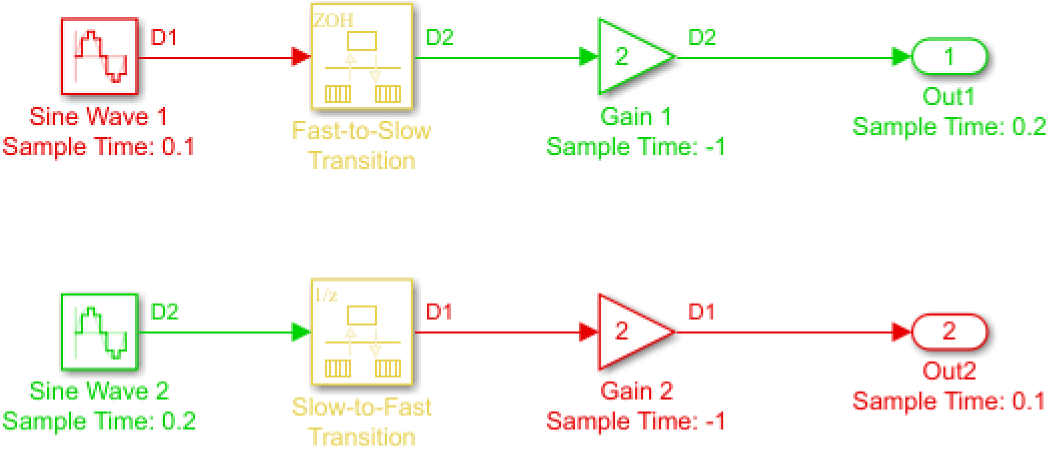
Block Labels

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
`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") to display the relative rates that the block bridges. Consider, for example, the following model:



Sample-time colors and the block behavior label show:

- The Rate Transition block at the top of the diagram acts as a zero-order hold in a fast-to-slow transition.
- The Rate Transition block at the bottom of the diagram acts as a unit delay in a slow-to-fast transition.

For more information, see "Handle Rate Transitions" (Simulink Coder).

Effects of Synchronous Sample Times

The following table summarizes how each label appears when 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**
inTs = outTs	inTsOffset < outTsOffset	None (error)	Buf	Copy or NoOp (see note that

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`	`Copy` or `NoOp` (see note that follows the table)	`Copy` or `NoOp` (see note that follows the table)	follows the table)
	`inTsOffset` > `outTsOffset`	None (error)	`Db_buf`	
`inTs` < `outTs` (Fast to slow)	`inTs` = `outTs / N` `inTsOffset`, `outTsOffset` = 0	`ZOH`	`Buf`	

	`inTs` = `outTs / N`
	`inTsOffset` ≤ `outTsOffset`
		None (error)		

	`inTs` = `outTs / N`
	`inTsOffset` > `outTsOffset`
		None (error)	`Db_buf`	

	`inTs` ≠ `outTs / N`	None (error)		
`inTs` > `outTs` (Slow to fast)	`inTs` = `outTs * N` `inTsOffset`, `outTsOffset` = 0	`1/z`	`Db_buf`	

	`inTs` = `outTs * N`
	`inTsOffset` ≤ `outTsOffset`
		None (error)		

	`inTs` = `outTs * N`
	`inTsOffset` > `outTsOffset`
		None (error)		

	`inTs` ≠ `outTs * N`	None (error)		

KEY

- `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 when 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	Copy	Copy	Copy
inTs ≠ outTs	None (error)	Db_buf	
KEY • inTs, outTs: Sample times of input and output ports, respectively			

Ports

Input

Port_1 — Input signal

scalar | vector | matrix | N-D array

Input signal to transition to a new sample rate, specified as a scalar, vector, matrix, or N-D array. To learn about the block parameters that enable you to trade data integrity and deterministic transfer for faster response or lower memory requirements, see “Transition Handling Options” on page 1-1515.

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

Output

`Port_1` — Output signal
scalar | vector | matrix | N-D array

Output signal is the input signal converted to the sample rate you specify. The default configuration ensures safe and deterministic data transfer. To learn about the block parameters that enable you to trade data integrity and deterministic transfer for faster response or lower memory requirements, see “Transition Handling Options” on page 1-1515.

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

Parameters

`Ensure data integrity during data transfer` — Ensure data integrity
on (default) | `off`

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**), 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 affect 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 (that is, nondeterministically) and data integrity. For more information, see “Rate Transition Block Options” (Simulink Coder).

Programmatic Use
Block Parameter: `Integrity`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Ensure deterministic data transfer (maximum delay)` — Ensure deterministic data transfer
on (default) | `off`

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 when 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).

Programmatic Use
Block Parameter: `Deterministic`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Initial conditions` — Initial conditions

`0` (default) | scalar | vector | matrix | N-D array

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`. The value you specify must be a scalar, or have the same dimensions as the input signal.

Programmatic Use
Block Parameter: `InitialCondition`
Type: character vector
Values: finite scalar
Default: `'0'`

`Output port sample time options` — Mode for specifying output port sample time

`Specify` (default) | `Inherit` | `Multiple of input port sample time`

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 select `Inherit` and all blocks connected to the output port also inherit sample time, the fastest sample time in the model applies.

Programmatic Use
Block Parameter: `OutPortSampleTimeOpt`
Type: character vector
Values: `'Specify' | 'Inherit' | 'Multiple of input port sample time'`
Default: `'Specify'`

`Output port sample time` — Output rate

`-1` (default) | scalar | vector

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.

Dependencies

To enable this parameter, set **Output port sample time options** to `Specify`.

Programmatic Use
Block Parameter: `OutPortSampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

`Sample time multiple(>0)` — Sample time multiple

`1` (default) | positive scalar

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. A value of `2` specifies that the output rate is twice the input rate.

Dependencies

To enable this parameter, set **Output port sample time options** to `Multiple of input port sample time`.

Programmatic Use
Block Parameter: `OutPortSampleTimeMultiple`
Type: character vector

Values: scalar
Default: `'1'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

- Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions
- Cannot use inside a triggered subsystem hierarchy

HDL Code Generation

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

For more information about HDL code generation, see Rate Transition.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Probe | Weighted Sample Time

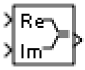
Topics

“Specify Sample Time”
“View Sample Time Information”
“Handle Rate Transitions” (Simulink Coder)

Introduced before R2006a

Real-Imag to Complex

Convert real and/or imaginary inputs to complex signal
Library: Simulink / 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.

Ports

Input

Re — Real part of complex output
scalar | vector | matrix

Real value to be converted to complex-valued output signal, specified as a scalar, vector, or matrix.

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

`Im` — Imaginary part of complex output
scalar | vector | matrix

Imaginary value to be converted to complex-valued output signal, specified as a scalar, vector, or matrix.

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

`Port_1` — Real or imaginary part of complex output
scalar | vector | matrix

Real or imaginary value to convert to complex output signal, specified as a finite scalar, vector, or matrix.

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

Output

`Port_1` — Complex signal
scalar | vector | matrix

Complex signal, formed from real and imaginary values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Input` — Real, imaginary, or both
`Real` (default) | `Imag` | `Real and imag`

Specify the type of input: a real input, an imaginary input, or both.

Programmatic Use
Block Parameter: `Input`
Type: character vector
Values: `'Real and imag' | 'Real' | 'Imag'`
Default: `'Real and imag'`

`Imag part` — Imaginary part of complex output
`0` (default) | finite scalar, vector, or matrix

Specify the imaginary value to use when converting the input to a complex-valued output signal.

Dependencies

To enable this parameter, set **Input** to `Real`.

Programmatic Use
Block Parameter: `ConstantPart`
Type: character vector
Values: imaginary value
Default: `'0'`

`Real part` — Real part of complex output

`0` (default) | finite scalar, vector, or matrix

Specify the constant real value to use when converting the input to a complex-valued output signal.

Dependencies

To enable this parameter, set **Input** to `Imag`.

Programmatic Use
Block Parameter: `ConstantPart`
Type: character vector
Values: finite, real-valued scalar, vector, or matrix
Default: `'0'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar

Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Real-Imag to Complex.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Complex to Magnitude-Angle | Complex to Real-Imag | Magnitude-Angle to Complex

Topics

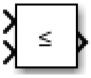
"Complex Signals"

Introduced before R2006a

Relational Operator

Perform specified relational operation on inputs

Library: Simulink / Commonly Used Blocks
Simulink / Logic and Bit Operations



Description

The Relational Operator performs the specified relational operation on the input. The value you choose for the **Relational operator** parameter determines whether the block accepts one or two input signals.

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" 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

For...	The output is...
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
• 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	`double` 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	`double` as the default data type for the input

Ports

Input

`Port_1` — First input signal

scalar | vector | matrix

First input signal, specified as a scalar, vector, or matrix.

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

`Port_2` — Second input signal

scalar | vector | matrix

Second input signal, specified as a scalar, vector, or matrix.

Dependencies

To enable this port, set the **Relational operator** to ==, ~=, <, <=, >=, or >.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal, consisting of zeros and ones, with the same dimensions as the input. You control the output data type with the **Output data type** parameter.

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

Parameters

Main

`Relational operator` — Relational operator

<= (default) | ~= | == | < | >= | > | `isInf` | `isNaN` | `isFinite`

Specify the operation for comparing two inputs or determining the signal type of one input.

- == — 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

Programmatic Use

Block Parameter: `Operator`
Type: character vector
Values: `'==' | '~=' | '<' | '<=' | '>=' | '>' | 'isInf' | 'isNaN' | 'isFinite'`

Default: `'<='`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | `off`

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

Programmatic Use
Parameter:`ZeroCross`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Data Type

`Require all inputs to have the same data type` — Require all inputs to have same data type

`off` (default) | `on`

To require that all block inputs have the same data type, select this check box. When you clear this check box, the inputs can have different data types.

Dependencies

This check box is not available when you set **Relational operator** to `isInf`, `isNaN`, or `isFinite` because in those modes, the block only has one input port.

Programmatic Use

Block Parameter: `InputSameDT`
Type: character vector
Values: `'off' | 'on`
Default: `'off'`

`Output data type` — Output data type

`boolean` (default) | `Inherit: Logical (see Configuration Parameters: Optimization)` | `fixdt(1,16)` | `<data type expression>`

Specify the output data type. When you select:

- `boolean` — the block output has data type `boolean`.
- `Inherit: Logical (see Configuration Parameters: Optimization)` — the block 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.
- `fixdt(1,16)` — the block output has the specified fixed-point data type `fixdt(1,16)`.

  Tip Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.
- `<data type expression>` — the block output has the data type you specify as a data type expression, 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`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Logical (see Configuration Parameters: Optimization)' | 'boolean' | 'fixdt(1,16)' | '<data type expression>'`
Default: `'boolean'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Simplest` (default) | `Convergent` | `Floor` | `Nearest` | `Round` | `Ceiling` | `Zero`

Specify the rounding mode for fixed-point operations. You can select:

`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.

For more information, see “Rounding” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Simplest'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`Yes`

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™.

For more information about HDL code generation, see Relational Operator.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

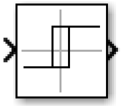
See Also

Logical Operator

Introduced before R2006a

Relay

Switch output between two constants
Library: Simulink / Discontinuities



Description

The output for the Relay block switches 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.

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.

Ports

Input

`Port_1` — Input signal
scalar

The input signal that switches the relay on or off.

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

Output

`Port_1` — Output signal
scalar

The output signal switches between two values determined by the parameters **Output when on** and **Output when off**.

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

Parameters

`Switch on point` — Input value which switches the relay on

`'eps'` (default) | `scalar`

When the input crosses this threshold, the relay switches on. 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.

The **Switch on point** parameter is converted to the input data type offline using round-to-nearest and saturation methods.

Programmatic Use
Block Parameter: `OnSwitchValue`
Type: character vector
Values: scalar
Default: `'eps'`

`Switch off point` — Input value which switches the relay off

`'eps'` (default) | `scalar`

When the input crosses this threshold the relay switches off. The value of **Switch off point** must be less than or equal to **Switch on point**. The **Switch off point** parameter is converted to the input data type offline using round-to-nearest and saturation.

Programmatic Use
Block Parameter: `OffSwitchValue`
Type: character vector
Values: scalar
Default: `'eps'`

`Output when on` — Output value when the relay is on

1 (default) | `scalar`

The output value when the relay is on.

Programmatic Use
Block Parameter: `OnOutputValue`
Type: character vector
Values: scalar
Default: `'1'`

`Output when off` — Output value when the relay is off

`0` (default) | `scalar`

The output value when the relay is off.

Programmatic Use
Block Parameter: `OffOutputValue`
Type: character vector
Values: scalar
Default: `'0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`

Type: character vector
Values: scalar
Default: `'-1'`

`Output minimum` — Minimum output value for range checking

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Data type of output signal

`Inherit: All ports same datatype` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>` | ...

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.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Same as input'`, `'Inherit: Inherit via back propagation'`, `'single'`, `'int8'`, `'uint8'`, `int16`, `'uint16'`, `'int32'`, `'uint32'`, `fixdt(1,16,0)`, `fixdt(1,16,2^0,0)`, `fixdt(1,16,2^0,0)`. `'<data type expression>'`
Default: `'Inherit: Same as input'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types
`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

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™.

For more information on HDL code generation, see Relay.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

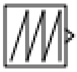
See Also

Backlash | Saturation

Introduced before R2006a

Repeating Sequence

Generate arbitrarily shaped periodic signal
Library: Simulink / 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`.

Ports

Output

`Port_1` — Periodic output signal
scalar

Output signal specified by the **Time values** and **Output values** parameters to create a periodic scalar signal.

Data Types: `double`

Parameters

`Time values` — Vector of output times

`[0 2]` (default) | vector

Vector of strictly monotonically increasing time values. The period of the generated waveform is the difference of the last and first values of this parameter.

Programmatic Use

Block Parameter: `rep_seq_t`
Type: character vector
Values: vector
Default: `[0 2]`

`Output values` — Vector of output values

`[0 2]` (default) | vector

Vector of output values that specify the output waveform. Each element corresponds to the time value in the **Time values**parameter.

Programmatic Use

Block Parameter: `rep_seq_y`
Type: character vector
Values: vector
Default: `[0 2]`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Algorithms

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*period`, where `n` is an integer. The sequence repeats at `t = n*period`. The block uses linear interpolation to compute the value of the waveform between the output times that you specify.

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

Consider using Repeating Sequence Stair or Repeating Sequence Interpolated blocks for code generation.

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: Simulink / Sources

Description

The Repeating Sequence Interpolated block outputs a periodic discrete-time sequence based on the values in **Vector of time values** and **Vector of output values** parameters. Between data points, the block uses the method you specify for the **Lookup Method** parameter to determine the output.

Ports

Output

`Port_1` — Periodic output signal
scalar

Output signal generated based on the values in the **Vector of time values** and **Vector of output values** parameters.

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

Parameters

Main

`Vector of output values` — Vector of output values

`[3 1 4 2 1].'` (default) | vector

Vector of output values that specify the output waveform. Each element corresponds to the time value in the **Time values**parameter.

Programmatic Use

Block Parameter: `OutValues`
Type: character vector
Values: vector
Default: `[3 1 4 2 1].'`

`Vector of time values` — Vector of time values

`[0 0.1 0.5 0.6 1].'` (default) | vector

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.

Programmatic Use

Block Parameter: `TimeValues`
Type: character vector
Values: vector
Default: `[0 0.1 0.5 0.6 1].'`

`Lookup Method` — Lookup method for output

`Interpolation-Use End Values` (default) | `Use Input Nearest` | `Use Input Below` | `Use Input Above`

Specify the lookup method to determine the output between data points.

Programmatic Use

Block Parameter: `LookUpMeth`
Type: character vector
Values: `'Interpolation-Use End Values'` | `'Use Input Nearest'` | `Use Input Below` | `Use Input Above`
Default: `'Interpolation-Use End Values'`

`Sample time` — Time interval between samples

`0.01` (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: `tsamp`
Type: character vector
Values: scalar
Default: `'0.01'`

Signal Attributes

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar

Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[ ]` (default) | scalar

Upper value of the output range that Simulink checks.

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" (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`double` (default) | `Inherit: Inherit via back propagation` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>` | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use

Block Parameter: `OutDataTypeStr`
Type: character vector

Values: , 'Inherit: Inherit via back propagation', 'single', 'int8', 'uint8', int16, 'uint16', 'int32', 'uint32', fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0). '<data type expression>'
Default: 'Double'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

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).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Cannot be used inside a triggered subsystem hierarchy.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

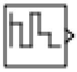
See Also

Repeating Sequence | Repeating Sequence Stair

Introduced before R2006a

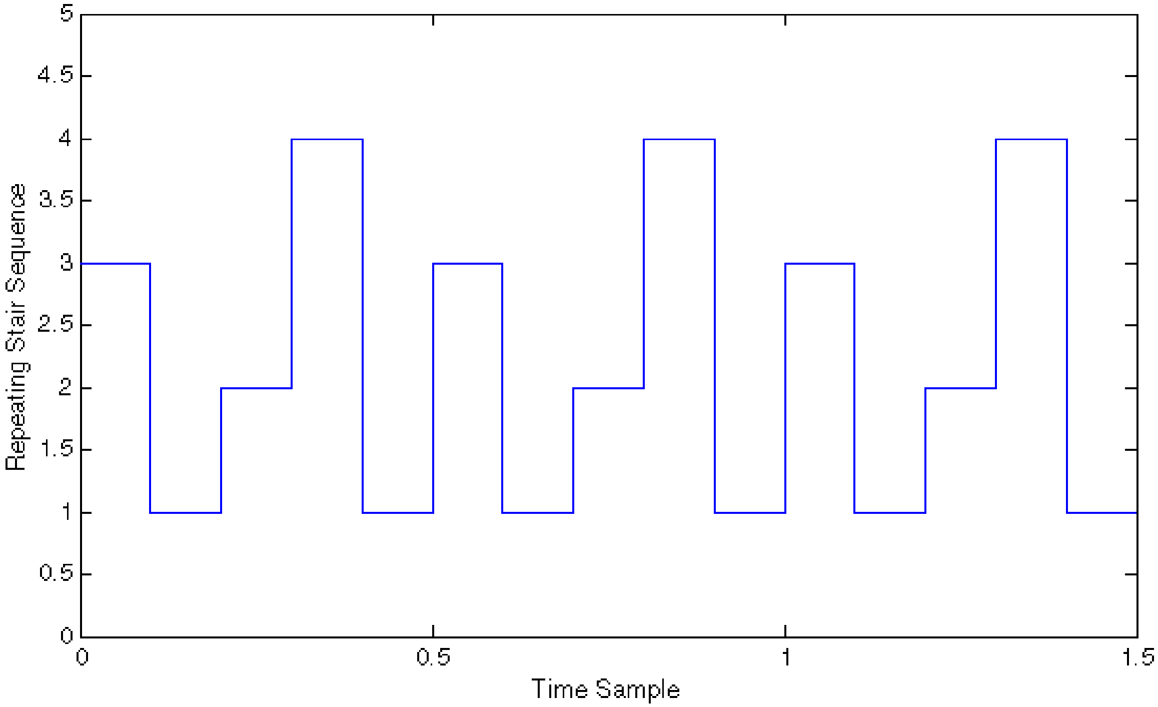
Repeating Sequence Stair

Output and repeat discrete time sequence
Library: Simulink / 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]'`. A value in **Vector of output values** is output at each time interval, and then the sequence repeats.



Ports

Output

`Port_1` — Repeating discrete output signal
scalar

Output signal generated based on the values in the **Vector of time values** and **Sample time** parameters.

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

Parameters

Main

`Vector of output values` — Vector of output values

`[3 1 4 2 1].'` (default) | vector

Specify the vector containing values of the repeating stair sequence.

Programmatic Use
Block Parameter: `OutValues`
Type: character vector
Values: vector
Default: `[3 1 4 2 1].'`

`Sample time` — Time interval between samples

`-1` (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: `tsamp`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

`Output minimum` — Minimum output value for range checking

`[ ]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`double` (default) | `Inherit: Inherit via back propagation` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>` | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use

Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via back propagation'` | `'single'` | `'int8'` | `'uint8'` | `int16` | `'uint16'` | `'int32'` | `'uint32'` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `fixdt(1,16,2^0,0)` | `'<data type expression>'`
Default: `'Double'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

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).

Programmatic Use

Block Parameter: `LockScale`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

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

Block 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

"Use Resettable Subsystems"
"Simulink Block Support for Variable-Size Signals"

Introduced in R2015a

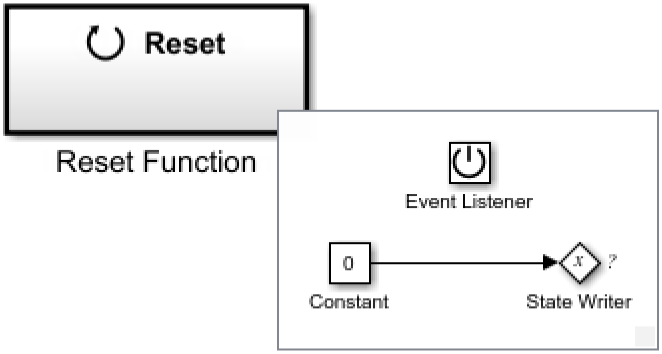
Reset Function

Executes contents on a model reset event

Library: Simulink / User-Defined Functions

Description

The Reset Function block is a pre-configured subsystem block that executes on a model reset event. By default, 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.

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	No
Zero-Crossing Detection	No

[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™.

See Also

Event Listener | Initialize Function | State Reader | State Writer | Terminate Function

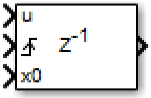
Topics

“Customize Initialize, Reset, and Terminate Functions”
“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: Simulink / 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.

Ports

Input

u — Data input signal
scalar | vector

Input data signal delayed according to parameters settings.

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

d — Delay length
scalar

Delay length specified as inherited from an input port. Enabled when you select the **Delay length: Source** parameter as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Enable — External enable signal
scalar

Enable signal that enables or disables execution of the block. To create this port, select the **Show enable port** parameter.

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

`External reset` — External reset signal

scalar

External signal that resets execution of the block to the initial condition. To create this port, select the **External reset** parameter.

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

`x0` — Initial condition

scalar | vector

Initial condition specified as inherited from an input port. Enabled when you select the **Initial Condition: Source** parameter as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal

scalar | vector

Output signal that is the input signal delayed by the length of time specified by the parameter **Delay length**. The initial value of the output signal depends on several conditions. See “Initial Block Output” on page 1-340.

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

Parameters

Main

`Delay length` — Delay length

`Dialog` (default) | `Input port`

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.

Programmatic Use
Block Parameter: `DelayLengthSource`
Type: character vector
Values: `'Dialog'` | `'Input port'` |
Default: `'Dialog'`
Block Parameter: `DelayLength`
Type: character vector
Values: scalar
Default: `'2'`
Block Parameter: `DelayLengthUpperLimit`
Type: character vector
Values: scalar
Default: `'100'`

`Initial condition` — Initial condition

`Input port` (default) | `Dialog`

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.

Programmatic Use
Block Parameter: `InitialConditionSource`
Type: character vector
Values: `'Dialog'` | `'Input port'` |
Default: `'Input Port'`
Block Parameter: `InitialCondition`
Type: character vector
Values: scalar
Default: `''`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Use circular buffer for state` — Circular buffer for storing state

`off` (default) | `on`

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.

Programmatic Use

Block Parameter: `UseCircularBuffer`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Prevent direct feedthrough` — Prevent direct feedthrough

`off` (default) | `on`

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.

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use

Block Parameter: `PreventDirectFeedthrough`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Remove delay length check in generated code` — Remove delay length out-of-range check

`off` (default) | `on`

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

Check Box	Result	When to Use
Cleared	Generated code includes conditional statements to check for out-of-range delay length.	For safety-critical applications

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use
Block Parameter: `RemoveDelayLengthCheckInGeneratedCode`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Diagnostic for delay length` — Diagnostic checks for delay length

`None` (default) | `Warning` | `Error`

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.

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use
Block Parameter: `DiagnosticForDelayLength`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`

Default: `'None'`

`Show enable port` — Create enable port

`off` (default) | `on`

Select to control execution of this block with an enable port. 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.

Programmatic Use

Block Parameter: `ShowEnablePort`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`External reset` — External state reset

`Rising` (default) | `None` | `Falling` | `Either` | `Level` | `Level hold`

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: • 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.

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector
Values: `'None'` | `'Rising'` | `'Falling'` | `'Either'` | `'Level'` | `'Level hold'`
Default: `'Rising'`

`Sample time (-1 for inherited)` — Discrete interval between sample time hits

`-1` (default) | `scalar`

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.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Value: real scalar
Default: `'-1'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: `StateName`
Type: character vector

Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, specify a value for **State name**. 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**.

Programmatic Use
Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default) | `<StorageClass.PackageName>`

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use
Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`

Default: 'Simulink.Signal'

Code generation storage class — State storage class for code generation
Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Model default | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.

- 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 "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder). For information about custom storage classes, see "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

Dependencies

To enable this parameter, specify a value for **State name**.

Programmatic Use
Block Parameter: StateStorageClass
Type: character vector
Values: 'Auto' | 'SimulinkGlobal' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...
Default: 'Auto'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated | bus
Direct Feedthrough	Yes

Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Consider using the Model Discretizer to map these continuous blocks into discrete equivalents that support code generation. From a model, select **Analysis > Control Design > Model Discretizer**.

Not reommended for production code.

HDL Code Generation

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

For information about HDL code generation, see Delay.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Delay | Unit Delay | Variable Integer Delay

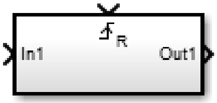
Topics

"Using Enabled Subsystems"

Introduced in R2012b

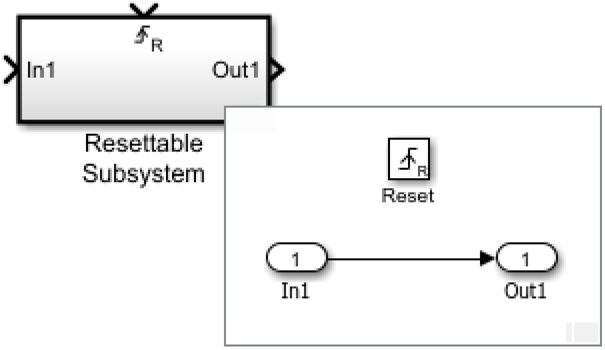
Resettable Subsystem

Subsystem whose block states reset with external trigger
Library: Simulink / 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

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]

Zero-Crossing Detection	`No`

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 code generation support depends on block implementation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks

Reset | Subsystem

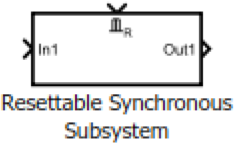
Topics

"Use Resettable Subsystems"

Introduced in R2015a

Resettable Synchronous Subsystem

Represent resettable subsystem that has synchronous reset and enable behavior



Library

HDL Coder / HDL Subsystems

Description

The Resettable Synchronous Subsystem block uses the `Synchronous` mode of the State Control block with the Resettable Subsystem block. If an **S** symbol appears in the subsystem, then it is synchronous. For more information about the block in HDL Coder, see Resettable 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: Simulink / Math Operations

Description

The Reshape block changes the dimensionality of the input signal to a dimensionality that you specify, using the **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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal whose dimensions are changed based on the **Output dimensionality** parameter.

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

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal created with the dimensions specified in the **Output dimensionality** parameter.

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

Parameters

`Output dimensionality` — Dimensions of output signal

`1-D array` (default) | `Column vector (2-D)` | `Row vector (2-D)` | `Customize` | `Derive from reference input port`

Specify the dimensionality of the output signal.

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, and so on. (This option leaves a vector input unchanged.)
Column vector	Converts a vector, matrix, or multidimensional input signal to a column matrix, a *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.
Row vector	Converts a vector, matrix, or multidimensional input signal to a row matrix, 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.
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).

Programmatic Use
Block Parameter: `OutputDimensionality`
Type: character vector
Value: `'1-D array'` | `'Column vector (2-D)'` | `'Row vector (2-D)'` | `'Customize'` | `'Derive from reference input port'`
Default: `'1-D array'`

`Output dimensions` — Custom dimensions of output signal

`[1,1]` (default) | `[integer]` | `[integer,integer]`

Specify the dimensions for the output signal. The value 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.

Dependency

To enable this parameter, set **Output dimensionality** to `Customize`.

Programmatic Use
Block Parameter: `OutputDimensions`
Type: character vector
Value: `'[integer,intger]'` |
Default: `'[1,1]'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
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™.

For more information on HDL code generation, see Reshape.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Squeeze

Topics

"Combine Buses into an Array of Buses"

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.

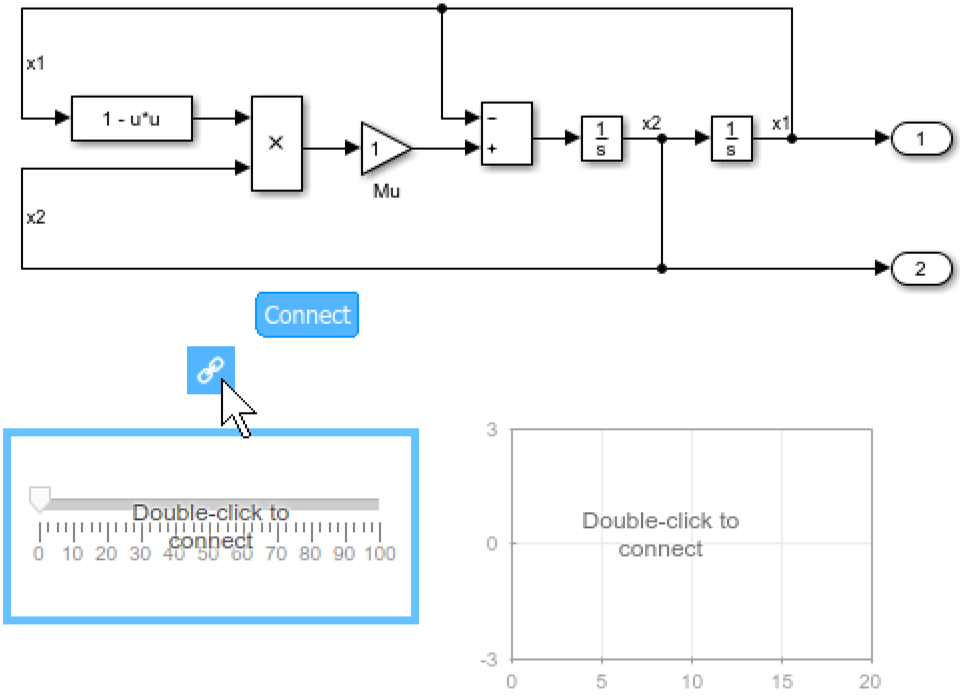
Double-clicking the Rocker Switch block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

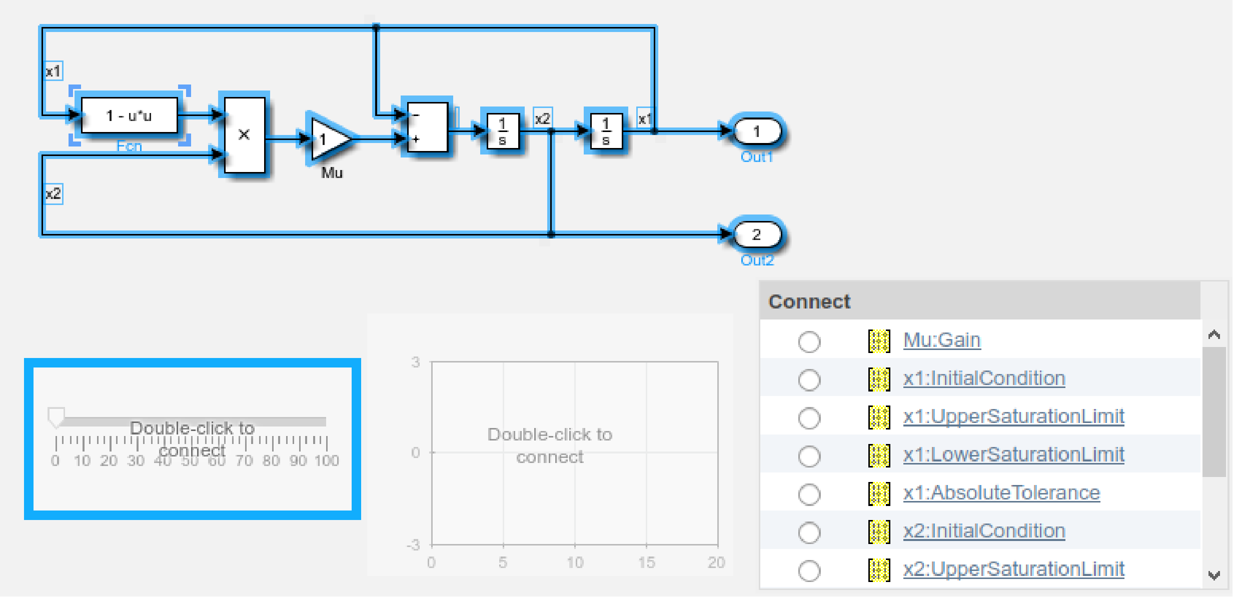
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○ Mu:Gain
○ x1:InitialCondition
○ x1:UpperSaturationLimit
○ x1:LowerSaturationLimit
○ x1:AbsoluteTolerance
○ x2:InitialCondition
○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

See Also

Rotary Switch | Slider Switch | Toggle Switch

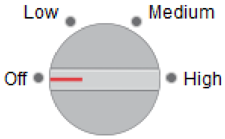
Topics

"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.

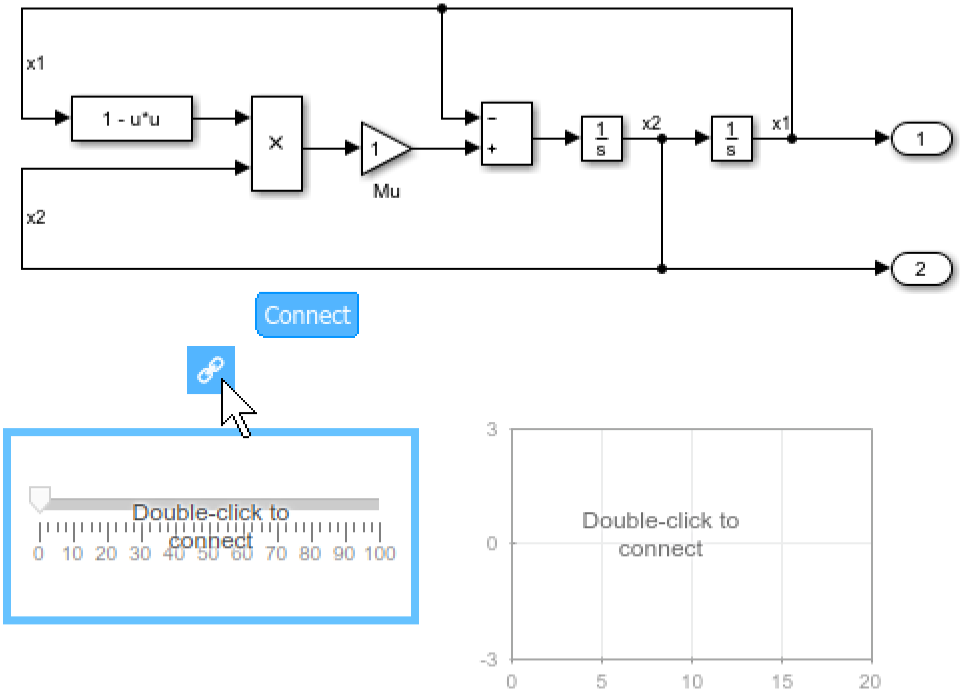
Double-clicking the Rotary Switch block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

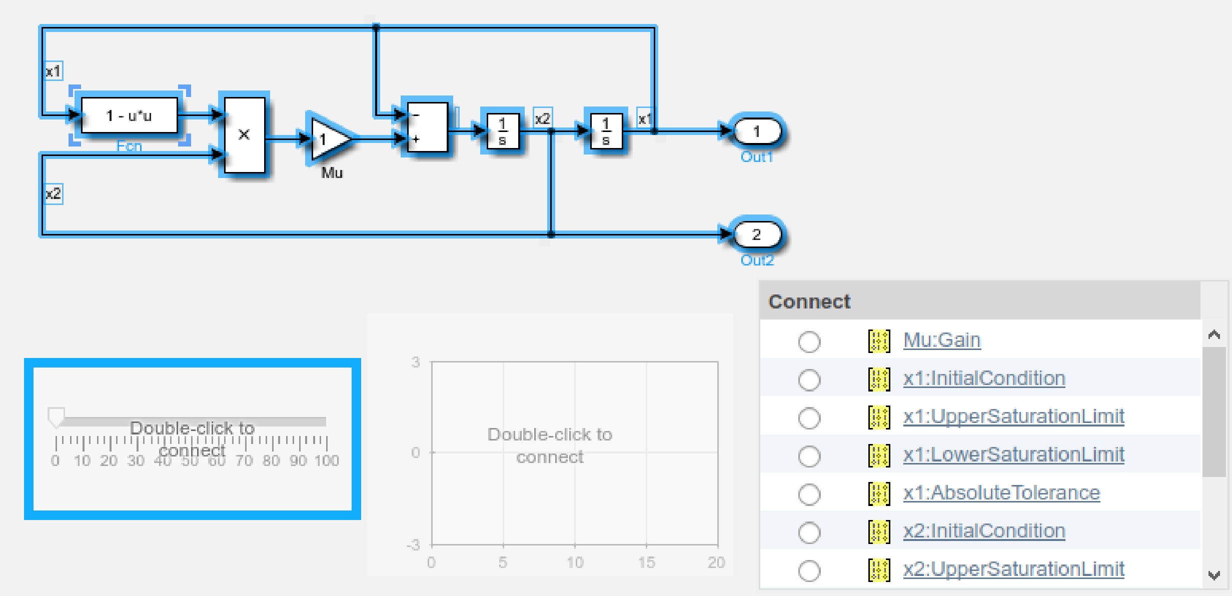
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

- ○ Mu:Gain
- ○ x1:InitialCondition
- ○ x1:UpperSaturationLimit
- ○ x1:LowerSaturationLimit
- ○ x1:AbsoluteTolerance
- ○ x2:InitialCondition
- ○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

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: Simulink / Math Operations

Description

The Rounding Function block rounds each element of the input signal to produce the output signal.

You select the type of rounding from the **Function** parameter list. The name of the selected function appears on the block.

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`.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal to which the rounding function is applied.

Data Types: `single` | `double`

Output

Port_1 — Output signal
scalar | vector | matrix

Output signal after the rounding function is applied to the input signal. The output signal has the same dimensions and data 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.

Data Types: `single` | `double`

Parameters

Function — Rounding function
`floor` (default) | `ceil` | `round` | `fix`

Choose the rounding function applied to the input signal.

Rounding function	Rounds each element of the input signal
`floor`	To the nearest integer value towards minus infinity
`ceil`	To the nearest integer towards positive infinity
`round`	To the nearest integer
`fix`	To the nearest integer towards zero

Programmatic Use
Block Parameter: `Operator`
Type: character vector
Values: `'floor'` | `'ceil'` | `'round'` | `'fix'`
Default: `'floor'`

Sample time — Specify sample time as a value other than -1
`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
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™.

For more information on HDL code generation, see Rounding Function.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

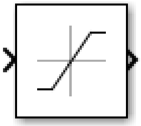
Fcn

Introduced before R2006a

Saturation

Limit input signal to the upper and lower saturation values

Library: Simulink / Commonly Used Blocks
Simulink / Discontinuities



Description

The Saturation block produces an output signal that is the value of the input signal bounded to the upper and lower saturation values. The upper and lower limits are specified by the parameters **Upper limit** and **Lower limit**.

Input	Output
Lower limit ≤ Input value ≤ Upper limit	Input value
Input value < Lower limit	Lower limit
Input value > Upper limit	Upper limit

Ports

Input

`Port_1` — Input signal
scalar | vector

The input signal to the saturation algorithm.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector

Output signal that is the value of the input signal, upper saturation limit, or lower saturation limit.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Main

`Upper limit` — Upper saturation boundary for the input signal
`0.5` (default) | scalar | vector

Specify the upper bound on the input signal. If the input signal is greater than this boundary, then the output signal is set to this saturation value. The **Upper limit** parameter is converted to the output data type using round-to-nearest and saturation. **Upper limit** must be greater than the **Output minimum** parameter and less than the **Output maximum** parameter.

Programmatic Use
Block Parameter: `UpperLimit`
Type: character vector
Value: real scalar or vector
Default: `'0.5'`

`Lower limit` — Lower saturation boundary for the input signal
`-0.5` (default) | scalar | vector

Specify the lower bound on the input signal. If the input signal is less than this boundary, then the output signal is set to this saturation value. The **Lower limit** parameter is converted to the output data type using round-to-nearest and saturation. **Lower limit** must be greater than the **Output minimum** parameter and less than the **Output maximum** parameter.

Programmatic Use

Block Parameter: `LowerLimit`
Type: character vector
Value: real scalar or vector
Default: `'-0.5'`

`Treat as gain when linearizing` — Specify the gain value

`On` (default) | Boolean

Select this check box to cause the commands to treat the gain as 1. The linearization commands in Simulink software treat this block as a gain in state space. Clear the box to have the commands treat the gain as 0.

Programmatic Use

Block Parameter: `LinearizeAsGain`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use

Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Signal Attributes

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Output minimum` — Minimum output value for range checking

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Same as input` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>` | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Same as input'`, `'Inherit: Inherit via back propagation'`, `'single'`, `'int8'`, `'uint8'`, `int16`, `'uint16'`, `'int32'`, `'uint32'`,

fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0). '<data type expression>'
Default: 'Inherit: Same as input'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Specify the rounding mode for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Choose one of these rounding modes.

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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

See Also

For more information, see “Rounding” (Fixed-Point Designer).

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

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™.

For more information on HDL code generation, see Saturation.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

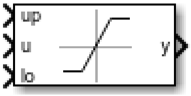
Backlash | Saturation Dynamic

Introduced before R2006a

Saturation Dynamic

Limit input signal to dynamic upper and lower saturation values

Library: Simulink / Discontinuities



Description

The Saturation Dynamic block produces an output signal that is the value of the input signal bounded to the saturation values from the input ports `up` and `lo`.

Input	Output
`lo` ≤ Input value ≤ `hi`	Input value
Input value < `lo`	Lower limit
Input value > `hi`	Upper limit

Ports

Input

u — Input signal
scalar | vector

The input signal to the saturation algorithm.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

up — Signal that provides the upper saturation limit
scalar | vector

Dynamic value providing the upper saturation limit. When the input is greater than **up** then the output value is bound to **up**.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`
Complex Number Support: Yes

`lo` — Signal that provides the lower saturation limit

scalar | vector

Dynamic value providing the lower saturation limit. When the input is less than **lo** then the output value is bound to **lo**.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`
Complex Number Support: Yes

Output

`Output 1` — Output signal

scalar | vector

Output signal that is the value of the input signal, upper saturation limit, or lower saturation limit.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

To edit the parameters for the Saturation Dynamic block, double-click the block icon.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Same as input` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>` | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Same as input'`, `'Inherit: Inherit via back propagation'`, `'single'`, `'int8'`, `'uint8'`, `int16`, `'uint16'`, `'int32'`, `'uint32'`, `fixdt(1,16,0)`, `fixdt(1,16,2^0,0)`, `fixdt(1,16,2^0,0)`. `'<data type expression>'`
Default: `'Inherit: Same as input'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Specify the rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Choose one of these rounding modes.

`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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

See Also

For more information, see “Rounding” (Fixed-Point Designer).

`Saturate on integer overflow` — Choose the behavior when integer overflow occurs

`on` (default) | boolean

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 `int8` (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.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Check for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean`[a] | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[a] This block is not recommended for use with Boolean signals.

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™.

For more information on HDL code generation, see Saturation Dynamic.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Saturation

Introduced before R2006a

Scan String

Scan input string and convert to signals per specified format

Library: Simulink / String

str
scan
"%d %f"
d
f

Description

The Scan String block scans an input string and converts it to signals per the format specified by the **Format** parameter. The block converts values to their decimal (base 10) representation and outputs the results as numeric or string signals. Use this block when you want to deconstruct a string, for example a sentence, into its individual components. For example, if the **Format** parameter is set to `"%s is %f."`, the block outputs two parts, a string signal and a single signal. If the input is the string `"Pi is 3.14"`, the two outputs are `"Pi"` and `"3.14"`.

The Scan String, String to Double, and String to Single blocks are identical blocks. When configured for String to Double, the block converts the input string signal to a double numerical output. When configured for String to Single, the block converts the input string signal to a single numerical output.

For code generation, configure models that contain this block for non-finite number support by selecting the **Configuration Parameters > Code Generation > Interface > Support non-finite numbers** check box.

Ports

Input

`Port_1` — Input string
scalar

Input string, specified as a scalar.

Data Types: `string`

Output

`d` — Output data whose format matches `%d` format

scalar

Output data whose format matches the specified format, defined as a scalar. Total maximum number of outputs is 128.

If the block cannot match an input string to a format operator specified in **Format**, it returns a warning and outputs an appropriate value (`0` or `""`) for each unmatched format operator.

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

`f` — Output data whose format matches %f format

scalar

Output data whose format matches the `%f` format, specified as a scalar. Total maximum number of outputs is 128.

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

`Port_N` — Output data whose format matches N format

scalar

Output data whose format matches *N* format, specified as a scalar. Total maximum number of outputs is 128.

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

Parameters

`Format` — Format operator for input

`"%d %f"` (default) | scalar

Format operator for input, specified as a scalar. If the block cannot match the input string with the specified format, it returns `0`. The return of `0` differs from the `sscanf` function return, which is an empty matrix if the function cannot match the input with the specified format.

- For the String to Double block, this parameter has a default value of `%lf`.
- For the String to Single block, this parameter has a default value of `%f`.

For more information about acceptable format operators, see the Algorithms section.

Block Characteristics

Data Types	`double | single | base integer | string`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Algorithms

The Scan String block uses this format specifier prototype:

```
%[width][length]specifier
```

Numeric Fields

This table lists available conversion specifiers to convert text to numeric outputs. The block converts values to their decimal (base 10) representation.

Output Port Data Type	Conversion Specifier	Description
Integer, signed	%d	Base 10
Integer, unsigned	%u	Base 10

Output Port Data Type	Conversion Specifier	Description
Floating-point number	`%f`, `%e`, or `%g`	Floating-point values. Input fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`. Input fields that represents floating-point numbers can include leading `+` or `-` symbols and exponential notation using `e` or `E`. The conversion specifiers `%f`, `%e`, and `%g` all treat input fields the same way.

Character Fields

This table lists available conversion specifiers to convert text so that the output is a string.

Character Field Type	Conversion Specifier	Description
String scalar	`%s`	Read the text until the block encounters whitespace.
	`%c`	Read any single character, including whitespace. To read multiple characters at a time, specify field width. For example, `%10c` reads 10 characters at a time.
Pattern-matching	`%[...]`	Read only the characters in the brackets up to the first nonmatching character or whitespace. Example: `%[mus]` reads `'summer'` as `'summ'`.

Character Field Type	Conversion Specifier	Description
	`%[^...]`	Do not read characters in the brackets up to the first nonmatching character or whitespace. Example: `%[m]` reads `'summer'` as `'su'`.

Optional Operators

- Field Width — To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, `%10s` reads up to 10 characters at a time, including whitespace. `%4f` reads up to four digits at a time, including the decimal point.
- Literal Text to Ignore — This block must match the specified text immediately before or after the conversion specifier.

  Example: `Hell%s` reads `"Hello!"` as `"o!"`.

Length Specifiers

The Scan String block supports the `h` and `l` length subspecifiers. These specifiers can change according to the **Configuration Parameters** > **Hardware Implementation** > **Number of bits** settings.

Length	i	u	f e g	s c [...] [^...]
No length specifier	`int`	`unsigned int`	`single`	`string`
`h`	`short`	`unsigned short`	—	—
`l`	`long`	`unsigned long`	`double`	—

Notes for Specifiers that Specify Integer Data Types (d, u)

- Target `int`, `long`, and `short` type sizes are controlled by settings in the **Configuration Parameters** > **Hardware Implementation** pane. For example, if the target `int` is 32 bits and the specifier is `%u`, then the expected input type will be `uint32`. For this example, the Scan String block requires that the output type be exactly `int32`. It cannot be any other data type.

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String to Double | String to Enum | String to Single | Substring | To String | `sscanf`

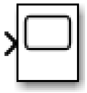
Topics

"Display and Extract Coordinate Data"
"Simulink Strings"

Introduced in R2018a

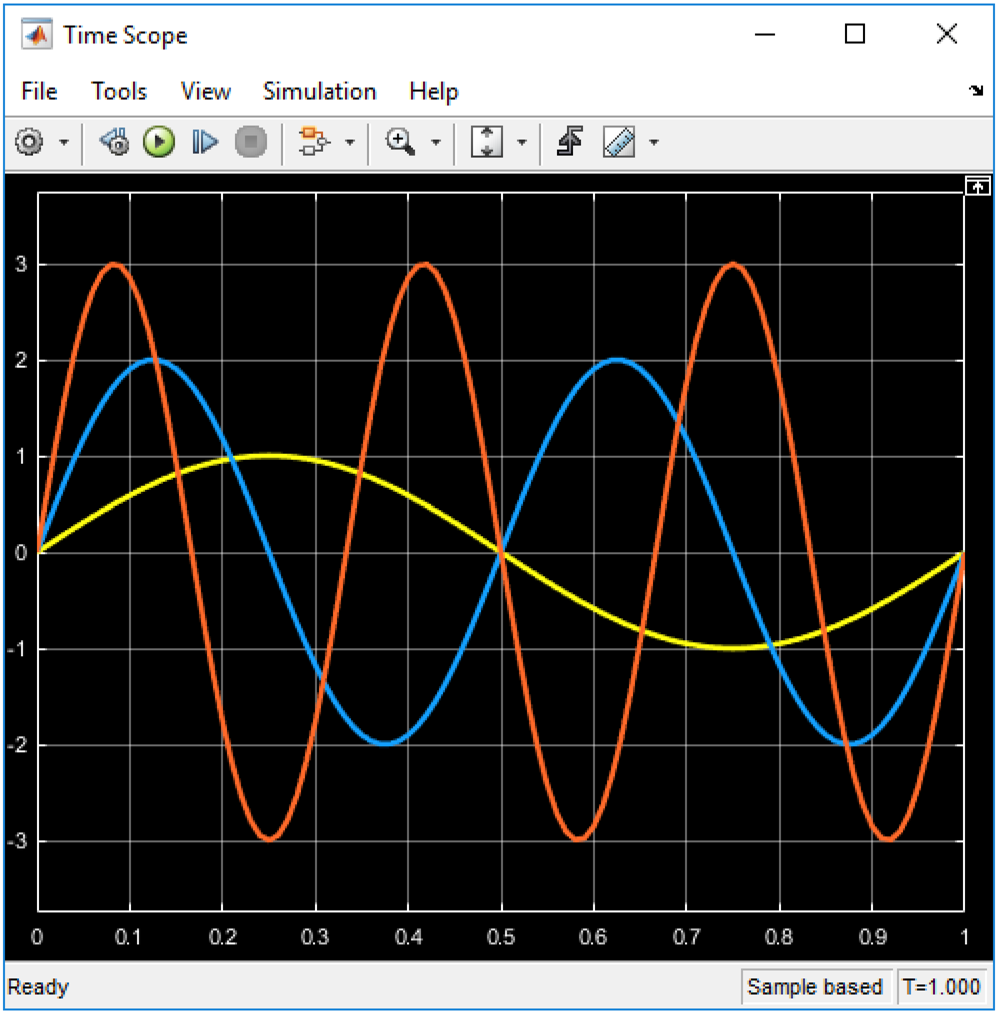
ScopeTime Scope

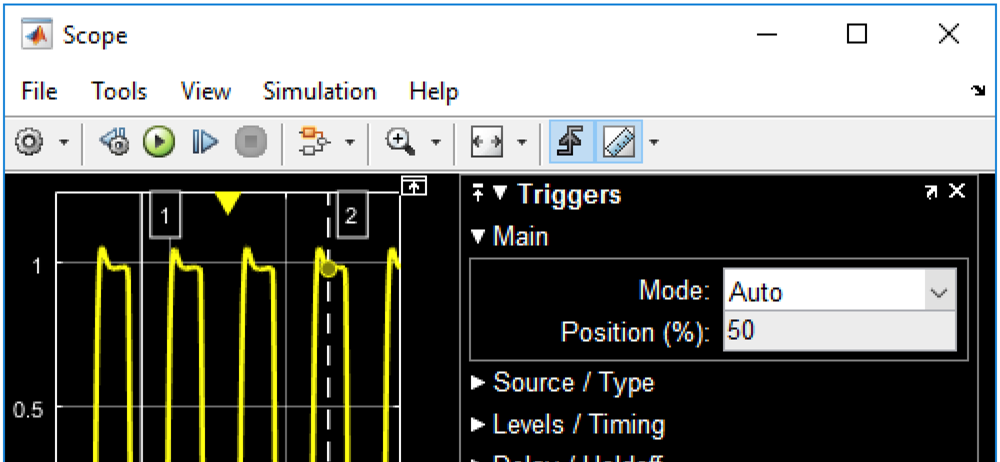
Display signals generated during simulation
Library: Simulink / Commonly Used Blocks
Simulink / Sinks



Description

The Simulink Scope block and DSP System Toolbox Time Scope block display time domain signals.





Mode: Auto

Position (%): 50

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 Scope Blocks 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 (array). 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.

Programmatic Use

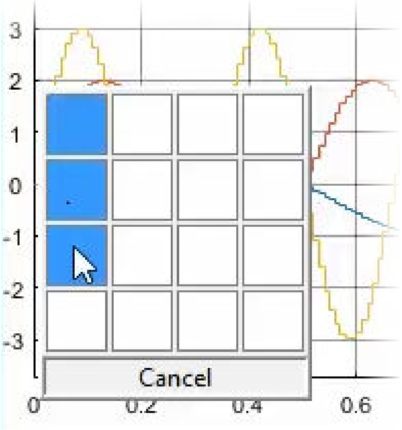
See NumInputPorts.

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 internal 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` — Enter any 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 the data from the left.

- `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 in 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

To activate this property, set:

- “Show time-axis label” on page 1-0 to on.
- “Maximize axes” on page 1-0 to off.

The “Active display” on page 1-0 property determines which display is affected.

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

To activate this property, set “Time-axis labels” on page 1-0 to `All` or `Bottom Displays Only`.

The “Active display” on page 1-0 property determines which display is affected.

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. For layouts with multiple columns and rows, display numbers are mapped down and then across.

Programmatic Use

See “Active display” on page 1-0 .

`Title` — Display name

`%<SignalLabel>` (default) | character vector | string

Title for a 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

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]`.

The “Active display” on page 1-0 property determines which display is affected.

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

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]`.

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See YLimits.

`Y-label` — Y-axis label

none (default for Scope) | `'Amplitude'` (default for Time Scope) | character vector | string

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

If you select **Plot signals as magnitude and phase**, this property does not apply. The *y*-axes are labeled `Magnitude` and `Phase`.

The “Active display” on page 1-0 property determines which display is affected.

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 `N`th 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. Use the **Dataset signal format** configuration parameter to select the 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 the *y*-axis range with the **Scale Y-axis Limits** toolbar button.
- `Auto` — Scale the *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 use for plotting

`80` (default) | integer between [1, 100]

Specify the percentage of the *y*-axis range used for plotting data. If you set this property to `100`, the 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 the maximum values of the *y*-axis range.
- `Center` — Center signals between the minimum and maximum values.
- `Bottom` — Align signals with the minimum values of the *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`, the data currently within the axes is scaled, not the entire signal in the data buffer.

`X-axis Data range (%)` — Percent of *x*-axis to use for plotting

`100` (default) | integer in the range [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 the maximum values of the *x*-axis range.
- `Center` — Center signals between the minimum and maximum values.
- `Bottom` — Align signals with the minimum values of the *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

solid line (default style) | 0.75 (default width) | yellow (default 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 shape.

Dependency

The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

[a] Virtual bus not supported. Nonvirtual bus supported only in normal and accelerator mode simulation. Data logging for nonvirtual bus supported only in the dataset format

Extended Capabilities

HDL Code Generation

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

For more information about HDL code generation, see Scope.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Floating Scope | Scope Viewer

Topics

“Scope Blocks and Scope Viewer Overview”
“Step Through a Simulation”
“Common Scope Block Interactions”
“Control Scope Blocks Programmatically”
Scope Block with Apple iOS Devices (Simulink Support Package for Apple iOS Devices)

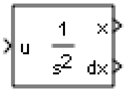
“Simulink Scope Block and Mobile Apps” (Simulink Support Package for Android Devices)

Introduced in R2015b

Second-Order Integrator

Integrate input signal twice

Library: Simulink / Continuous



Description

The Second-Order Integrator block and the Second-Order Integrator Limited block solve the second-order initial value problem:

$$\frac{d^2x}{dt^2} = u,$$

$$\left.\frac{dx}{dt}\right|_{t=0} = dx_o,$$

$$\left.x\right|_{t=0} = x_o,$$

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*. 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 1-1651.

Simulink software can use several 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 “Solver Types”.) 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:

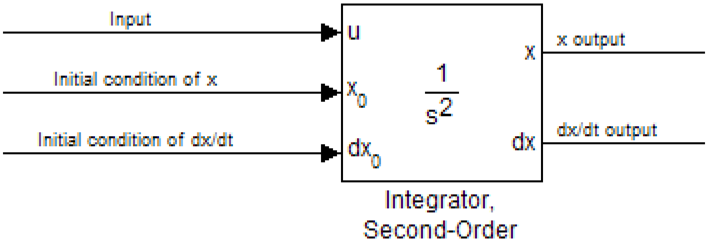
- Specify whether the source of each state initial condition is internal or external
- Specify a value for the state initial conditions
- Define upper and lower limits on either or both states
- Specify absolute tolerances for each state
- Specify names for both states
- Choose an external reset condition
- 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.



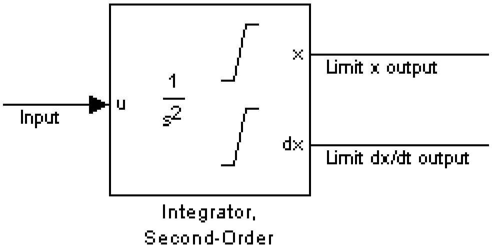
Note

- 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 1-1651) 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.

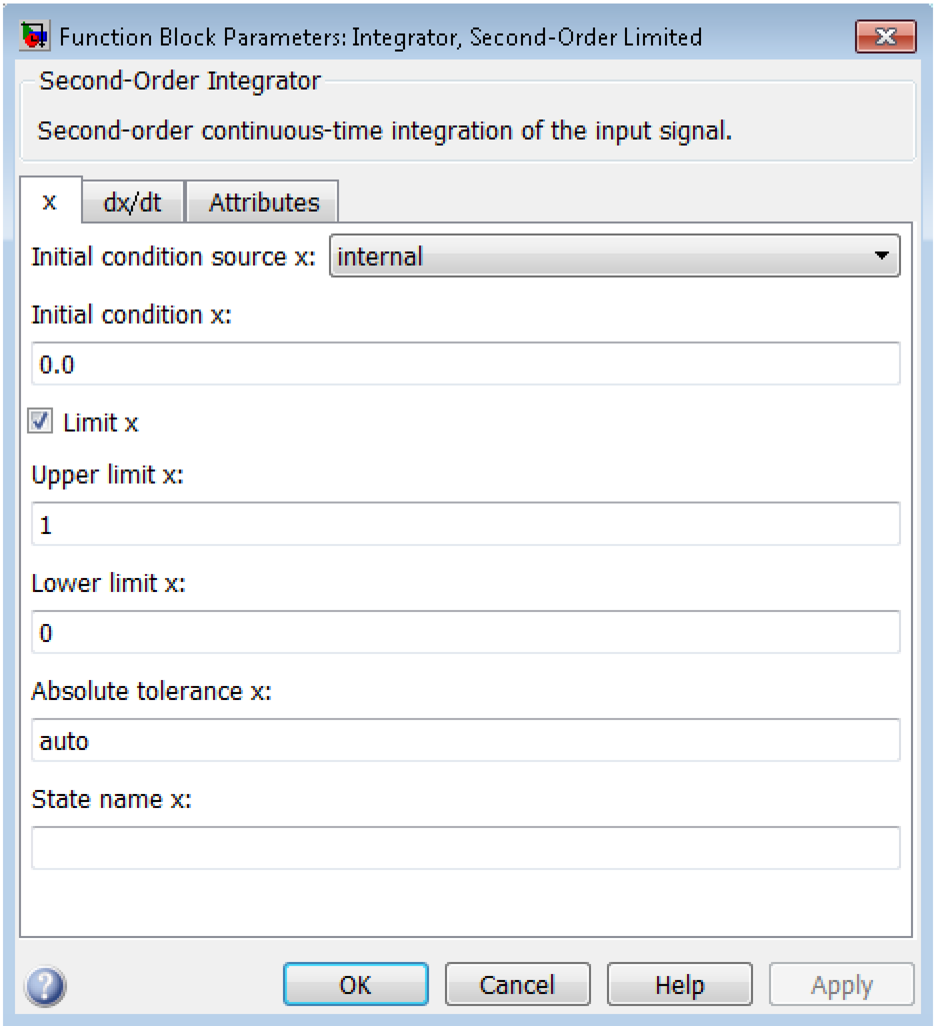
The block appearance changes when you limit one or both states. With both states limited, the block appears as follows.



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.



Function Block Parameters: Integrator, Second-Order Limited

Second-Order Integrator

Second-order continuous-time integration of the input signal.

Initial condition source x: internal

Initial condition x:

0.0

☑ Limit x

Upper limit x:

1

Lower limit x:

0

Absolute tolerance x:

auto

State name x:

The block then determines the values of the states as follows:

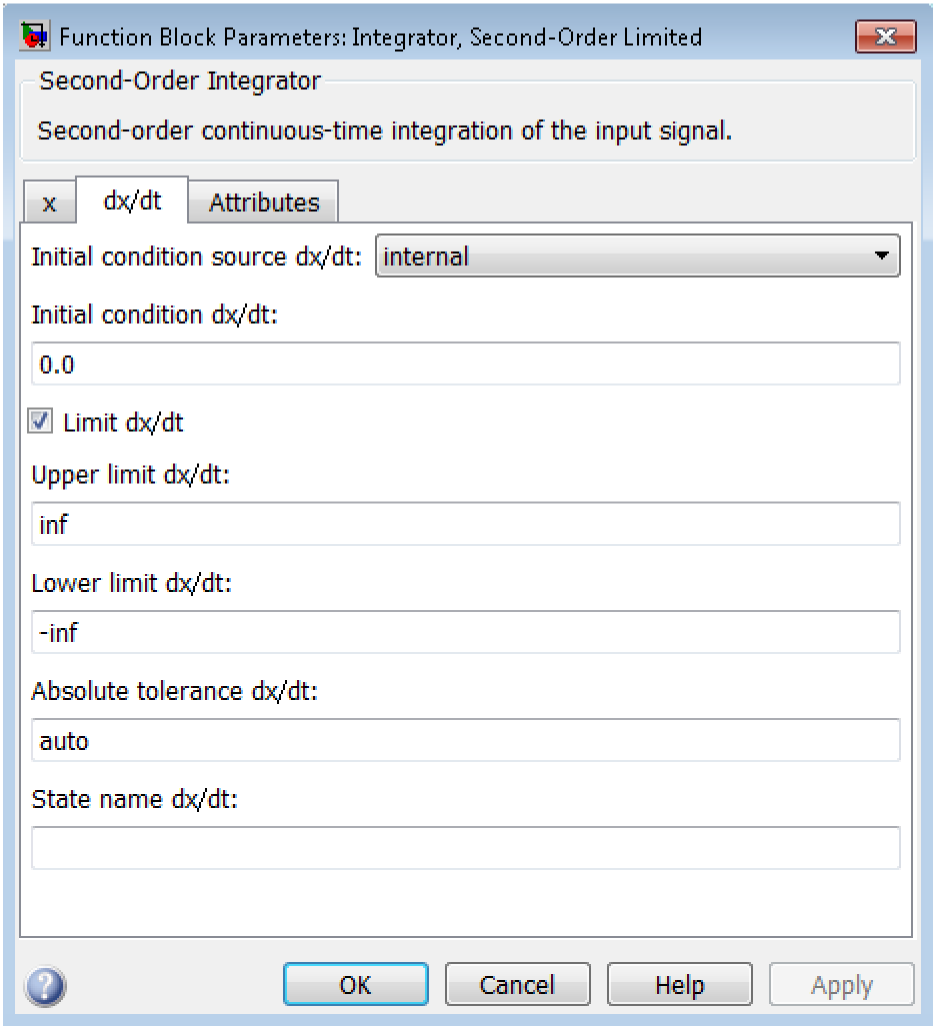
- When *x* is less than or equal to its lower limit, the value of *x* is held at its lower limit and *dx/dt* is set to zero.
- When *x* is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.

- 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 1-1658.

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 block 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*.



Function Block Parameters: Integrator, Second-Order Limited

Second-Order Integrator

Second-order continuous-time integration of the input signal.

Initial condition source dx/dt: internal

Initial condition dx/dt: 0.0

☑ Limit dx/dt

Upper limit dx/dt: inf

Lower limit dx/dt: -inf

Absolute tolerance dx/dt: auto

State name 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 it 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.

For such cases, the upper limit of *dx/dt* must be strictly positive and the lower limit of *dx/dt* must be strictly negative.

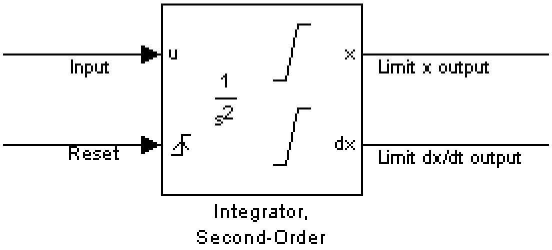
When both states are limited, the block determines the states as follows:

- 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 1-1658

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 box, 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 “Simulation of a Bouncing Ball”.

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

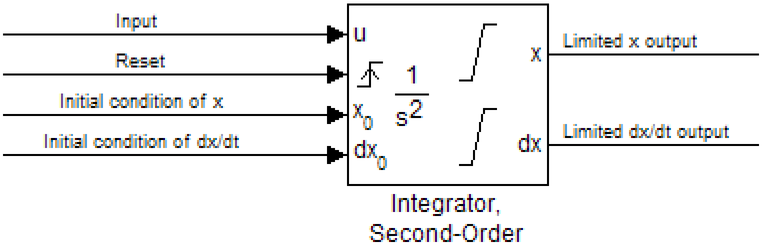
You can control whether to display the *x* or the *dx/dt* output port using the `ShowOutput` parameter. You can display one output port or both; however, you must select at least one.

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 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.



Ports

Input

u — Input signal u
scalar | vector | matrix

Input signal `u` to the integrator system, specified as a scalar, vector, or matrix.

Data Types: `double`

x0 — Initial condition x0
scalar | vector | matrix

External signal specifying the initial condition `x0` to the integrator system. You can specify the initial condition as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the **Initial condition source x** parameter to `external`.

Data Types: `double`

`dx0` — Initial condition `dx0`

scalar | vector | matrix

External signal specifying the initial condition `dx0` to the integrator system. You can specify the initial condition `dx0` as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the **Initial condition source dx/dt** parameter to `external`.

Data Types: `double`

Output

x — Output signal x

scalar | vector | matrix

`x` state output signal, provided as a scalar, vector, or matrix.

Data Types: `double`

dx — Output signal dx

scalar | vector | matrix

`dx` state output signal, specified as a scalar, vector, or matrix.

Data Types: `double`

Parameters

x

`Initial condition source x` — Source of initial condition for state *x*

`internal` (default) | `external`

Specify the source of the initial conditions for state *x*.

- `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 connected to the $\mathbf{X_0}$ input port.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

Selecting `internal` enables the **Initial condition x** parameter and removes the $\mathbf{X_0}$ input port.

Selecting `external` disables the **Initial condition x** parameter and enables the $\mathbf{X_0}$ input port.

Programmatic Use

Block Parameter: `ICSourceX`
Type: character vector, string
Values: `'internal'` | `'external'`
Default: `'internal'`

`Initial condition x` — Initial condition of state *x*

`0.0` (default) | scalar | vector | matrix

Specify the initial condition of state *x*.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

To enable this parameter, set **Initial condition source x** to `internal`.

Setting **Initial condition source x** to `external` disables this parameter and enables the $\mathbf{X_0}$ input port.

Programmatic Use

Block Parameter: `ICX`
Type: character vector, string

Values: scalar | vector | matrix
Default: `'0.0'`

`Limit x` — Limit values of state *x*

`off` | `on`

Limit state *x* of the block to a value between the **Lower limit x** and **Upper limit x** parameters. The default value of the Second-Order Integrator block is `off`. The default value of the Second-Order Integrator Limited is `on`.

- To limit state *x* to a value between the **Lower limit x** and **Upper limit x** parameters, select this check box.
- To remove range limitations on state *x*, clear this check box.

Dependencies

Selecting this check box enables the **Upper limit x** and **Lower limit x** parameters.

Programmatic Use

Block Parameter: `LimitX`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'` (Second-Order Integrator) | `'on'` (Second-Order Integrator Limited)

`Upper limit x` — Upper limit of state *x*

`1` | `inf` | scalar | vector | matrix

Specify the upper limit of state *x*. The default value for the Second-Order Integrator block is `inf`. The default value for the Second-Order Integrator Limited block is `1`.

Tips

The upper saturation limit for state *x* must be strictly greater than the lower saturation limit.

Dependencies

To enable this parameter, select the **Limit x** check box.

Programmatic Use

Block Parameter: `UpperLimitX`
Type: character vector, string
Values: `'1' | 'inf' |` scalar | vector | matrix

Default: `'1'` (Second-Order Integrator Limited) | `'inf'` (Second-Order Integrator)

`Lower limit x` — Lower limit of state *x*

`0` (default) | `-inf` | scalar | vector | matrix

Specify the lower limit of state *x*. The default value for the Second-Order Integrator block is `-inf`. The default value for the Second-Order Integrator Limited block is `0`.

Tip

The lower saturation limit for state *x* must be strictly less than the upper saturation limit.

Dependencies

To enable this parameter, select the **Limit x** check box.

Programmatic Use
Block Parameter: `LowerLimitX`
Type: character vector, string
Values: `'0' | '-inf' |` scalar | vector | matrix
Default: `'0'` (Second-Order Integrator Limited) | `'-inf'` (Second-Order Integrator)

`Wrap x` — Enable wrapping of x

`off` (default) | `on`

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.

If you specify **Wrapped upper value x** as `inf` and **Wrapped lower value x** as `-inf`, wrapping will never occur.

Dependencies

Selecting this check box enables **Wrapped upper value x** and **Wrapped lower value x**.

Programmatic Use
Block Parameter: `WrapX`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Wrapped upper value x` — Upper value for wrapping x

`pi` (default) | scalar | vector | matrix

Specify the upper value for wrapping *x*.

Dependencies

To enable this parameter, select the **Wrap x** check box.

Programmatic Use

Block Parameter: `WrappedUpperValueX`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'pi'`

`Wrapped lower value x` — Lower value for wrapping x

`-pi` (default) | scalar | vector | matrix

Specify the lower value for wrapping *x*.

Dependencies

To enable this parameter, select the **Wrap x** check box.

Programmatic Use

Block Parameter: `WrappedLowerValueX`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'-pi'`

`Absolute tolerance x` — Absolute tolerance for computing state *x*

`auto` (default) | `-1` | scalar | vector

Specify the absolute tolerance for computing state *x*.

- 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.

Programmatic Use

Block Parameter: `AbsoluteToleranceX`
Type: character vector, string
Values: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

`State name x` — Name for state *x*

`''` (default) | character vector | string

Assign a unique name to state *x*.

Tips

- 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, string or a cell array.

Programmatic Use

Block Parameter: `StateNameX`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

dx/dt

`Initial condition source dx/dt` — Source of initial condition for state *dx/dt*

`internal` (default) | `external`

Specify the source of initial conditions for state *dx/dt* as `internal` or `external`.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

- Selecting `internal` enables the **Initial condition dx/dt** parameter and removes the **$\mathbf{dx_0}$** input port.
- Selecting `external` disables the **Initial condition dx/dt** parameter and enables the **$\mathbf{dx_0}$** input port.

Programmatic Use

Block Parameter: `ICSourceDXDT`
Type: character vector
Values: `'internal'` | `'external'`
Default: `'internal'`

`Initial condition dx/dt` — Initial condition of state *dx/dt*

`0.0` (default) | scalar | vector | matrix

Specify the initial condition of state *dx/dt*.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

- Setting **Initial condition source dx/dt** to `internal` enables this parameter.
- Setting **Initial condition source dx/dt** to `external` disables this parameter.

Programmatic Use

Block Parameter: `ICDXDT`
Type: character vector
Values: scalar | vector | matrix
Default: `'0.0'`

`Limit dx/dt` — Limit values of state *dx/dt*

`off` | `on`

Limit the *dx/dt* state of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters. The default value of the Second-Order Integrator block is `off`. The default value of the Second-Order Integrator Limited is `on`.

Tip

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.

Dependencies

Selecting this check box enables the **Upper limit dx/dt** and **Lower limit dx/dt** parameters.

Programmatic Use

Parameter: `LimitDXDT`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'` (Second-Order Integrator) | `'on'` (Second-Order Integrator Limited)

`Upper limit dx/dt` — Upper limit of state *dx/dt*

`inf` (default) | scalar | vector | matrix

Specify the upper limit for state *dx/dt*.

Dependencies

If you limit *x*, then this parameter must have a strictly positive value.

To enable this parameter, select the **Limit dx/dt** check box.

Programmatic Use

Block Parameter: `UpperLimitDXDT`
Type: character vector
Values: scalar | vector | matrix
Default: `'inf'`

`Lower limit dx/dt` — Lower limit of state *dx/dt*

`-inf` (default) | scalar | vector | matrix

Specify the lower limit for state *dx/dt*.

Dependencies

If you limit *x*, then this parameter must have a strictly negative value.

To enable this parameter, select the **Limit dx/dt** check box.

Programmatic Use
Block Parameter: `LowerLimitDXDT`
Type: character vector
Values: scalar | vector | matrix
Default: `'-inf'`

`Absolute tolerance dx/dt` — Absolute tolerance for computing state *dx/dt*

`auto` (default) | `-1` | scalar | vector

Specify the absolute tolerance for computing state *dx/dt*.

- 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.

Programmatic Use
Block Parameter: `AbsoluteToleranceDXDT`
Type: character vector, string, scalar, or vector
Values: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

`State name dx/dt` — Name for state *dx/dt*

`' '` (default) | character vector | string

Assign a unique name to state *dx/dt*.

Tips

- 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, string, or a cell array.

Programmatic Use
Block Parameter: `StateNameDXDT`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

Attributes

`External reset` — Reset states to their initial conditions

`none` (default) | `rising` | `falling` | `either`

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

- `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.

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector, string
Values: `'none'` | `'rising'` | `'falling'` | `'either'`
Default: `'none'`

`Enable zero-crossing detection` — Enable zero-crossing detection

on (default) | `off`

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

Programmatic Use
Parameter:`ZeroCross`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

`Reinitialize dx/dt when x reaches saturation` — Reset *dx/dt* when *x* reaches saturation

`off` (default) | `on`

At the instant when state *x* reaches saturation, reset *dx/dt* to its current initial conditions.

Tip

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.

Programmatic Use
Block Parameter: `ReinitDXDTwhenXreachesSaturation`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Ignore state limits and the reset for linearization` — Ignore state limits and external reset for linearization

`off` (default) | `on`

For linearization purposes, have Simulink ignore the specified state limits and the external reset.

Programmatic Use
Block Parameter: `IgnoreStateLimitsAndResetForLinearization`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Show output` — Output ports to display

`both` (default) | `x` | `dxdt`

Specify the output ports on the block.

- `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.

Programmatic Use
Block Parameter: `ShowOutput`
Type: character vector, string
Values: `'both'` | `'x'` | `'dxdt'`
Default: `'both'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select `Analysis > Control Design > Model Discretizer`. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Integrator | Integrator Limited | Second-Order Integrator Limited

Topics

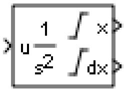
"Zero-Crossing Detection"
"Error Tolerances for Variable-Step Solvers"
"Algebraic Loops"

Introduced in R2010a

Second-Order Integrator Limited

Integrate input signal twice

Library: Simulink / Continuous



Description

The Second-Order Integrator block and the Second-Order Integrator Limited block solve the second-order initial value problem:

$$\frac{d^2x}{dt^2} = u,$$
$$\left.\frac{dx}{dt}\right|_{t=0} = dx_o,$$
$$\left.x\right|_{t=0} = x_o,$$

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*. 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 1-1675.

Simulink software can use several 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 “Solver Types”.) 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:

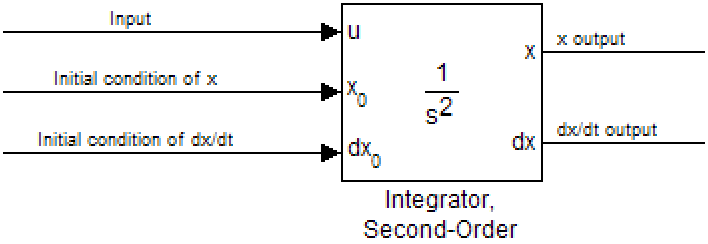
- Specify whether the source of each state initial condition is internal or external
- Specify a value for the state initial conditions
- Define upper and lower limits on either or both states
- Specify absolute tolerances for each state
- Specify names for both states
- Choose an external reset condition
- 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.



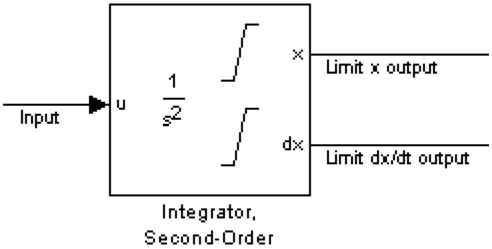
Note

- 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 1-1675) 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.

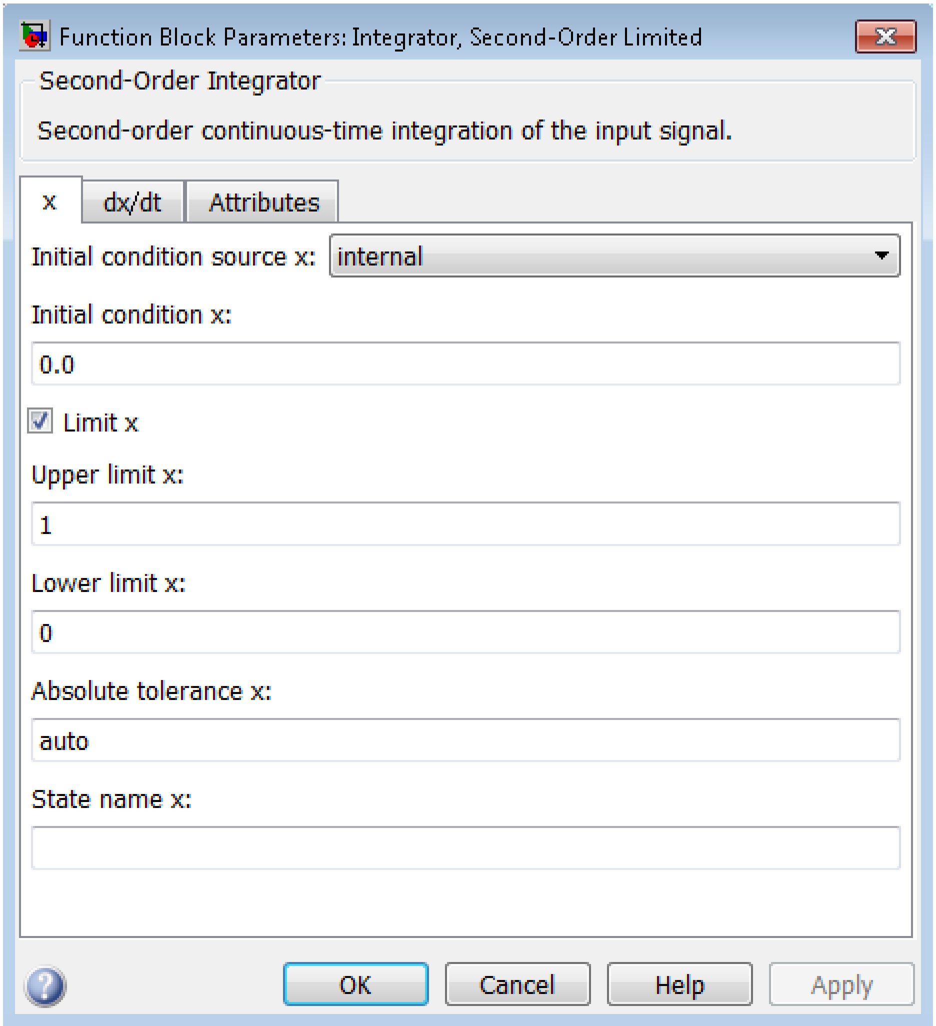
The block appearance changes when you limit one or both states. With both states limited, the block appears as follows.



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.



Second-Order Integrator

Second-order continuous-time integration of the input signal.

Initial condition source x: internal

Initial condition x: 0.0

☑ Limit x

Upper limit x: 1

Lower limit x: 0

Absolute tolerance x: auto

State name x:

The block then determines the values of the states as follows:

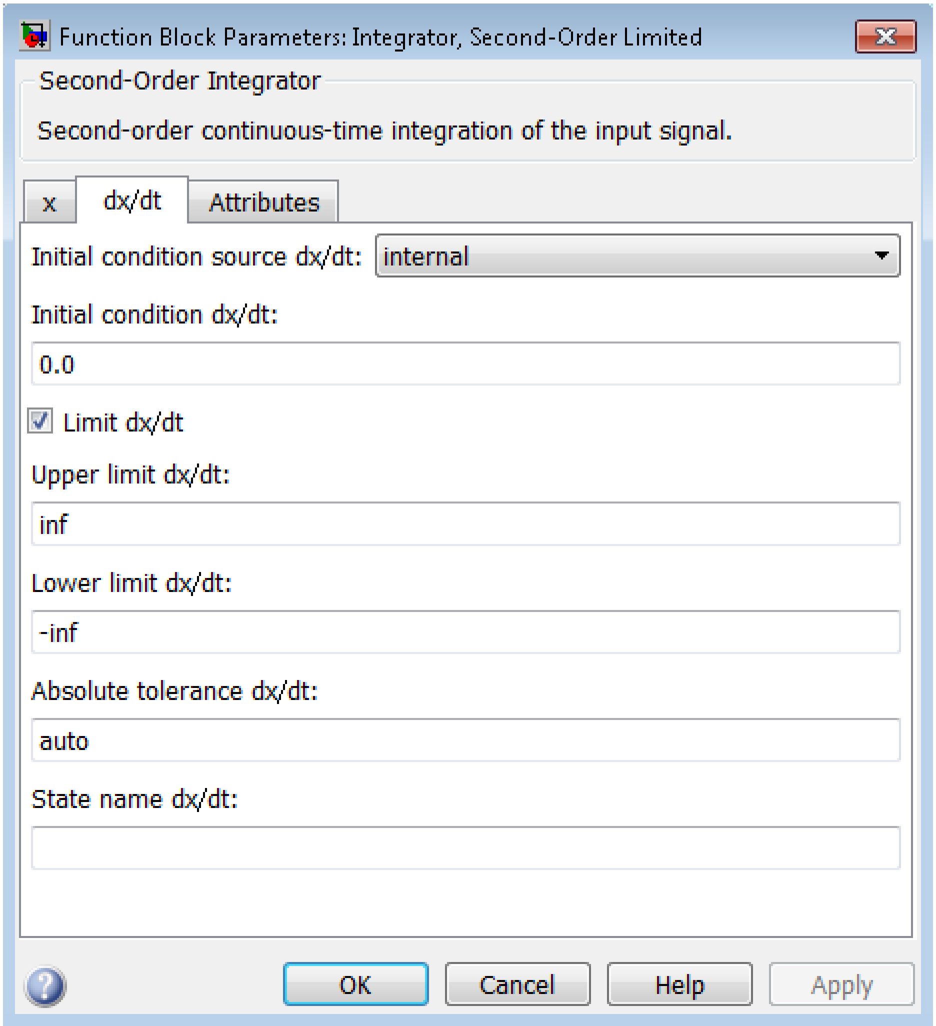
- When *x* is less than or equal to its lower limit, the value of *x* is held at its lower limit and *dx/dt* is set to zero.
- When *x* is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.

- 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 1-1682.

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 block 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*.



Function Block Parameters: Integrator, Second-Order Limited

Second-Order Integrator

Second-order continuous-time integration of the input signal.

Initial condition source dx/dt: internal

Initial condition dx/dt: 0.0

☑ Limit dx/dt

Upper limit dx/dt: inf

Lower limit dx/dt: -inf

Absolute tolerance dx/dt: auto

State name 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 it 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.

For such cases, the upper limit of *dx/dt* must be strictly positive and the lower limit of *dx/dt* must be strictly negative.

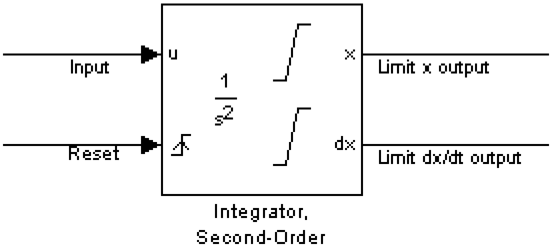
When both states are limited, the block determines the states as follows:

- 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 1-1682

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 box, 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 “Simulation of a Bouncing Ball”.

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

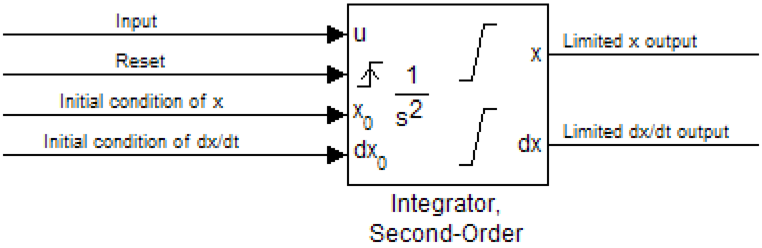
You can control whether to display the *x* or the *dx/dt* output port using the `ShowOutput` parameter. You can display one output port or both; however, you must select at least one.

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 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.



Ports

Input

u — Input signal `u`
scalar | vector | matrix

Input signal `u` to the integrator system, specified as a scalar, vector, or matrix.

Data Types: `double`

x0 — Initial condition `x0`
scalar | vector | matrix

External signal specifying the initial condition `x0` to the integrator system. You can specify the initial condition as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the **Initial condition source x** parameter to `external`.

Data Types: `double`

`dx0` — Initial condition `dx0`

scalar | vector | matrix

External signal specifying the initial condition `dx0` to the integrator system. You can specify the initial condition `dx0` as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the **Initial condition source dx/dt** parameter to `external`.

Data Types: `double`

Output

x — Output signal x

scalar | vector | matrix

`x` state output signal, provided as a scalar, vector, or matrix.

Data Types: `double`

dx — Output signal dx

scalar | vector | matrix

`dx` state output signal, specified as a scalar, vector, or matrix.

Data Types: `double`

Parameters

x

`Initial condition source x` — Source of initial condition for state *x*

`internal` (default) | `external`

Specify the source of the initial conditions for state *x*.

- `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 connected to the $\mathbf{X_0}$ input port.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

Selecting `internal` enables the **Initial condition x** parameter and removes the $\mathbf{X_0}$ input port.

Selecting `external` disables the **Initial condition x** parameter and enables the $\mathbf{X_0}$ input port.

Programmatic Use

Block Parameter: `ICSourceX`
Type: character vector, string
Values: `'internal'` | `'external'`
Default: `'internal'`

`Initial condition x` — Initial condition of state *x*

`0.0` (default) | scalar | vector | matrix

Specify the initial condition of state *x*.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

To enable this parameter, set **Initial condition source x** to `internal`.

Setting **Initial condition source x** to `external` disables this parameter and enables the $\mathbf{X_0}$ input port.

Programmatic Use

Block Parameter: `ICX`
Type: character vector, string

Values: scalar | vector | matrix
Default: `'0.0'`

`Limit x` — Limit values of state *x*

`off` | `on`

Limit state *x* of the block to a value between the **Lower limit x** and **Upper limit x** parameters. The default value of the Second-Order Integrator block is `off`. The default value of the Second-Order Integrator Limited is `on`.

- To limit state *x* to a value between the **Lower limit x** and **Upper limit x** parameters, select this check box.
- To remove range limitations on state *x*, clear this check box.

Dependencies

Selecting this check box enables the **Upper limit x** and **Lower limit x** parameters.

Programmatic Use

Block Parameter: `LimitX`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'` (Second-Order Integrator) | `'on'` (Second-Order Integrator Limited)

`Upper limit x` — Upper limit of state *x*

`1` | `inf` | scalar | vector | matrix

Specify the upper limit of state *x*. The default value for the Second-Order Integrator block is `inf`. The default value for the Second-Order Integrator Limited block is `1`.

Tips

The upper saturation limit for state *x* must be strictly greater than the lower saturation limit.

Dependencies

To enable this parameter, select the **Limit x** check box.

Programmatic Use

Block Parameter: `UpperLimitX`
Type: character vector, string
Values: `'1' | 'inf' |` scalar | vector | matrix

Default: `'1'` (Second-Order Integrator Limited) | `'inf'` (Second-Order Integrator)

`Lower limit x` — Lower limit of state *x*

`0` (default) | `-inf` | scalar | vector | matrix

Specify the lower limit of state *x*. The default value for the Second-Order Integrator block is `-inf`. The default value for the Second-Order Integrator Limited block is `0`.

Tip

The lower saturation limit for state *x* must be strictly less than the upper saturation limit.

Dependencies

To enable this parameter, select the **Limit x** check box.

Programmatic Use
Block Parameter: `LowerLimitX`
Type: character vector, string
Values: `'0' | '-inf' |` scalar | vector | matrix
Default: `'0'` (Second-Order Integrator Limited) | `'-inf'` (Second-Order Integrator)

`Wrap x` — Enable wrapping of x

`off` (default) | `on`

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.

If you specify **Wrapped upper value x** as `inf` and **Wrapped lower value x** as `-inf`, wrapping will never occur.

Dependencies

Selecting this check box enables **Wrapped upper value x** and **Wrapped lower value x**.

Programmatic Use
Block Parameter: `WrapX`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Wrapped upper value x` — Upper value for wrapping x

`pi` (default) | scalar | vector | matrix

Specify the upper value for wrapping *x*.

Dependencies

To enable this parameter, select the **Wrap x** check box.

Programmatic Use

Block Parameter: `WrappedUpperValueX`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'pi'`

`Wrapped lower value x` — Lower value for wrapping x

`-pi` (default) | scalar | vector | matrix

Specify the lower value for wrapping *x*.

Dependencies

To enable this parameter, select the **Wrap x** check box.

Programmatic Use

Block Parameter: `WrappedLowerValueX`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'-pi'`

`Absolute tolerance x` — Absolute tolerance for computing state *x*

`auto` (default) | `-1` | scalar | vector

Specify the absolute tolerance for computing state *x*.

- 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.

Programmatic Use
Block Parameter: `AbsoluteToleranceX`
Type: character vector, string
Values: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

`State name x` — Name for state *x*

`''` (default) | character vector | string

Assign a unique name to state *x*.

Tips

- 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, string or a cell array.

Programmatic Use
Block Parameter: `StateNameX`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

dx/dt

`Initial condition source dx/dt` — Source of initial condition for state *dx/dt*

`internal` (default) | `external`

Specify the source of initial conditions for state *dx/dt* as `internal` or `external`.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

- Selecting `internal` enables the **Initial condition dx/dt** parameter and removes the **$\mathbf{dx_0}$** input port.
- Selecting `external` disables the **Initial condition dx/dt** parameter and enables the **$\mathbf{dx_0}$** input port.

Programmatic Use

Block Parameter: `ICSourceDXDT`
Type: character vector
Values: `'internal'` | `'external'`
Default: `'internal'`

`Initial condition dx/dt` — Initial condition of state *dx/dt*

`0.0` (default) | scalar | vector | matrix

Specify the initial condition of state *dx/dt*.

Limitations

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

- Setting **Initial condition source dx/dt** to `internal` enables this parameter.
- Setting **Initial condition source dx/dt** to `external` disables this parameter.

Programmatic Use

Block Parameter: `ICDXDT`
Type: character vector
Values: scalar | vector | matrix
Default: `'0.0'`

`Limit dx/dt` — Limit values of state *dx/dt*

`off` | `on`

Limit the *dx/dt* state of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters. The default value of the Second-Order Integrator block is `off`. The default value of the Second-Order Integrator Limited is `on`.

Tip

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.

Dependencies

Selecting this check box enables the **Upper limit dx/dt** and **Lower limit dx/dt** parameters.

Programmatic Use

Parameter: `LimitDXDT`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'` (Second-Order Integrator) | `'on'` (Second-Order Integrator Limited)

`Upper limit dx/dt` — Upper limit of state *dx/dt*

`inf` (default) | scalar | vector | matrix

Specify the upper limit for state *dx/dt*.

Dependencies

If you limit *x*, then this parameter must have a strictly positive value.

To enable this parameter, select the **Limit dx/dt** check box.

Programmatic Use

Block Parameter: `UpperLimitDXDT`
Type: character vector
Values: scalar | vector | matrix
Default: `'inf'`

`Lower limit dx/dt` — Lower limit of state *dx/dt*

`-inf` (default) | scalar | vector | matrix

Specify the lower limit for state *dx/dt*.

Dependencies

If you limit *x*, then this parameter must have a strictly negative value.

To enable this parameter, select the **Limit dx/dt** check box.

Programmatic Use

Block Parameter: `LowerLimitDXDT`
Type: character vector
Values: scalar | vector | matrix
Default: `'-inf'`

`Absolute tolerance dx/dt` — Absolute tolerance for computing state *dx/dt*

`auto` (default) | `-1` | scalar | vector

Specify the absolute tolerance for computing state *dx/dt*.

- 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.

Programmatic Use

Block Parameter: `AbsoluteToleranceDXDT`
Type: character vector, string, scalar, or vector
Values: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

`State name dx/dt` — Name for state *dx/dt*

`' '` (default) | character vector | string

Assign a unique name to state *dx/dt*.

Tips

- 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, string, or a cell array.

Programmatic Use
Block Parameter: `StateNameDXDT`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

Attributes

`External reset` — Reset states to their initial conditions

`none` (default) | `rising` | `falling` | `either`

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

- `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.

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector, string
Values: `'none'` | `'rising'` | `'falling'` | `'either'`
Default: `'none'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | `off`

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

Programmatic Use
Parameter:`ZeroCross`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

`Reinitialize dx/dt when x reaches saturation` — Reset *dx/dt* when *x* reaches saturation

`off` (default) | `on`

At the instant when state *x* reaches saturation, reset *dx/dt* to its current initial conditions.

Tip

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.

Programmatic Use
Block Parameter: `ReinitDXDTwhenXreachesSaturation`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Ignore state limits and the reset for linearization` — Ignore state limits and external reset for linearization

`off` (default) | `on`

For linearization purposes, have Simulink ignore the specified state limits and the external reset.

Programmatic Use
Block Parameter: `IgnoreStateLimitsAndResetForLinearization`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Show output` — Output ports to display

`both` (default) | `x` | `dxdt`

Specify the output ports on the block.

- `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.

Programmatic Use
Block Parameter: `ShowOutput`
Type: character vector, string
Values: `'both'` | `'x'` | `'dxdt'`
Default: `'both'`

Block Characteristics

Data Types	`double`
Multidimensional Signals	`No`
Variable-Size Signals	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select `Analysis > Control Design > Model Discretizer`. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Integrator | Integrator Limited | Second-Order Integrator

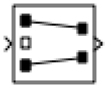
Topics

“Zero-Crossing Detection”
“Error Tolerances for Variable-Step Solvers”
“Algebraic Loops”

Introduced in R2010a

Selector

Select input elements from vector, matrix, or multidimensional signal
Library: Simulink / Signal Routing



Description

The Selector block generates as output selected or reordered elements of an input vector, matrix, or multidimensional signal.

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. If you define dimensions as shown in the next table, the output is `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.

Row	Index Option	Index	Output Size
1	`Select all`		
2	`Starting index (dialog)`	2	5
3	`Index vector (dialog)`	[1 3 5]	

Row	Index Option	Index	Output Size
4	`Starting index (port)`		8
5	`Index vector (port)`		
6	`Starting and ending indices (port)`		

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".

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | multidimensional

Input signal and source of elements to output signal.

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

`IndxN` — Nth index signal

scalar | vector | matrix

External port specifying an index for the selection of the corresponding output element.

Dependencies

To enable an external index port, in the corresponding row of the **Index Option** table, set **Index Option** to `Index vector (port)`, `Starting index (port)`, or `Starting and ending indices (port)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `enumerated`

Output

`Port_1` — Output signal

scalar | vector | matrix | multidimensional

Output signal generated from selected or reordered elements of input signal.

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

Parameters

`Number of input dimensions` — Number of dimensions of the input signal

`1` (default) | integer

Specifies the number of dimensions of the output signal.

Programmatic Use
Block Parameter: `NumberOfDimensions`
Type: character vector
Values: `integer`
Default: `'1'`

`Index mode` — Index mode

`One-based` (default) | `Zero-based`

Specifies the indexing mode. If `One-based` is selected, an index of 1 specifies the first element of the input vector. If `Zero-based` is selected, an index of 0 specifies the first element of the input vector.

Programmatic Use
Block Parameter: `IndexMode`
Type: character vector
Values: `'One-based' | 'Zero-based'`
Default: `'One-based'`

`Index Option` — Index method for elements

`Index vector (dialog)` (default) | `Select all` | `Index vector (port)` | `Starting index (dialog)` | `Starting index (port)` | `Starting and ending indices (port)`

Defines, 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.

Programmatic Use
Block Parameter: `IndexOptionArray`
Type: character vector
Values: `'Select all'`|`'Index vector (dialog)'`|`'Index option (port)'`|`'Starting index (dialog)'`|`'Starting index (port)'`|`Starting and ending indices (port)`
Default: `'Index vector (dialog)'`

Index — Index of elements

1 (default) | integer

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.

Programmatic Use
Block Parameter: `IndexParamArray`
Type: character vector
Values: cell array
Default: `'{ }'`

`Output Size` — Width of the block output signal

1 (default) | integer

Specifies the width of the block output signal.

Programmatic Use
Block Parameter: `OutputSizeArray`
Type: character vector
Values: cell array
Default: `'{ }'`

`Input port size` — Width of the input signal

3 (default) | integer

Specify the width of the block input signal for 1-D signals. Enter `-1` to inherit from the driving block.

Programmatic Use
Block Parameter: `InputPortWidth`
Type: character vector
Values: `integer`
Default: `'1'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For more information on HDL code generation, see Selector.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

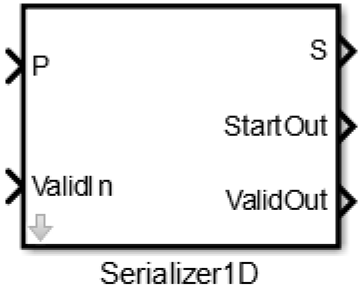
See Also

Assignment | Bus Selector | Switch

Introduced before R2006a

Serializer1D

Convert vector signal to scalar or smaller vectors



Library

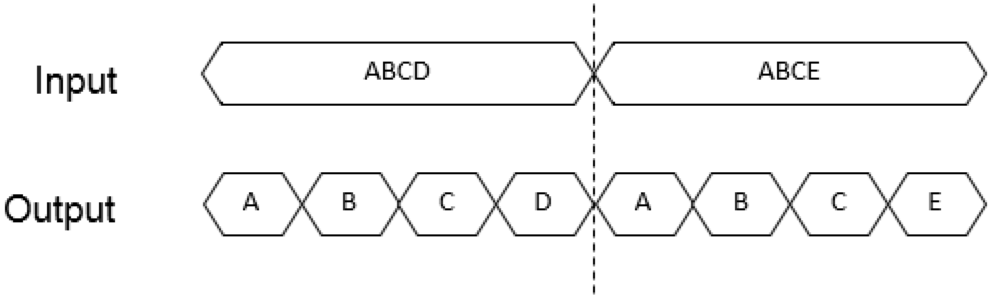
HDL Coder / HDL Operations

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:

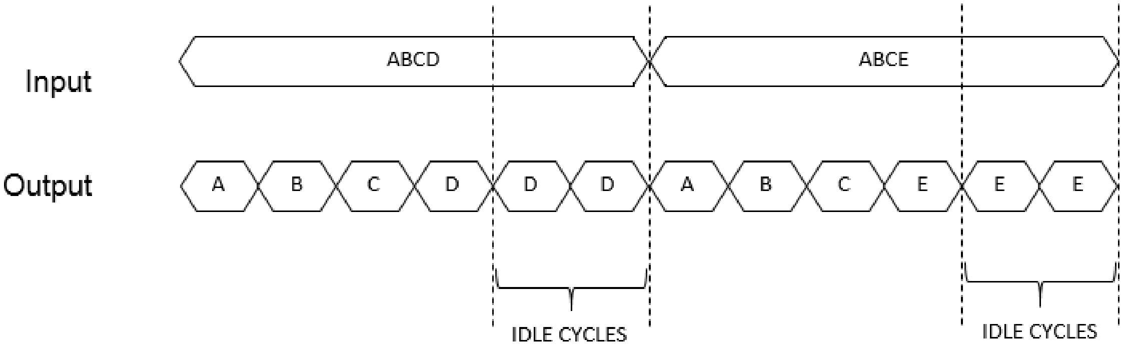
$$Output\ Sample\ Time = Input\ Sample\ Time\,/\,(Ratio + Idle\ Cycles)$$

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.

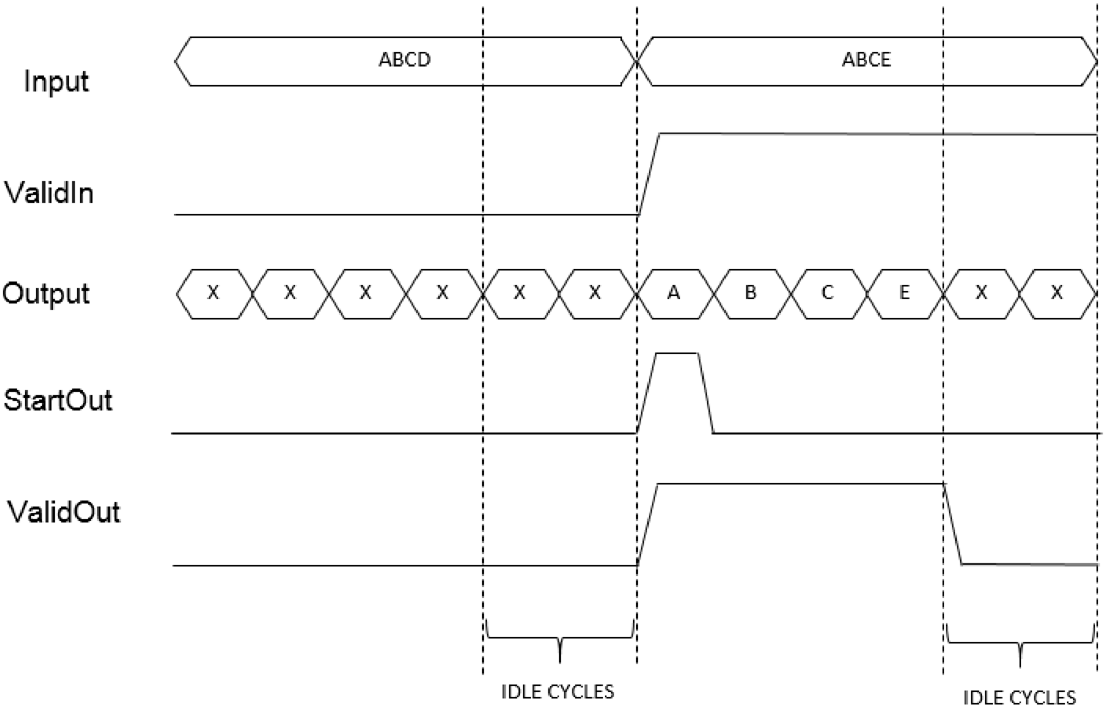


For each slow vector signal, the output has six fast cycles consisting of the four serialized scalar signals and two idle cycles. The sample time at the output is

$Output\ Sample\ Time = Input\ Sample\ Time\,/\,6$.

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: Simulink / 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-2 MATLAB or a Level-1 or Level-2 C MEX S-function (see “MATLAB 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 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.

Parameters

`S-function name` — Name of the S-function

character array (default)

Use this parameter to specify the name of your S-function.

Programmatic Use

Block Parameter: `FunctionName`
Type: character vector
Value: name of the S-function
Default: `'system'`

`S-function parameters` — Additional S-function parameters

cell array (default)

Specify the additional S-function parameters.

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.

Programmatic Use
Block Parameter: `Parameters`
Type: character vector
Value: S-function parameters
Default: `' '`

`S-function modules` — List additional files for code generation

cell array (default)

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).

Programmatic Use
Block Parameter: `SFunctionModules`
Type: character vector
Value: character vector of file names
Default: `' '`

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[ba] | `bus`[a] | `string`[a]

Direct Feedthrough	`Yes`[a]
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`Yes`[a]
Zero-Crossing Detection	`Yes`[a]

[a] Actual data type or capability support depends on block implementation.

[b] See Writing Fixed-Point S-Functions for details on using fixed-point data types in S-functions.

Extended Capabilities

C/C++ Code Generation

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

- Actual code generation support depends on block implementation.
- S-functions that call into MATLAB are not supported for code generation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

- Actual data type support depends on block implementation.
- See “Integrate External Code” (Fixed-Point Designer) for details on using fixed-point data types in S-functions.

See Also

Level-2 MATLAB S-Function | S-Function Builder

Topics

“Implementing S-Functions”
“Create a Basic C MEX S-Function”

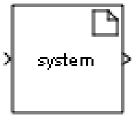
"Write Level-2 MATLAB S-Functions"

Introduced before R2006a

S-Function Builder

Integrate C or C++ code to create S-functions

Library: Simulink / User-Defined Functions



Description

The S-function builder integrates new or existing C or C++ code and creates a C MEX S-function from specifications 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 in order to call your C or C++ code in the instance's `mdlStart`, `mdlOutputs`, `mdlDerivatives`, `mdlUpdate` and `mdlTerminate` methods. To learn how Simulink engine interacts with S-functions, see “Simulink Engine Interaction with C S-Functions”.

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”.

Ports

Input

In — Input to an S-function builder

scalar | vector | matrix

The S-Function Builder can accept and 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.

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

Output

`Out` — Output from an S-function builder
scalar | vector | matrix

The S-Function Builder can 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.

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

Parameters

See “S-Function Builder Dialog Box” in the online documentation for information on using the S-Function Builder block's parameter dialog box.

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`

Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

S-functions that call into MATLAB are not supported for code generation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Level-2 MATLAB S-Function | S-Function

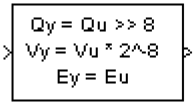
Topics

“Build S-Functions Automatically”
“S-Function Builder Dialog Box”

Introduced before R2006a

Shift Arithmetic

Shift bits or binary point of signal



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 `sfix(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 `sfix(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`	• Floating point • Built-in integer • Fixed point
`s`	• 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	• Sign of `u` • Word length of `u` • Slope of `u * 2^(max(binary points to shift))` • Bias of `u * 2^(max(binary points to shift – bits to shift))` , for bit shifts where the direction is bidirectional or right • Bias of `u * 2^(max(binary points to shift + 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	• Built-in integer • Fixed-point integer
Binary points to shift	• 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 1-1720.

Check for out-of-range 'Bits to shift' in generated code

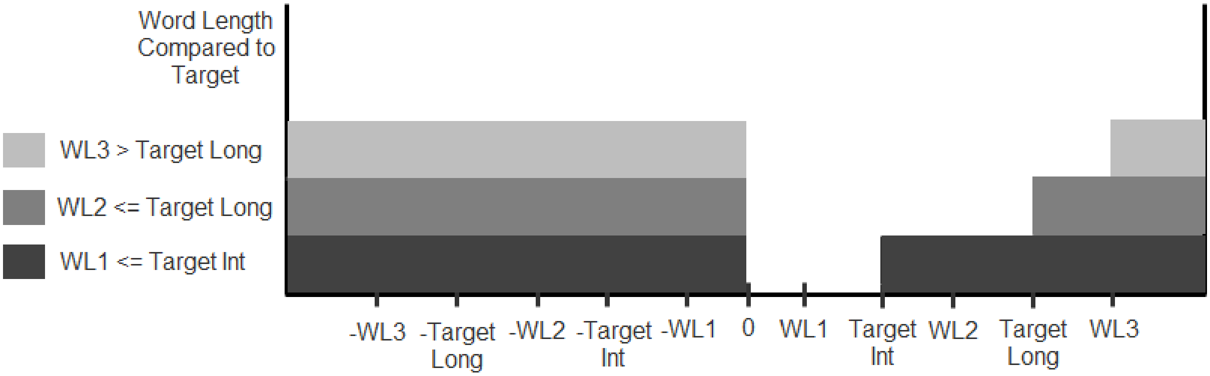
Select this check box to include conditional statements in the generated code that protect against out-of-range bit shift values. This check box is available when **Bits to shift: Source** is `Input port`.

For more information, see "Code Generation for Out-of-Range Bit Shift Values" on page 1-1722.

Out-of-Range Bit Shifts

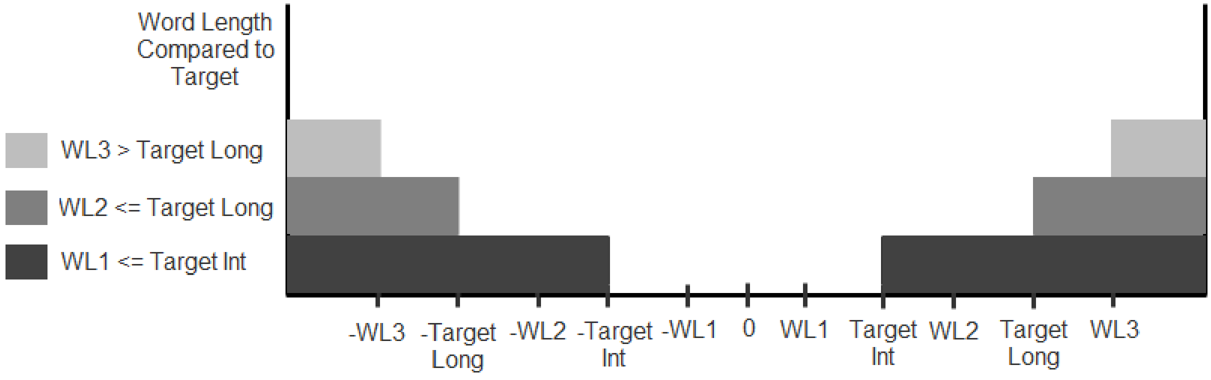
Definition of an Out-of-Range Bit Shift

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.



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.



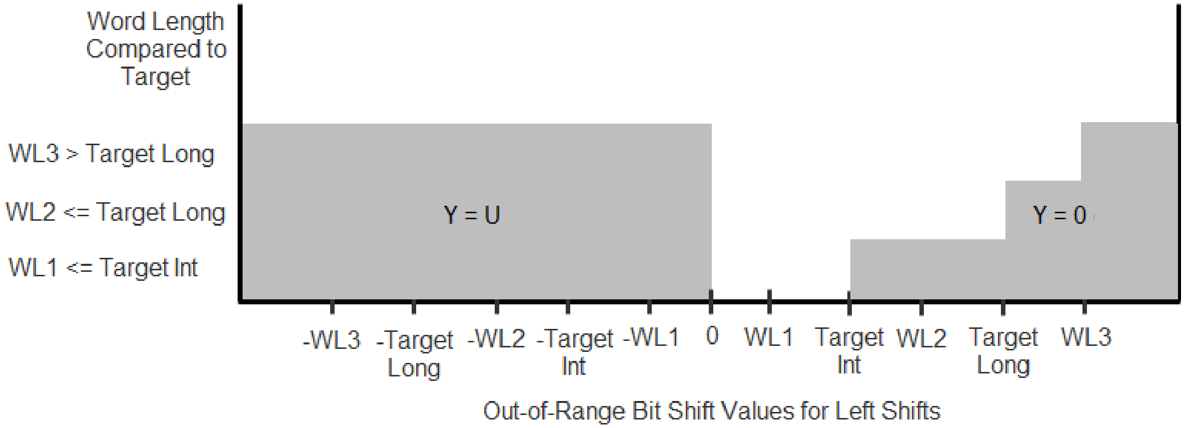
Out-of-Range Bit Shift Values for Bidirectional Shifts

The diagnostic for out-of-range bit shifts responds as follows, depending on the mode of operation:

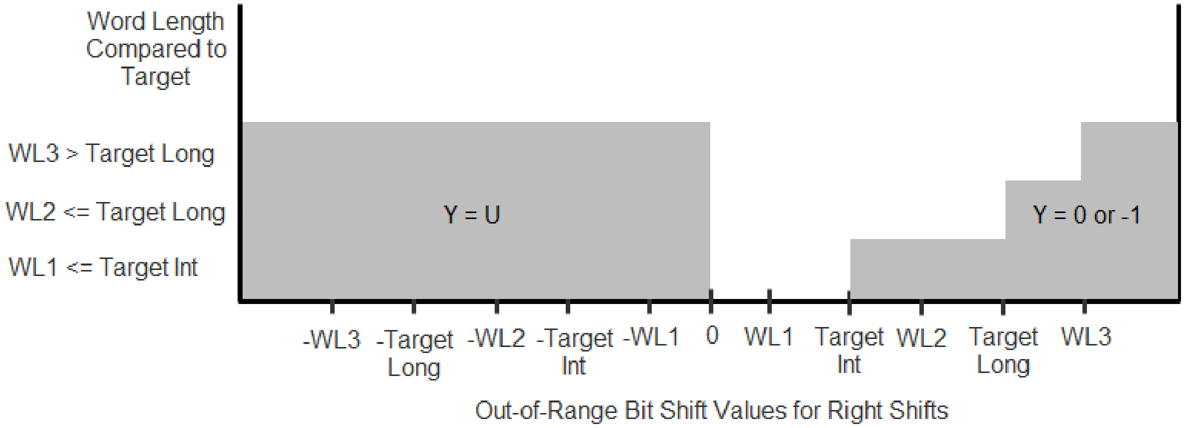
Mode	Diagnostic for out-of-range shift value		
	None	Warning	Error
Simulation	Do not report any warning or error.	Report a warning but continue simulation.	Report an error and stop simulation.
Accelerator modes and code generation	Has no effect.		

Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values

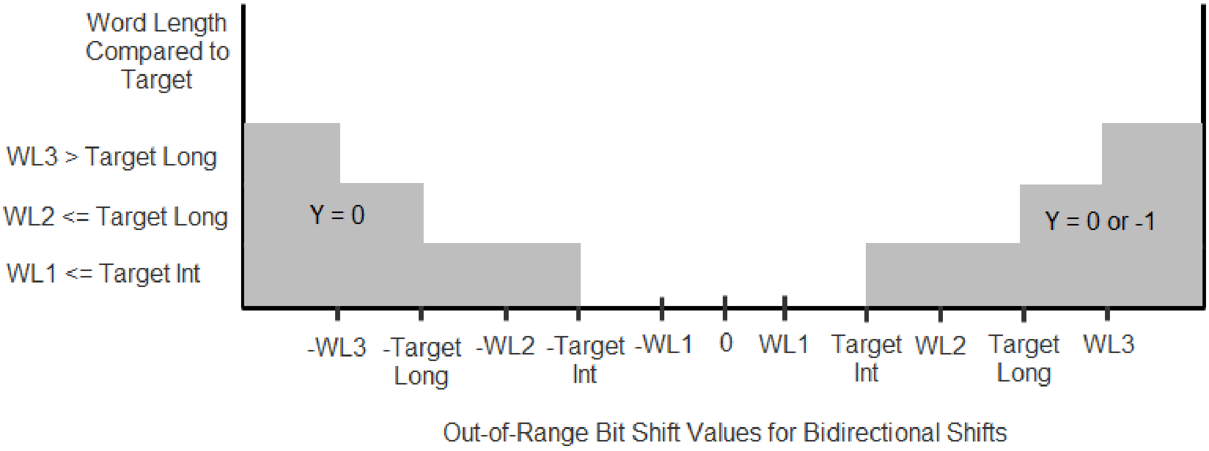
Suppose that `U` is the input, `WL` is the input word length, and `Y` is the output. The output for an out-of-range bit shift value for left shifts is as follows:



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**.

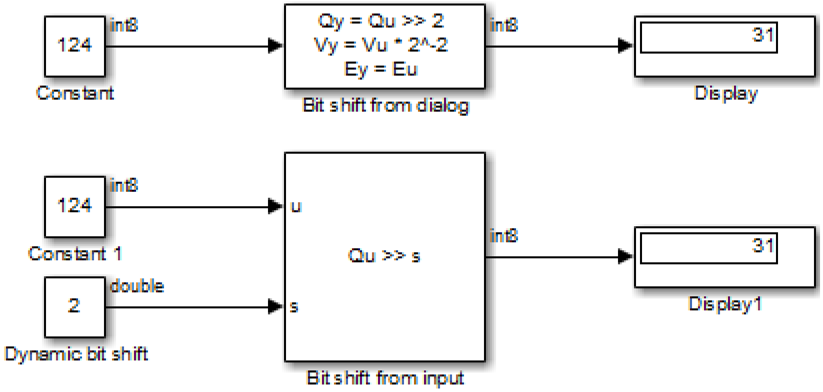
Check Box Setting	Generated Code	Simulation Results Compared to Generated Code
Selected	Includes conditional statements to protect against out-of-range bit shift values.	Simulation and Accelerator mode results match those of code generation.
Cleared	Does not protect against out-of-range bit shift values.	• 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`

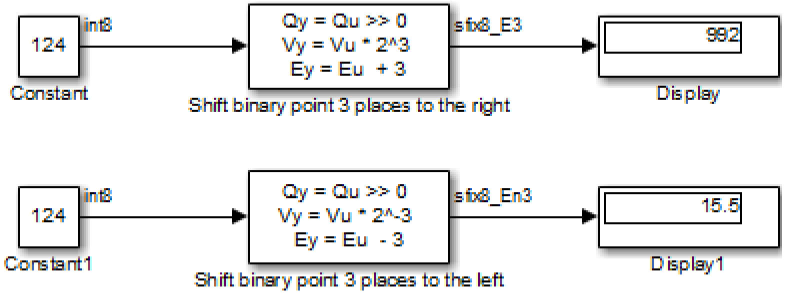
Block	Parameter	Setting
Bit shift from input	**Bits to shift: Source**	Input port
	Bits to shift: Direction	Right

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`
`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

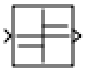
Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Sign

Indicate sign of input

Library: Simulink / Math Operations



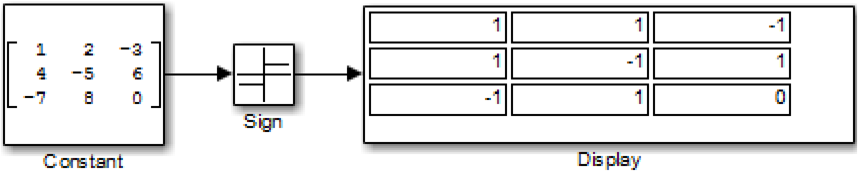
Description

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:

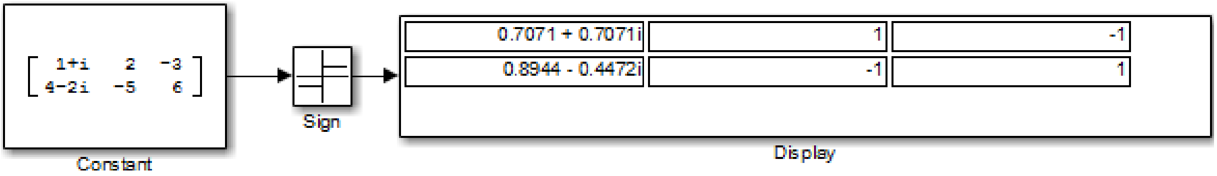


Block Behavior for Complex Inputs

When the input u is a complex scalar, the block output matches the MATLAB result for:

```
sign(u)            =            u./            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:



Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal whose sign will determine the output.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal that is the sign of the input signal.

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

Parameters

`Enable zero-crossing detection` — Enable zero-crossing detection

on (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

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™.

This block has a single, default HDL architecture.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Abs

Introduced before R2006a

Signal Builder

Create and generate interchangeable groups of signals whose waveforms are piecewise linear

Library: Simulink / Sources

Description

The Signal Builder block allows you to create interchangeable groups of piecewise linear signal sources and use them in a model. You can quickly switch the signal groups into and out of a model to facilitate testing. In the Signal Builder window, create signals and define the output waveforms.To open the window, double-click the block. See “Signal Groups”.

Note Use the `signalbuilder` function to create and access Signal Builder blocks programmatically.

Ports

Output

`Signal 1` — First output signal
scalar | vector | matrix

First output signal from the signal group currently visible in the Signal Builder window.

Data Types: `double` | `bus`

`Signal n` — nth output signal
scalar | vector | matrix

`n`th output signal from the signal group currently visible in the Signal Builder window. `n` corresponds to the signal index.

Data Types: `double` | `bus`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

See Also

Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

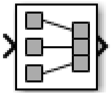
Topics

"Scenarios"
"Signal Basics"

Introduced before R2006a

Signal Conversion

Convert signal to new type without altering signal values
Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal to convert
scalar | vector | matrix | N-D array

Input signal to convert, specified as a scalar, vector, matrix, or N-D array.

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

Output

`Port_1` — Converted signal
scalar | vector | matrix | N-D array

Output signal is the input signal converted to the specified type.

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

Parameters

`Output` — Type of conversion

`Signal copy` (default) | `Virtual bus` | `Nonvirtual bus`

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`

- 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 discontiguous 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 discontiguous areas (represented by the block input) to the contiguous areas (represented by the block 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.

- The `Virtual bus` option converts a nonvirtual bus to a virtual bus.
- The `Nonvirtual bus` option converts a virtual bus to a nonvirtual bus.

Programmatic Use

Block Parameter: `ConversionOutput`
Type: character vector
Values: `'Signal copy' | 'Virtual bus' | 'Nonvirtual bus'`
Default: `'Signal copy'`

`Data type` — Nonvirtual bus data type

`Inherit: auto` (default) | `Bus: <object name>` | `<data type expression>`

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.

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.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Dependencies

To enable this parameter, set **Output** to `Nonvirtual bus`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: auto' | 'Bus: <object name>' | '<data type expression>'`
Default: `'Inherit: auto'`

`Exclude this block from 'Block reduction' optimization` — Exclude from block reduction optimization

`off` (default) | `on`

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”.

Programmatic Use
Block Parameter: `OverrideOpt`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `string`
Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Signal Conversion.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Bus Creator | Data Type Conversion

Topics

“Buses”

Introduced before R2006a

Signal Editor

Display, create, edit, and switch interchangeable scenarios

Library: Simulink / Sources

Scenario
Signal 1

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.

You can port Signal Builder block configurations to the Signal Editor block using the `signalBuilderToSignalEditor` function. Internal storage format and preprocessing of data differs between the Signal Builder and Signal Editor blocks. When using the variable step solver, this difference causes different simulation time steps and mismatched output between the two blocks. The difference between the outputs of both blocks can be minimized by reducing the value of **Max step size** of the variable step solver. Another option is to insert more data points in the input signal of Signal Editor to better represent its shape. This can be done using the Signal Editor user interface. To better match the output from both blocks, use the fixed-step solver or set the sample time for both blocks to the same discrete sample time (greater than 0). For more information on discrete sample times, see “Discrete Sample Time”.

To programmatically get the total number of scenarios and signals in the block, use the `get_param` `NumberOfScenarios` and `NumberOfSignals` properties. These properties contain these values as character vectors. To convert these values to doubles, use the `str2double` function.

Limitations

The Signal Editor block does not support:

- Function-calls
- Array of buses
- Buses while using rapid accelerator mode
- `timetable` objects
- Ground signals

The Signal Editor block supports dynamic strings. It does not support strings with maximum length. In addition, strings in the Signal Editor block cannot output:

- Non-scalar MATLAB strings.
- String data that contains missing values.
- String data that contains non-ASCII characters.

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` | `string` | `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. When using the block on multiple platforms, consider specifying just the MAT-file name and having the MAT-file be on the MATLAB path.

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 create and edit scenario MAT-files.

Programmatic Use

Block Parameter: `Filename`
Type: character vector
Values: character vector
Default: `'untitled.mat'`

`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.
- With fast restart enabled, you can:
  - Change the active scenario
  - Change the active signal
  - Start the Signal Editor user interface and edit data

While you can change the active signal, you cannot edit the signal properties in the block.

Programmatic Use
Block Parameter: `ActiveScenario`
Type: character vector | numeric
Values: character vector | index value
Default: `'Scenario'`

`To create and edit scenarios, launch Signal Editor user interface` — Start Signal Editor

button click

To start Signal Editor user interface, click .

`Active signal` — 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.

To enable this parameter, ensure that the specified MAT-file exists.

Tip Do not use the `set_param` function to set the active signal property Name-Value argument (`'ActiveSignal'`) in combination with another Name-Value pair argument for the Signal Editor block.

Programmatic Use
Block Parameter: `ActiveSignal`
Type: character vector | numeric
Values: character vector | index vector
Default: `'Signal 1'`

`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.

Programmatic Use

Block Parameter: `IsBus`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`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.

Programmatic Use

Block Parameter: `OutputBusObjectStr`
Type: character vector
Values: bus object name
Default: `'Bus: BusObject'`

`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.

Programmatic Use

Block Parameter: `Unit`
Type: character vector
Values: `'inherit'` | supported physical unit
Default: `'inherit'`

`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.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: `-1` | sample time in seconds
Default: `'0'`

`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.

Programmatic Use
Block Parameter: `Interpolate`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`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.

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`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
	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.

Programmatic Use

Block Parameter: `OutputAfterFinalValue`
Type: character vector
Values: `'Setting to zero' | 'Extrapolation' | 'Holding final value'`
Default: `'Setting to zero'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`No`

Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

See Also

Signal Builder | `signalBuilderToSignalEditor` | `signalEditor` | `str2double`

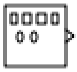
Topics

"Create Bus Objects with the Bus Editor"
"Create and Edit Signal Data"

Introduced in R2017b

Signal Generator

Generate various waveforms
Library: Simulink / Sources



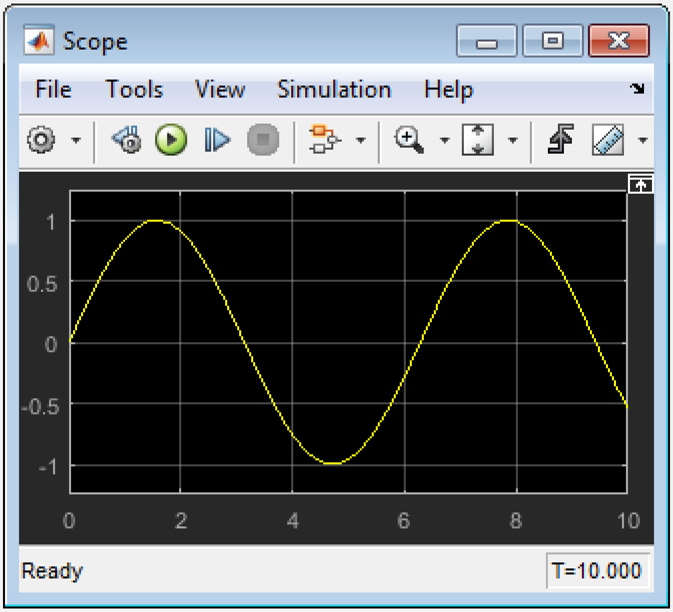
Description

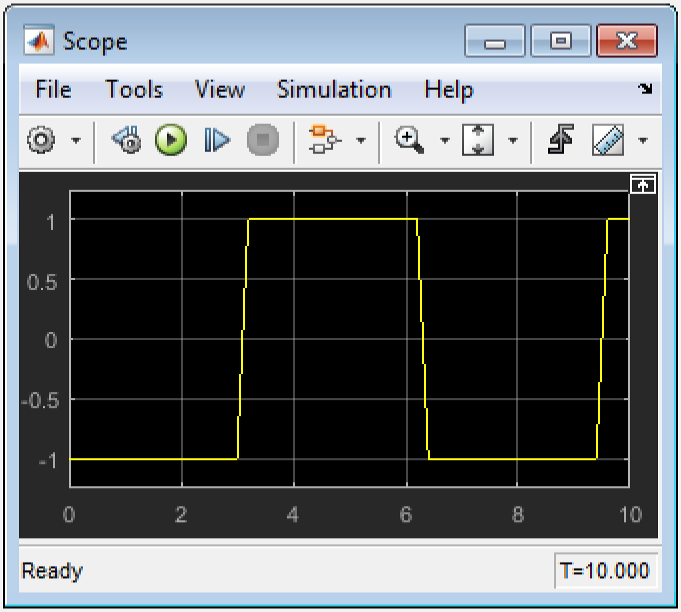
Supported Operations

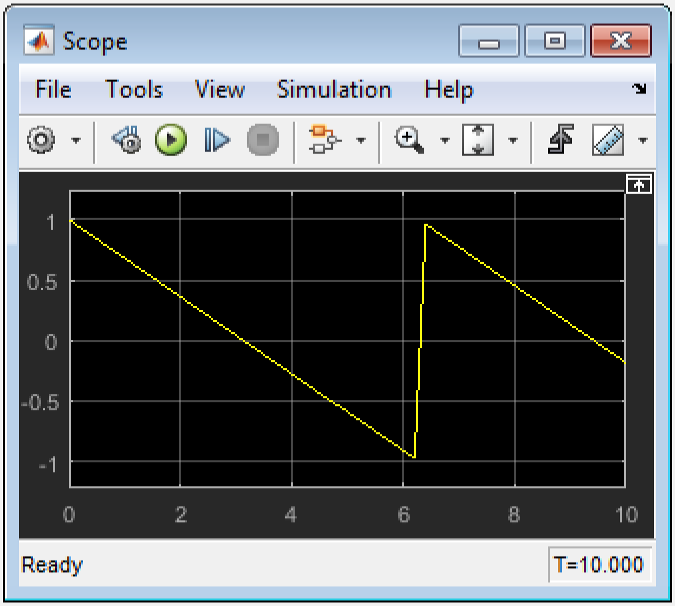
The Signal Generator block can produce one of four different waveforms:

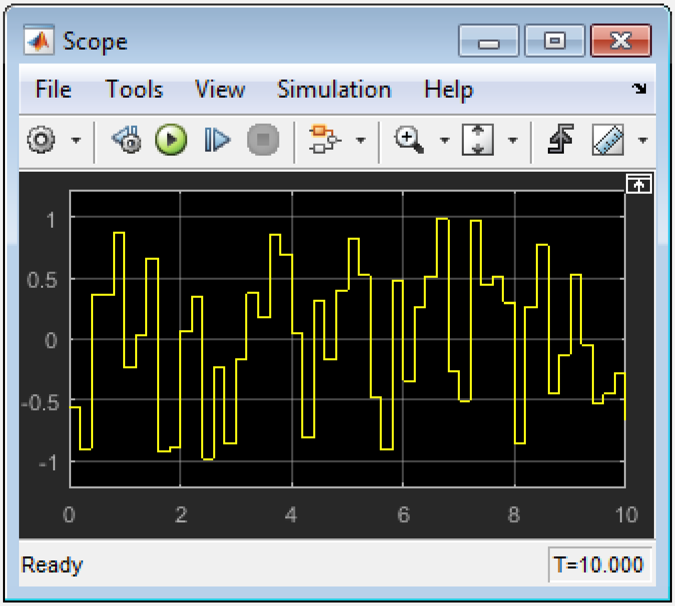
- sine
- square
- sawtooth
- random

You can express signal parameters in hertz or radians per second. Using default parameter values, you get one of the following waveforms:

Waveform	Scope Output
Sine wave	 Scope File Tools View Simulation Help 1 0.5 0 -0.5 -1 0 2 4 6 8 10 Ready T=10.000

Waveform	Scope Output
Square wave	 Scope File Tools View Simulation Help 1 0.5 0 -0.5 -1 0 2 4 6 8 10 Ready T=10.000

Waveform	Scope Output
Sawtooth wave	 Scope File Tools View Simulation Help 1 0.5 0 -0.5 -1 0 2 4 6 8 10 Ready T=10.000

Waveform	Scope Output
Random wave	 Scope File Tools View Simulation Help 1 0.5 0 -0.5 -1 0 2 4 6 8 10 Ready T=10.000

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.

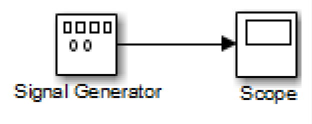
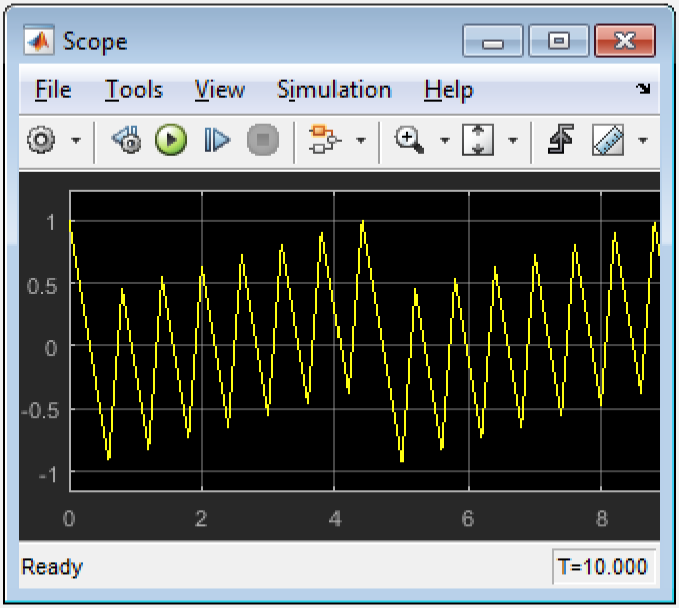
You can vary the output settings of the Signal Generator block while a simulation is in progress to determine quickly the response of a system to different types of inputs.

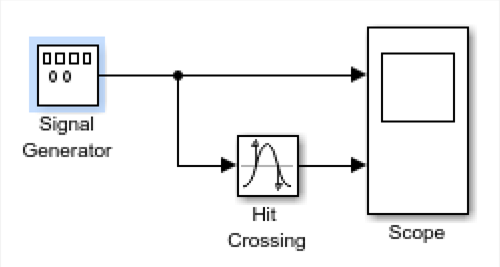
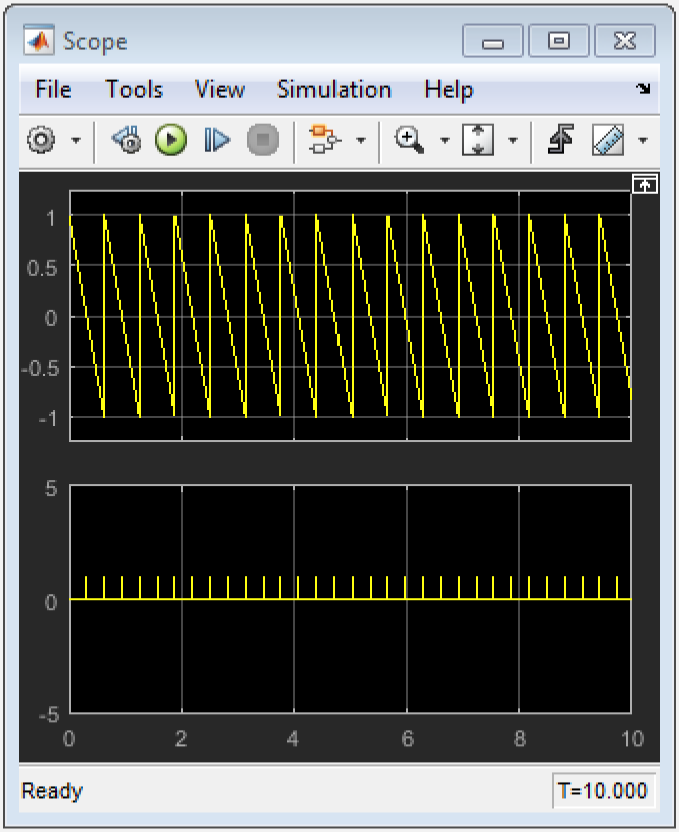
The **Amplitude** and **Frequency** parameters determine the amplitude and frequency of the output signal. The parameters must be of the same dimensions after scalar expansion. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensions as the **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. These models show the difference in Signal Generator block output with and without the Hit Crossing block.

Model That Uses a Variable-Step Solver	Signal Generator Block Output
	 Scope File Tools View Simulation Help 1 0.5 0 -0.5 -1 0 2 4 6 8 Ready T=10.000

Model That Uses a Variable-Step Solver	Signal Generator Block Output
	 Scope File Tools View Simulation Help 1 0.5 0 -0.5 -1 5 0 -5 0 2 4 6 8 10 Ready T=10.000

Ports

Output

Port_1 — Generated output signal
scalar | vector | matrix

Output signal specified as one of these waveforms.

- sine
- square
- sawtooth
- random

Data Types: `double`

Parameters

`Wave form` — Wave form to generate

`sine` (default) | `square` | `sawtooth` | `random`

Specify the wave form.

Programmatic Use
Block Parameter: `WaveForm`
Type: character vector
Values: `'sine'` | `'square'` | `'sawtooth'` | `'random'`
Default: `'sine'`

`Time (t)` — Source of time variable

`Use simulation time` (default) | `Use external signal`

Specify whether to use simulation time or an external signal as the source of values for the waveform time variable. If you specify an external source, the block displays an input port for connecting the source.

Programmatic Use
Block Parameter: `TimeSource`
Type: character vector
Values: `'Use simulation time'` | `'Use external signal'`
Default: `'Use simulation time'`

`Amplitude` — Signal amplitude

`1` (default) | scalar

Specify the amplitude of the generated waveform.

Programmatic Use
Block Parameter: `Amplitude`

Type: character vector
Values: real scalar
Default: '1'

Frequency — Signal frequency

1 (default) | scalar

Specify the frequency of the generated waveform.

Programmatic Use
Block Parameter: Frequency
Type: character vector
Values: real scalar
Default: '1'

Units — Signal units

rad/sec (default) | Hertz

Specify the signal units as Hertz or rad/sec.

Programmatic Use
Block Parameter: Units
Type: character vector
Values: 'rad/sec' | 'Hertz'
Default: 'rad/sec'

Interpret vector parameters as 1-D — Treat vectors as 1-D

on (default) | off

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.

- When you select this check box, the block 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.
- When you clear this check box, the block does not output a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector.

Programmatic Use
Block Parameter: VectorParams1D
Type: character vector
Values: 'on' | 'off'

Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Cannot be used inside a triggered subsystem hierarchy.

These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.

See Also

Pulse Generator | Signal Builder | Waveform Generator

Topics

“Scenarios”

Introduced before R2006a

Signal Specification

Specify desired dimensions, sample time, data type, numeric type, and other attributes of signal

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array

Input signal to check attributes of, specified as a scalar, vector, matrix, or N-D array. The block checks the attributes of the input signal against the desired attributes you specify in the block dialog box. If the attributes do not match, the block generates an error.

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

Output

`Port_1` — Output signal

scalar | vector | matrix | N-D array

Output signal is the input signal when all attributes of the signal match those specified in the dialog box. If the attributes do not match, the block generates an error.

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

Parameters

`Minimum` — Minimum value for range checking

`[]` (default) | scalar

Specify the minimum value for the block output 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 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

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: scalar
Default: `'[ ]'`

`Maximum` — Maximum value for range checking

`[ ]` (default) | scalar

Specify the maximum value for the block output as a finite real double scalar value.

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

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: scalar
Default: `'[ ]'`

`Data type` — Output data type

`Inherit: auto` (default) | `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>` | `Bus: <object name>` | `<data type expression>`

Specify the desired output data type. If the data type of the input signal does not match the value you specify, the block generates an error.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use

Block 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'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding Output data type

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Require nonvirtual bus` — Accept only nonvirtual bus signals

`off` (default) | `on`

If you specify a bus object as the data type, use this parameter to specify whether to accept only nonvirtual bus signals.

- `off` — Specifies that a signal must come from a *virtual* bus.
- `on` — Specifies that a signal must come from a *nonvirtual* bus.

Dependencies

To enable this parameter, set **Data type** to one of these values:

- `Bus: <object name>`

- `<data type expression>` that specifies a bus object

Programmatic Use
Block Parameter: `BusOutputAsStruct`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Unit (e.g., m, m/s^2, N*m)` — Physical unit of the input signal to the block

`inherit` (default) | `<Enter unit>`

Specify the physical unit of the input signal to the block. 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.

Programmatic Use
Block Parameter: `Unit`
Type: character vector
Values: `'inherit'` | `'<Enter unit>'`
Default: `'inherit'`

`Dimensions (-1 for inherited)` — Dimensions of input and output signals

`-1` (default) | `n` | `[m n]`

Specify the dimensions of the input and output signals.

- `-1` — Specifies that signals inherit dimensions.
- `n` — Specifies a vector of width `n`.
- `[m n]` — Specifies a matrix with `m` rows and `n` columns.

Programmatic Use
Block Parameter: `Dimensions`
Type: character vector

Values: `'-1'` | `n` | `[m n]`
Default: `'-1'`

`Variable-size signal` — Allow signal to be variable-size, fixed-size, or both

`Inherit` (default) | `No` | `Yes`

Specify the signal to be of variable-size, fixed size, or both.

- `Inherit` — Allows variable-size and fixed-size signals.
- `No` — Does not allow variable-size signals.
- `Yes` — Allows only variable-size signals.

Dependencies

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.

Programmatic Use

Block Parameter: `VarSizeSig`
Type: character vector
Values: `'Inherit'` | `'No'` | `'Yes'`
Default: `'Inherit'`

`Sample time (-1 for inherited)` — Time interval between samples

`-1` (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

Programmatic Use

Parameter: `SampleTime`
Type: character vector
Values: Any valid sample time
Default: `'-1'`

`Signal type` — Complexity of signal

`auto` (default) | `real` | `complex`

Specify the complexity of the input and output signals.

- `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.

Programmatic Use
Parameter: `SignalType`
Type: character vector
Values: `'auto'` | `'real'` | `'complex'`
Default: `'auto'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For more information about HDL code generation, see Signal Specification.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Unit System Configuration

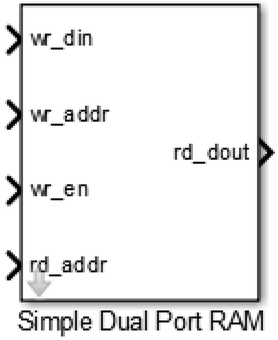
Topics

“Unit Specification in Simulink Models”
“Variable-Size Signal Basics”
“Specify Sample Time”

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`.

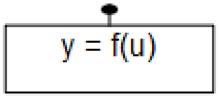
See Also

Dual Port RAM | Dual Rate Dual Port RAM | Single Port RAM

Introduced in R2014a

Simulink Function

Function defined with Simulink blocks
Library: Simulink / 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.

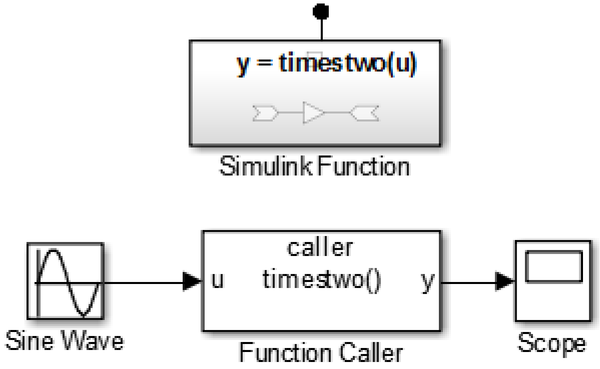
You can visualize Simulink Function calls in the Sequence Viewer. The viewer shows when calls were made with the argument and the return values. See Sequence Viewer block reference.

Function Interface

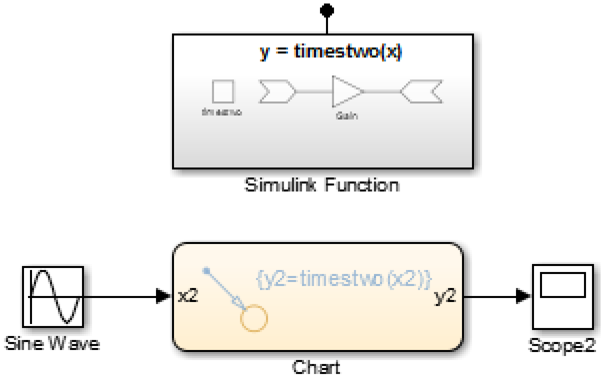
The function interface appears on the face of a Simulink Function block. Editing the block text adds and deletes Argument Inport blocks and Argument Outport blocks from the function definition. Editing also sets the **Function name** parameter in the Trigger block within the Simulink Function block.

For example, entering `y = 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.

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

`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` | `struct`

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` | `struct`

`ArgOut` — Argument output from a subsystem block
scalar | vector | matrix

An Argument Outport 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` | `struct`

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[a] | `string`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[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 code generation support depends on block implementation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks

Argument Inport | Argument Outport | Chart | Function Caller | Function-Call Subsystem | Inport | MATLAB Function | Outport | Subsystem | Trigger

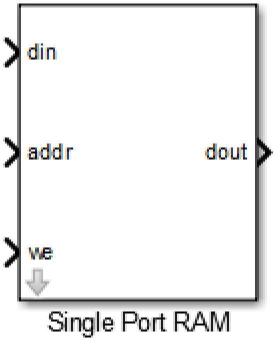
Topics

“Simulink Functions”
“Scoped Simulink Function Blocks in Models”
“Using Simulink Function Blocks and Exported Stateflow Functions”

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

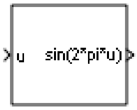
Dual Port RAM | Dual Rate Dual Port RAM | Simple Dual Port RAM

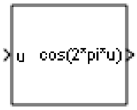
Introduced in R2014a

Sine, Cosine

Implement fixed-point sine or cosine wave using lookup table approach that exploits quarter wave symmetry

Library: Simulink / Lookup Tables





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 block can output the following functions of the input signal, depending upon what you select for the **Output formula** parameter:

- sin(2π*u*)
- cos(2π*u*)
- exp(*j*2π*u*)
- sin(2π*u*) and cos(2π*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.

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.

Tip To simulate a model containing this block without a Fixed-Point Designer license, you must use data type override. For more information, see “Share Fixed-Point Models”.

Ports

Input

u — Input signal to implement as fixed-point sine or cosine wave

real-valued signal

Input signal, u, specified as a real-valued scalar, vector, matrix, or array.

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.

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

Output

`sin(2*pi*u)` — Fixed-point sine wave

real-valued fixed-point signal

Fixed-point sine wave, implemented using a lookup table approach.

Dependencies

This port is enabled when the **Output formula** is set to `sin(2*pi*u)` or `sin(2*pi*u) and cos(2*pi*u)`.

Data Types: `fixed point`

`cos(2*pi*u)` — Fixed-point cosine wave

real-valued fixed-point signal

Fixed-point cosine wave, implemented using a lookup table approach.

Dependencies

This port is enabled when the **Output formula** is set to cos(2*pi*u) or sin(2*pi*u) and cos(2*pi*u).

Data Types: fixed point

exp(j*2*pi*u) — exp(j*2*pi*u)

complex-valued fixed-point signal

exp(j*2*pi*u), implemented using a lookup table approach.

Dependencies

This port is enabled when the **Output formula** is set to exp(j*2*pi*u).

Data Types: fixed point

Parameters

Output formula — Select the signal(s) to output

cos(2*pi*u) | sin(2*pi*u) | exp(j*2*pi*u) | sin(2*pi*u) and cos(2*pi*u)

Programmatic Use
Block Parameter: Formula
Values: 'sin(2*pi*u)' | 'cos(2*pi*u)' | 'exp(j*2*pi*u)' | 'sin(2*pi*u) and cos(2*pi*u)'

Number of data points for lookup table — Specify the number of data points to retrieve from the lookup table

(2^5)+1 (default) | integer, greater than or equal to 2

The implementation is most efficient when you specify the lookup table data points to be (2^*n*)+1, where *n* is an integer. To be compatible with the **Output word length** parameter, the **Number of data points for lookup table** must be less than or equal to (2^(**Output word length**-2)+1).

Programmatic Use
Block Parameter: NumDataPoints
Type: scalar
Value: integer >= 2
Default: '(2^5)+1'

`Output word length` — Specify the word length for the fixed-point data type of the output signal

`16` (default) | integer from 2 to 53

The fraction length of the output is the output word length minus 2. To be compatible with the **Number of data points for lookup table** parameter, (2^(**Output word length** - 2) +1) must be greater than or equal to **Number of data points for lookup table**.

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.

Programmatic Use
Block Parameter: `OutputWordLength`
Type: scalar
Value: `integer from 2 to 53`
Default: `'16'`

`Internal rule priority for lookup table` — Specify the internal rule for intermediate calculations

`Speed` (default) | `Precision`

Select `Speed` for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

Programmatic Use
Block Parameter: `InternalRulePriority`
Values: `'Speed' | 'Precision'`
Default: `'Speed'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`

Multidimensional Signals	No
Variable-Size Signals	No
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™.

For information about HDL code generation, see Cosine.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Sine Wave | Trigonometric Function

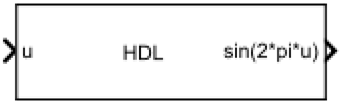
Topics

“About Lookup Table Blocks”

Introduced before R2006a

Sine HDL Optimized

Implement fixed-point sine wave by using lookup table approach that exploits quarter wave symmetry



Sine
HDL Optimized

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(2π*u*)
- cos(2π*u*)
- exp(*i*2π*u*)
- sin(2π*u*) and cos(2π*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

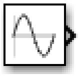
Cosine HDL Optimized | Sine, Cosine | Trigonometric Function

Introduced in R2016b

Sine Wave

Generate sine wave, using simulation time as time source

Library: Simulink / 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 block calculates the output waveform.

$$y = amplitude \times \sin(frequency \times time + phase) + bias.$$

In time-based 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.

For more information, see “Specify Sample Time”.

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.

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*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 these 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.
Set up the Sine Wave block to use the `sin()` math library function to calculate block output. **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.	The `sin()` math library function computes block output at each time step *independently* of output values from other time steps, preventing the accumulation of round-off errors.

Sample-Based Mode

Sample-based mode uses this formula to compute the output of the Sine Wave block.

$$y = A\sin(2\pi(k+o)/p)+b$$

- 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 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. 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.

Ports

Output

`Port_1` — Sine wave output signal

scalar | vector

Output sine wave signal created based on the block parameter values.

Data Types: `double`

Parameters

`Sine type` — Type of sine wave

`Time based` (default) | `Sample based`

Specify the type of sine wave that this block generates. Some parameters in the dialog box appear depending on whether you select time-based or sample-based.

Programmatic Use
Block Parameter: `SineType`
Type: character vector
Values: `'Time based'` | `'Sample based'`
Default: `'Time based'`

`Time (t)` — Source of time variable

`Use simulation time` (default) | `Use external signal`

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 creates an input port for the time source. When you select an external time source, the block is the same as the Sine Wave Function block.

Programmatic Use

Block Parameter: `TimeSource`
Type: character vector
Values: `'Use simulation time'` | `'Use external signal'`
Default: `'Use simulation time'`

`Amplitude` — Amplitude of the sine wave

`1` (default) | scalar

Specify the amplitude of the output sine wave signal.

Programmatic Use

Block Parameter: `Amplitude`
Type: character vector
Value: scalar
Default: `'1'`

`Bias` — Constant added to sine wave

`0` (default) | scalar

Specify the constant value added to the sine to produce the output.

Programmatic Use

Block Parameter: `Bias`
Type: character vector
Value: scalar
Default: `'0'`

`Frequency (rad/sec)` — Frequency of sine wave

`1` (default) | scalar

Specify the frequency, in rad/sec.

Dependencies

To enable this parameter, set **Sine type** to `Time based`.

Programmatic Use

Block Parameter: `Frequence`
Type: character vector
Value: scalar
Default: `'1'`

`Phase (rad)` — Phase shift of sine wave

`0` (default) | scalar

Specify the phase shift of the sine wave.

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).

Dependencies

To enable this parameter, set **Sine type** to `Time based`.

Programmatic Use

Block Parameter: `Phase`
Type: character vector
Value: scalar
Default: `'0'`

`Samples per period` — Samples per period

`0` (default) | `integer`

Specify the number of samples per period.

Dependencies

To enable this parameter, set **Sine type** to `Sample based`.

Programmatic Use

Block Parameter: `Samples`
Type: character vector
Value: scalar
Default: `'10'`

`Number of offset samples` — Offset in number of time samples

`0` (default) | `integer`

Specify the offset (discrete phase shift) in number of sample times.

Dependencies

To enable this parameter, set **Sine type** to `Sample based`.

Programmatic Use

Block Parameter: `Offset`
Type: character vector
Value: scalar
Default: `'0'`

`Sample time` — Sample period

`0` (default) | scalar

Specify the sample period in seconds. The default is `0`. If the sine type is sample-based, the sample time must be greater than 0. See “Specify Sample Time”.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Value: scalar
Default: `'0'`

`Interpret vector parameters as 1-D` — Output dimensions for one-row or one-column matrices

`off` (default) | `on`

Specify the output dimensions to be a 1-D vector signal when other parameters are one-row and one-column matrices. If you do not select this box, the block 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.

Programmatic Use

Block Parameter: `VectorParams1D`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Depends on absolute time when placed inside a triggered subsystem hierarchy. These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.

See Also

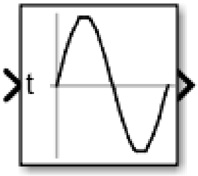
Sine Wave Function | Sine, Cosine

Introduced before R2006a

Sine Wave Function

Generate sine wave, using external signal as time source

Library: Simulink / Math Operations



Description

The Sine Wave Function 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 block that appears in the Math Operations library. If you select `Use simulation time` for the **Time** parameter in the block dialog box, you get the Sine Wave Function block.

Time-Based Mode

The block calculates the output waveform.

$$y = amplitude \times \sin(frequency \times time + phase) + bias.$$

In time-based 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.

For more information, see “Specify Sample Time”.

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.

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*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 these 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.
Set up the Sine Wave block to use the `sin()` math library function to calculate block output. **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.	The `sin()` math library function computes block output at each time step *independently* of output values from other time steps, preventing the accumulation of round-off errors.

Sample-Based Mode

Sample-based mode uses this formula to compute the output of the Sine Wave block.

$$y = A\sin(2\pi(k+o)/p)+b$$

- 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 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. 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.

Ports

Input

`Port_1` — Time source signal

scalar

Input signal representing the time source in the sine wave calculation.

Data Types: `double`

Output

`Output 1` — Output sine wave signal

scalar

Output signal that is the created sine wave.

Data Types: `double`

Parameters

`Sine type` — Type of sine wave

`Time based` (default) | `Sample based`

Specify the type of sine wave that this block generates. Some parameters in the dialog box appear depending on whether you select time-based or sample-based.

Programmatic Use
Block Parameter: `SineType`
Type: character vector
Values: `'Time based'` | `'Sample based'`
Default: `'Time based'`

`Time (t)` — Source of time variable

`Use external signal` (default) | `Use simulation 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 creates an input port for the time source.

Programmatic Use

Block Parameter: `TimeSource`
Type: character vector
Values: `'Use simulation time'` | `'Use external signal'`
Default: `'Use external signal'`

`Amplitude` — Amplitude of the sine wave

`1` (default) | scalar

Specify the amplitude of the output sine wave signal.

Programmatic Use

Block Parameter: `Amplitude`
Type: character vector
Value: scalar
Default: `'1'`

`Bias` — Constant added to sine wave

`0` (default) | scalar

Specify the constant value added to the sine to produce the output.

Programmatic Use

Block Parameter: `Bias`
Type: character vector
Value: scalar
Default: `'0'`

`Frequency (rad/sec)` — Frequency of sine wave

`1` (default) | scalar

Specify the frequency, in radians per second.

Dependency

To enable this parameter, set **Sine type** to `Time based`.

Programmatic Use

Block Parameter: `Frequence`
Type: character vector
Value: scalar
Default: `'1'`

`Phase (rad)` — Phase shift of sine wave

`0` (default) | scalar

Specify the phase shift of the sine wave.

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).

Dependencies

To enable this parameter, set **Sine type** to `Time based`.

Programmatic Use

Block Parameter: `Phase`
Type: character vector
Value: scalar
Default: `'0'`

`Samples per period` — Samples per period

`0` (default) | `integer`

Specify the number of samples per period.

Dependencies

To enable this parameter, set **Sine type** to `Sample based`.

Programmatic Use

Block Parameter: `Samples`

Type: character vector
Value: scalar
Default: `'10'`

`Number of offset samples` — Offset in number of time samples

`0` (default) | `integer`

Specify the offset (discrete phase shift) in number of sample times.

Dependencies

To enable this parameter, set **Sine type** to `Sample based`.

Programmatic Use
Block Parameter: `Offset`
Type: character vector
Value: scalar
Default: `'0'`

`Sample time` — Sample period

`0` (default) | scalar

Specify the sample period in seconds. The default is `0`. If the sine type is sample-based, the sample time must be greater than 0. See “Specify Sample Time”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Value: scalar
Default: `'0'`

`Interpret vector parameters as 1-D` — Output dimensions for one-row or one-column matrices

`off` (default) | `on`

Specify the output dimensions to be a 1-D vector signal when other parameters are one-row and one-column matrices. If you do not select this box, the block 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.

Programmatic Use
Block Parameter: `VectorParams1D`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Depends on absolute time when placed inside a triggered subsystem hierarchy. These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.

See Also

Sine Wave | Sine, Cosine

Introduced before R2006a

Slider

Tune parameter value with sliding scale

Library: Simulink / Dashboard

Description

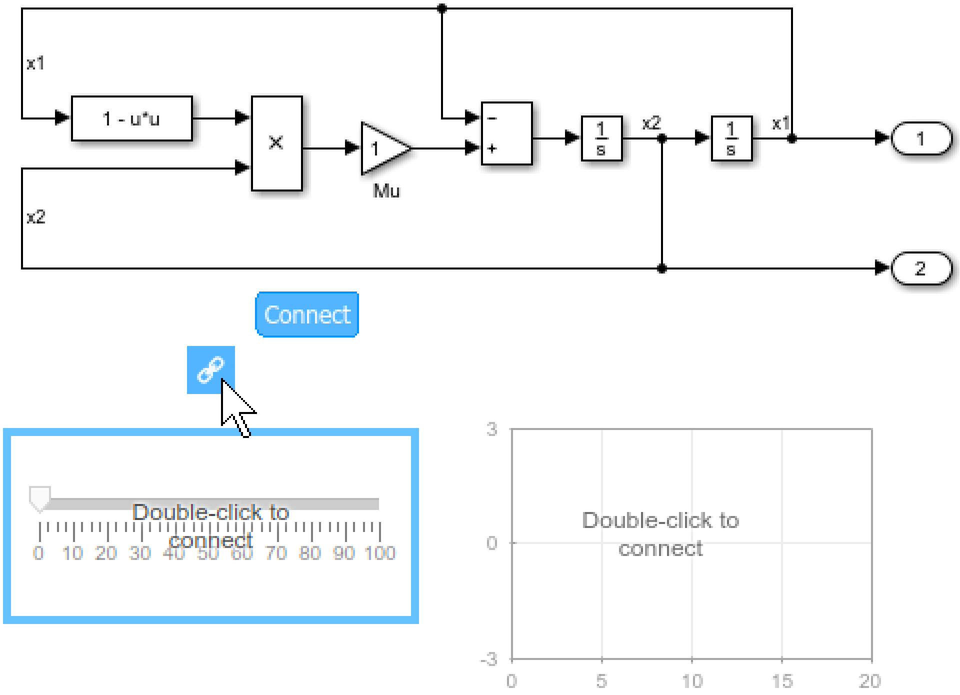
The Slider block tunes the value of the connected block parameter during simulation. For example, you can connect the Slider block to a Gain block in your model and adjust its value during simulation. You can modify the range of the Slider block's scale to fit your data. Use the Slider block with other Dashboard blocks to create an interactive dashboard to control your model.

Connecting Dashboard Blocks

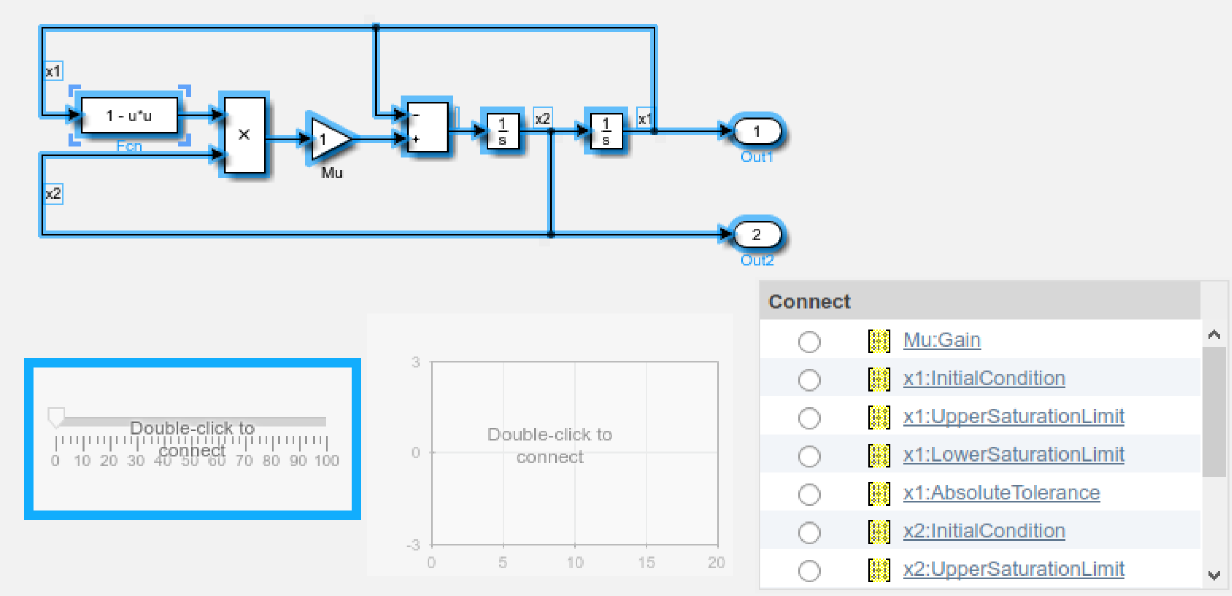
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○ Mu:Gain
○ x1:InitialCondition
○ x1:UpperSaturationLimit
○ x1:LowerSaturationLimit
○ x1:AbsoluteTolerance
○ x2:InitialCondition
○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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 arc. 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 arc. 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 arc. 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.

See Also

Knob | Rotary Switch

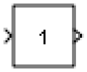
Topics

"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: Simulink / Math Operations



Description

The Slider Gain block performs a scalar gain that you can modify during simulation. Modify the gain using the slider parameter.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

The Slider Gain block accepts real or complex-valued scalar, vector, or matrix input. The block supports fixed-point data types. If the input of the Slider Gain block is real and gain is complex, the output is complex.

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

Output

`Port_1` — Input multiplied by gain
scalar | vector | matrix

The Slider Gain block outputs the input multiplied by a constant gain value. When the input to the block is real and gain is complex, the output is complex.

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

Parameters

Slider gain — Gain value

0 (default) | real value

Chose the gain value applied to the input.

Programmatic Use
Block Parameter: gain
Type: character vector
Values: real scalar
Default: '1'

Low — Lower limit of the slider range

0 (default) | real value

Specify the lower limit of the slider range.

Programmatic Use
Block Parameter: low
Type: character vector
Values: real scalar
Default: '0'

High — Upper limit of slider range

2 (default) | real value

Specify the upper limit of the slider range. The default is 2.

Programmatic Use
Block Parameter: high
Type: character vector
Values: real scalar
Default: '2'

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

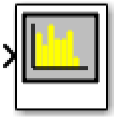
Gain

Introduced before R2006a

Spectrum Analyzer

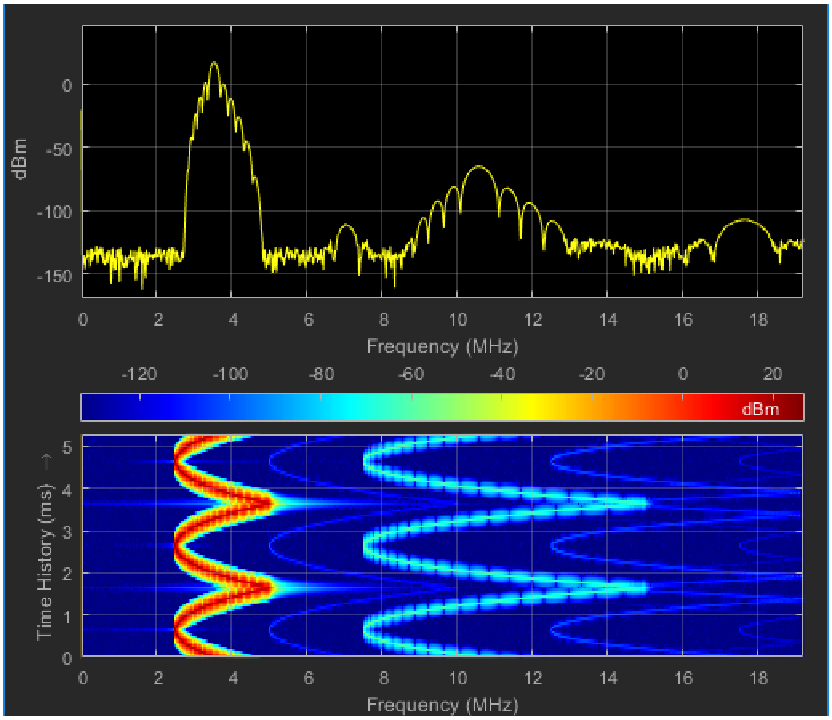
Display frequency spectrum

Library: DSP System Toolbox / Sinks



Description

The Spectrum Analyzer block, referred to here as the scope, displays the frequency spectra of signals.



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 “Conditionally Executed Subsystems Overview” 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. Input signals can have these characteristics:

- **Signal Domain** — Frequency or time signals
- **Type** — Discrete (sample-based and frame-based).
- **Data type** — Any data type that Simulink supports. See “Data Types Supported by Simulink”.
- **Dimension** — One dimensional (vector), two dimensional (matrix), or multidimensional (array). Input must have fixed number of channels. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

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

`Input domain` — Domain of the input signal

`Time` (default) | `Frequency`

The domain of the input signal you want to visualize. If you visualize time-domain signals, the signal is transformed to the frequency spectrum based on the algorithm specified by the Method parameter.

Programmatic Use

See InputDomain.

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

Dependency

To use this parameter, set “Input domain” (DSP System Toolbox) to `Time`.

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

Dependency

To use this parameter, set “Input domain” (DSP System Toolbox) to Time.

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

Dependency

To use this parameter, set “Input domain” (DSP System Toolbox) to `Time`.

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:

- Set “Input domain” (DSP System Toolbox) to `Time`.
- 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:

- Set “Input domain” (DSP System Toolbox) to `Time`.
- 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 “FStop (Hz)” (DSP System Toolbox) 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:

- Set “Input domain” (DSP System Toolbox) to `Time`.
- 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:

- Set “Input domain” (DSP System Toolbox) to `Time`.
- 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.

`Frequency (Hz)` — Frequency vector

`Auto` (default) | `Input port` | monotonically increasing vector

Set the frequency vector which determines the *x*-axis of the display.

- `Auto` — The frequency vector is calculated from the length of the input. See “Frequency Vector” (DSP System Toolbox).
- `Input port` — When selected, an input port appears on the block for the frequency vector input.
- Custom vector — Enter a custom vector as the frequency vector. The length of the custom vector must be equal to the frame size of the input signal.

Tunable: No

Dependency

To use this parameter, set “Input domain” (DSP System Toolbox) to `Frequency`.

Programmatic Use

See FrequencyVector.

`RBW (Hz)` — Resolution bandwidth

`Auto` (default) | `Input port` | 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 greater than two:

$$\frac{span}{RBW} > 2$$

For frequency input only, you can use an input port to set the RBW value.

Tunable: Yes

Dependency

To use this parameter, set either:

- “Input domain” (DSP System Toolbox) to `Time` and the **RBW (Hz)/Window length/ Number of frequency bands** dropdown to `RBW (Hz)`.
- “Input domain” (DSP System Toolbox) to `Frequency`.

Programmatic Use

See RBW.

`Input units` — Units of frequency input

`Auto` (default) | `dBm` | `dBV` | `dBW` | `Vrms` | `Watts`

Select the units of the frequency-domain input. This property allows the Spectrum Analyzer to scale frequency data if you choose a different display unit with the “Units” (DSP System Toolbox) property.

Tunable: No

Dependency

This option is only available when “Input domain” (DSP System Toolbox) is set to `Frequency`.

Programmatic Use

See InputUnits.

`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`

Dependency

To use this parameter, set "Input domain" (DSP System Toolbox) to `Time`.

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 even integer

Specify the number of filter taps or coefficients for each frequency band. This number must be a positive even integer. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is equal to *Taps Per Band* + *FFT Length*.

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.

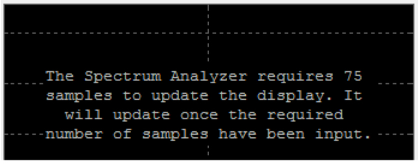
`Samples/update` — Required number of input samples

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.



The Spectrum Analyzer requires 75 samples to update the display. It will update once the required number of samples have been input.

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`.

Programmatic Use

See SpectrogramChannel.

`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

Dependencies

To enable this parameter, set:

- "Input domain" (DSP System Toolbox) to `Time`
- "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.

Dependency

To use this parameter, set "Input domain" (DSP System Toolbox) to `Time`.

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 choose 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).

Dependency

To use this parameter, set "Input domain" (DSP System Toolbox) to `Time`.

Programmatic Use

See "FrequencyOffset" (DSP System Toolbox).

`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 box 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 | string

Specify the display title. 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 | string

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)` | `parula(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 box 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 box 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. 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`.

Programmatic Use

See AxesScalingNumUpdates.

`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

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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:

$$\frac{span}{RBW} > 2$$

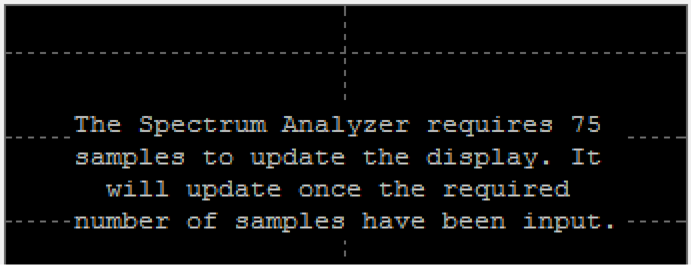
By default, the **RBW (Hz)** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span. When you set **RBW (Hz)** to `Auto`, *RBW* is calculated as:

$$RBW_{auto} = \frac{span}{1024}$$

5 When in **Window Length** mode, you specify N_{window} and the resulting *RBW* is:

$$\frac{NENBW * Fs}{N_{window}}$$

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:



```
The Spectrum Analyzer requires 75
samples to update the display. It
  will update once the required
number of samples have been input.
```

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:

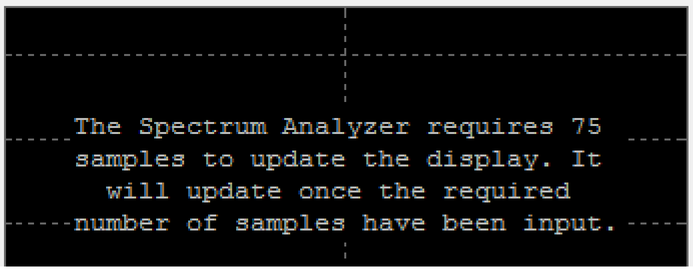
$$RBW = \frac{F_s}{\text{Input Frame Size}}$$

When the number of frequency bands is manually specified, the resulting RBW is:

$$RBW = \frac{F_s}{FFTLength}$$

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:



The Spectrum Analyzer requires 75 samples to update the display. It will update once the required number of samples have been input.

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.

$$P_{spectrum}(f) = PSD(f) \times RBW = PSD(f) \times \frac{F_s \times NENBW}{N_{window}} = \frac{1}{P} \sum_{p=1}^{P} \frac{\left| \sum_{n=1}^{N_{FFT}} x^{(p)}[n] e^{-j2\pi f(n-1)T} \right|^2}{\left[\sum_{n=1}^{N_{window}} w[n] \right]^2}$$

Frequency Vector

When set to `Auto`, the frequency vector for frequency-domain input is calculated by the software.

When the PlotAsTwoSidedSpectrum property is set to true, the frequency vector is:

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2} \right]$$

When the PlotAsTwoSidedSpectrum property is set to false, the frequency vector is:

$$\left[0, \frac{SampleRate}{2} \right]$$

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

   $$\frac{\textit{100 - Occupied BW \%}}{\textit{2}}$$

   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

$$\frac{100 - \textit{Occupied BW \%}}{2}$$

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 near 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 $W_{sum} = W_0 + W_1 + W_2 +...+ W_n$.

The sum of powers of the second and higher-order harmonics are computed as $P_{harmonic} = P_2 + P_3 + P_4 + \ldots + P_n$.

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.

$$THD = 10 \cdot \log_{10}\left(\frac{P_{harmonic}}{P_1}\right)$$

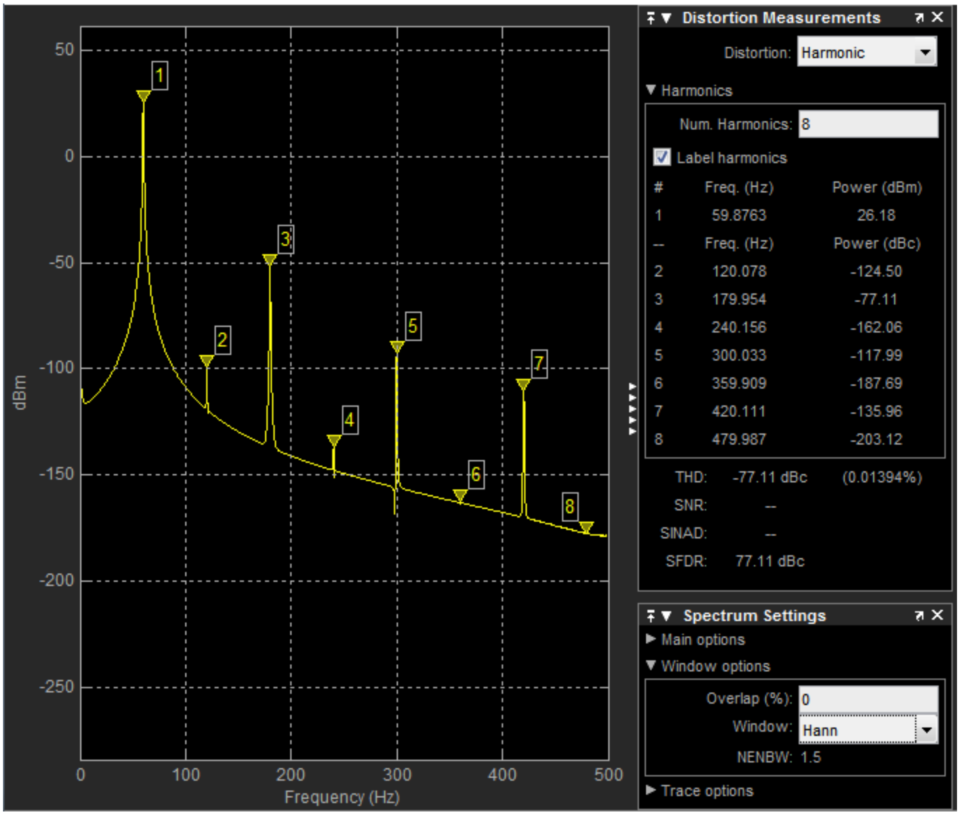
$$SINAD = 10 \cdot \log_{10}\left(\frac{P_1}{P_{harmonic} + P_{noise}}\right)$$

$$SNR = 10 \cdot \log_{10}\left(\frac{P_1}{P_{noise}}\right)$$

$$SFDR = 10 \cdot \log_{10}\left(\frac{P_1}{\max\left(P_{maxspur}, \max(P_2, P_3, \ldots, P_n)\right)}\right)$$

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.



Distortion: Harmonic

Harmonics

Num. Harmonics: 8

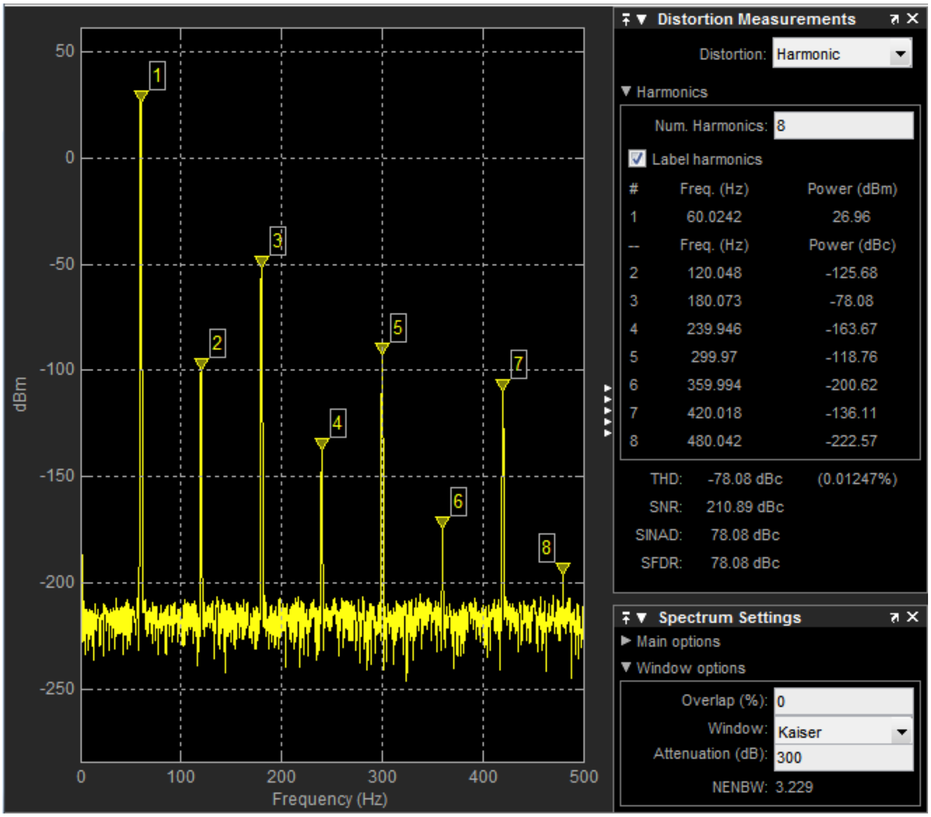
☑ Label harmonics

#	Freq. (Hz)	Power (dBm)
1	59.8763	26.18
--	Freq. (Hz)	Power (dBc)
2	120.078	-124.50
3	179.954	-77.11
4	240.156	-162.06
5	300.033	-117.99
6	359.909	-187.69
7	420.111	-135.96
8	479.987	-203.12

THD: -77.11 dBc (0.01394%)
SNR: --
SINAD: --
SFDR: 77.11 dBc

Overlap (%): 0
Window: Hann
NENBW: 1.5

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.



Distortion Measurements

Distortion: Harmonic

▼ Harmonics

Num. Harmonics: 8

☑ Label harmonics

#	Freq. (Hz)	Power (dBm)
1	60.0242	26.96
--	Freq. (Hz)	Power (dBc)
2	120.048	-125.68
3	180.073	-78.08
4	239.946	-163.67
5	299.97	-118.76
6	359.994	-200.62
7	420.018	-136.11
8	480.042	-222.57

THD:	-78.08 dBc	(0.01247%)
SNR:	210.89 dBc	
SINAD:	78.08 dBc	
SFDR:	78.08 dBc	

Spectrum Settings

▶ Main options

▼ Window options

Overlap (%): 0

Window: Kaiser

Attenuation (dB): 300

NENBW: 3.229

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 Nth 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):

- $TOI_{lower} = P_{F1} + (P_{F2} - P_{(2F1-F2)})/2$
- $TOI_{upper} = P_{F2} + (P_{F1} - P_{(2F2-F1)})/2$
- $TOI = + (TOI_{lower} + TOI_{upper})/2$

Extended Capabilities

HDL Code Generation

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

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

See Also

Objects
`Spectrum Analyzer Configuration`

System Objects
`dsp.SpectrumAnalyzer`

Functions
`getMeasurementsData` | `getSpectralMaskStatus` | `getSpectrumData`

Blocks
Array Plot | Time Scope

Topics

"Display Frequency-Domain Data in Spectrum Analyzer" (DSP System Toolbox)
"Spectral Analysis" (DSP System Toolbox)
"Display Frequency-Domain Data in Spectrum Analyzer" (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.

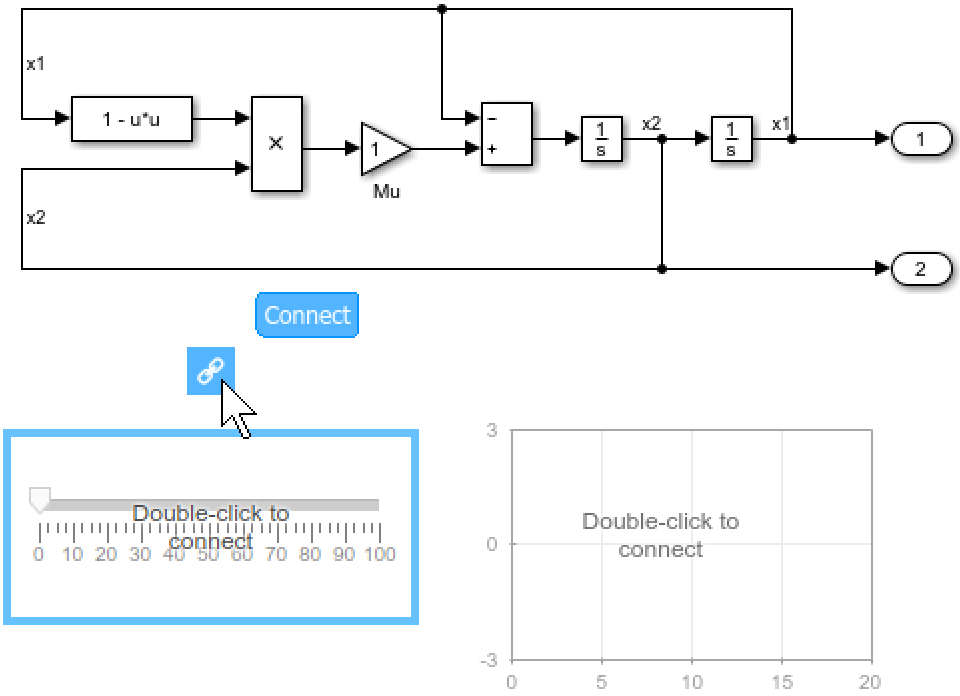
Double-clicking the Slider Switch block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

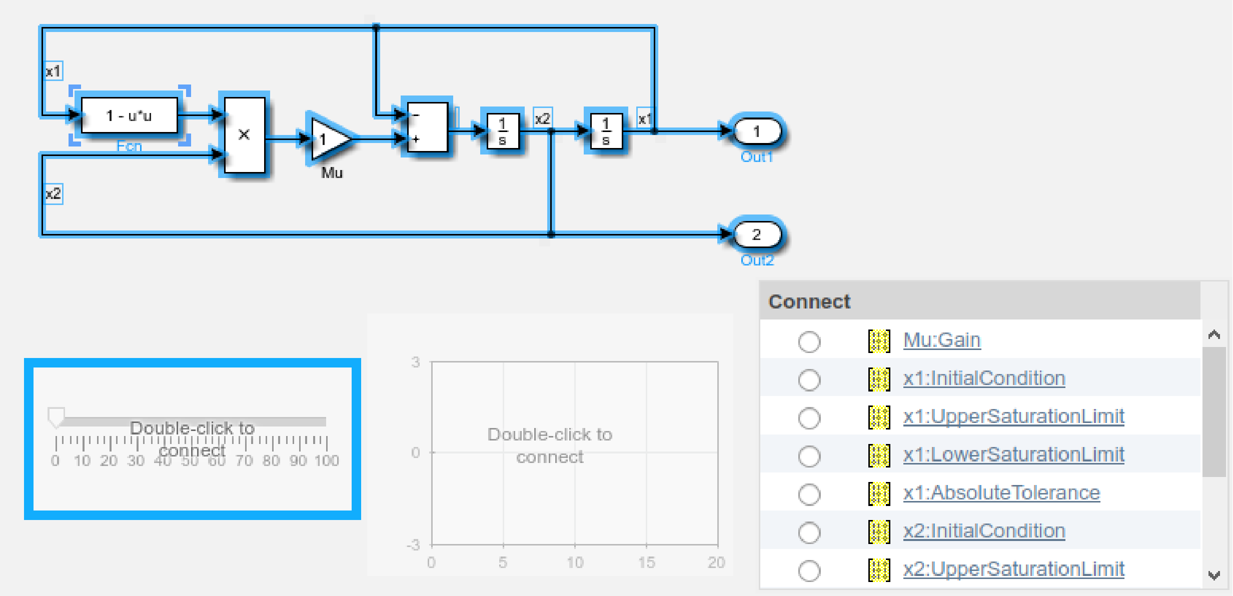
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect	
○	Mu:Gain
○	x1:InitialCondition
○	x1:UpperSaturationLimit
○	x1:LowerSaturationLimit
○	x1:AbsoluteTolerance
○	x2:InitialCondition
○	x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

See Also

Rocker Switch | Rotary Switch | Toggle Switch

Topics

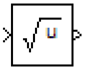
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a

Sqrt

Calculate square root, signed square root, or reciprocal of square root

Library: Simulink / Math Operations



Description

The Sqrt block calculates the square root, signed square root, or reciprocal of square root on the input signal. Select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
sqrt	Square root of the input	$u^{0.5}$	sqrt
signedSqrt	Square root of the absolute value of the input, multiplied by the sign of the input	$\mathrm{sign}(u)*\lvert u\rvert^{0.5}$	—
rSqrt	Reciprocal of the square root of the input	$u^{-0.5}$	—

The block icon changes to match the function.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix

Input signal to the block to calculate the square root, signed square root, or reciprocal of square root. The `sqrt` function accepts real or complex inputs, except for complex fixed-point signals. `signedSqrt` and `rSqrt` do not accept complex inputs.

If the input is negative, set the **Output signal** to complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal that is the square root, signed square root, or reciprocal of square root of the input signal. When the input is an integer or fixed-point type, the output must be floating point.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

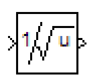
Main

`Function` — Function the block performs

`sqrt` (default) | `signedSqrt` | `rSqrt`

Specify the mathematical function that the block calculates. The block icon changes to match the function you select.

Function	Block Icon
`sqrt`	$\sqrt{u}$
`signedSqrt`	$\pm\sqrt{\lvert u \rvert}$

Function	Block Icon
rSqrt	

Programmatic Use
Block Parameter: `Operator`
Type: character vector
Values: `'sqrt'` | `'signedSqrt'` | `'rSqrt'`
Default: `'sqrt'`

`Output signal type` — Output signal type

`auto` (default) | `real` | `complex`

Specify the output signal type of the block.

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`

Programmatic Use
Block Parameter: `OutputSignalType`
Type: character vector
Values: `'auto'` | `'real'` | `'complex'`
Default: `'auto'`

`Sample time` — Specify sample time as a value other than -1

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Algorithm

`Method` — Method to compute reciprocal of square root

`Exact` (default) | `Newton-Raphson`

Specify the method for computing the reciprocal of a square root. This parameter is only valid for the `rSqrt` function.

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.

Programmatic Use
Block Parameter: `AlgorithmType`

Type: character vector
Values: `'Exact'` | `'Newton-Raphson'`
Default: `'Exact'`

`Number of iterations` — Number of iterations used for Newton Raphson algorithm

3 (default) | integer

Specify the number of iterations to perform the Newton-Raphson algorithm. This parameter is valid with the `rSqrt` function and the `Newton-Raphson` value for **Method**.

Note If you enter 0, the block output is the initial guess of the Newton-Raphson algorithm.

Programmatic Use
Block Parameter: `Iterations`
Type: character vector
Values: integer
Default: `'3'`

Data Types

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Intermediate results data type` — Data type of intermediate results

`Inherit:Inherit via internal rule` (default) | `Inherit: Inherit: Inherit from input` | `Inherit: Inherit: Inherit from output` | `double` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Specify the data type for intermediate results when you set **Function** to `sqrt` or `rSqrt` on the **Main** pane.

The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

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 *not* 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.

Programmatic Use
Block Parameter: `IntermediateResultsDataTypeStr`
Type: character vector
Values: `'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)`, `fixdt(1,16,2^0,0)`. `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Output — Output data type

Inherit: Same as first input (default) | Inherit: Inherit via internal rule | Inherit: Inherit via back propagation | double | single | int8 | int32 | uint32 | fixdt(1,16,2^0,0) | <data type expression> | ...

Specify the output data type. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: '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), fixdt(1,16,2^0,0). '<data type expression>'
Default: 'Inherit: Same as first input'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types

off (default) | on

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).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Saturate on integer overflow — Choose the behavior when integer overflow occurs

on (default) | boolean

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 `int8` (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.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use

Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For information on HDL code generation support, see Sqrt,Signed Sqrt.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Math Function | Trigonometric Function

Introduced in R2010a

Squeeze

Remove singleton dimensions from multidimensional signal
Library: Simulink / 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, vector, and matrix signals pass through the Squeeze block unchanged.

Ports

Input

`Port_1` — Multidimensional input signal
multidimensional signal

Input signal that has any singleton dimensions removed in the output.

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

Output

`Port_1` — Output signal with no singleton dimensions
multidimensional signal

Output signal with no singleton dimensions. For example, a multidimensional array of size 3-by-1-by-2 changes into a 3-by-2 signal. If there are no singleton dimensions in the input, then the input signal is passed through unchanged to the output.

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

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

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: Simulink / 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
- Discrete FIR Filter
- Integrator
- Second-Order Integrator
- Conditional susystem blocks such as Enabled Subsystem, Triggered Subsystem, and Function-Call Subsystem.
- 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 `[1xN]`.

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**.

Programmatic Use
Block Parameter: `StateOwnerBlock`
Type: character vector
Value: `''` | `'<model path/block name>'`
Default: `''`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

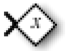
Event Listener | Initialize Function | Reset Function | State Writer | Terminate Function

Topics

“Customize Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”

State Writer

Write to a block state
Library: Simulink / 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
- Discrete FIR Filter
- Integrator
- Second-Order Integrator
- Conditional susystem blocks such as Enabled Subsystem, Triggered Subsystem, and Function-Call Subsystem.
- S-Function (with one data type work vector declared as a discrete-state vector)

Ports

Input

In — 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**.

Programmatic Use
Block Parameter: `StateOwnerBlock`
Type: character vector
Value: `''` | `'<model path/block name>'`
Default: `''`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

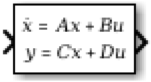
Event Listener | Initialize Function | Reset Function | State Reader | Terminate Function

Topics

“Customize Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”

State-Space

Implement linear state-space system
Library: Simulink / Continuous



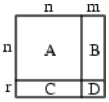
Description

The State-Space block implements a system whose behavior you define as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$
$$x\big|_{t=t_0} = x_0,$$

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.



$$\begin{array}{c|c|c|} & n & m \\ \hline n & A & B \\ \hline r & C & D \\ \hline \end{array}$$

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:

$$\dot{x} = Ax$$
$$y = Cx$$
$$x\big|_{t=t_0} = x_0.$$

Simulink software converts a matrix containing zeros to a sparse matrix for efficient multiplication.

Ports

Input

`Port_1` — Input signal

scalar | vector

Real-valued input vector of type `double`, where the width equals the number of columns in the **B** and **D** matrices. For more information, see “Description” on page 1-1872.

Data Types: `double`

Output

`Port_1` — Output vector

scalar | vector

Real-valued output vector of data type `double`, with width equal to the number of rows in the **C** and **D** matrices. For more information, see “Description” on page 1-1872.

Data Types: `double`

Parameters

`A` — Matrix coefficient, A

`1` (default) | scalar | vector | matrix

Specify the matrix coefficient `A`, as a real-valued *n*-by-*n* matrix, where *n* is the number of states. For more information on the matrix coefficients, see “Description” on page 1-1872.

Programmatic Use
Block Parameter: `A`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'1'`

`B` — Matrix coefficient, B

1 (default) | scalar | vector | matrix

Specify the matrix coefficient `B`, as a real-valued *n*-by-*m* matrix, where *n* is the number of states and *m* is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-1872.

Programmatic Use
Block Parameter: `B`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'1'`

`C` — Matrix coefficient, C

1 (default) | scalar | vector | matrix

Specify the matrix coefficient **C** as a real-valued *r*-by-*n* matrix, where *r* is the number of outputs and *n* is the number of states. For more information on the matrix coefficients, see “Description” on page 1-1872.

Programmatic Use
Block Parameter: `C`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'1'`

`D` — Matrix coefficient, D

1 (default) | scalar | vector | matrix

Specify the matrix coefficient **D** as a real-valued *r*-by-*m* matrix, where *r* is the number of outputs and *m* is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-1872.

Programmatic Use
Block Parameter: `D`
Type: character vector, string

Values: scalar | vector | matrix
Default: '1'

Initial conditions — Initial state vector

0 (default) | scalar | vector

Specify the initial state vector.

Limitations

The initial conditions of this block cannot be inf or NaN.

Programmatic Use

Block Parameter: X0
Type: character vector, string
Values: scalar | vector
Default: '0'

Absolute tolerance — Absolute tolerance for computing block states

auto (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify auto or -1.

- 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.
- 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.

Programmatic Use

Block Parameter: AbsoluteTolerance
Type: character vector, string
Values: 'auto' | '-1' | any positive real-valued scalar or vector
Default: 'auto'

State Name (e.g., 'position') — Assign unique name to each state

' ' (default) | 'position' | {'a', 'b', 'c'} | a | ...

Assign a unique name to each state. If this field is blank (`' '`), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'position'`.
- 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 in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

Limitations

- 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.

Programmatic Use

Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`

Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select **Analysis > Control Design > Model Discretizer**. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Discrete State-Space | Transfer Fcn

Topics

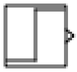
"States"

Introduced before R2006a

Step

Generate step function

Library: Simulink / 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.

Ports

Output

`Port_1` — Output step signal
scalar | vector

Output step function signal defined by the parameters **Step time**, **Initial value**, and **Final value**.

Data Types: `double`

Parameters

`Step time` — Time when step occurs

`1` (default) | scalar

Specify the time, in seconds, when the output jumps from the **Initial value** parameter to the **Final value** parameter.

Programmatic Use
Block Parameter: `Time`
Type: character vector
Values: scalar
Default: `'1'`

`Initial value` — Output value before step

`0` (default) | scalar

Specify the block output until the simulation time reaches the **Step time** parameter.

Programmatic Use
Block Parameter: `Before`
Type: character vector
Values: scalar
Default: `'0'`

`Final value` — Output value after step

`1` (default) | scalar

Specify the block output when the simulation time reaches and exceeds the **Step time** parameter.

Programmatic Use
Block Parameter: `After`
Type: character vector
Values: scalar
Default: `'1'`

`Sample time` — Sample rate

`0` (default) | scalar

Specify the sample rate of step. See “Specify Sample Time” for more information.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'0'`

`Interpret vector parameters as 1-D` — Treat vectors as 1-D

`on` (default) | `off`

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.

- When you select this check box, the block 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`.
- When you clear this check box, the block does not output a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector.

Programmatic Use

Block Parameter: `VectorParams1D`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use

Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

Block Characteristics

Data Types	`double`

Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code.

See Also

Ramp | Repeating Sequence Stair | Signal Builder

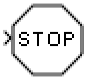
Topics

"Signal Basics"

Introduced before R2006a

Stop Simulation

Stop simulation when input is nonzero
Library: Simulink / 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* 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”.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Stop simulation when input signal is nonzero. This port accepts real signals of `double` or `Boolean` data types.

Data Types: `double` | `Boolean`

Block Characteristics

Data Types	`double` | `Boolean`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely varying execution times. While the code is functionally valid and acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Generated code stops executing when the stop condition is true.

HDL Code Generation

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

When you use this block in your model, HDL Coder™ does not generate HDL code for it. See Stop Simulation.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Topics

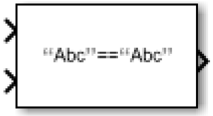
“Simulate a Model Interactively”

Introduced before R2006a

String Compare

Compare two input strings

Library: Simulink / String



Description

String Compare compares two strings. To see if two strings are identical, use this block. You can specify if the match is case sensitive and how much of the string to compare.

Ports

Input

`Port_1` — First string to compare
scalar

First string to compare, specified as a scalar.

Data Types: `string`

`Port_2` — Second string to compare
scalar

Second string to compare, specified as a scalar.

Data Types: `string`

Output

`Port_1` — True or false result
scalar

True or false result, specified as a scalar:

- `1` — Match.
- `0` — No match.

Data Types: `Boolean`

Parameters

`Case sensitive` — Case sensitivity for string comparison

on (default) | off

Case sensitivity for string comparison:

☑ on

Consider string case when comparing strings.

□ off

Do not consider string case when comparing strings.

`Compare Option` — Amount of characters to compare

`Entire string` (default) | `First N characters`

Amount of string to compare:

- `Entire string` — Compare both entire strings.
- `First N characters` — Compare the first *N* characters of both strings.

Dependencies

Setting this parameter to `First N characters` enables the **Number of characters** parameter.

`Number of characters` — Number of characters to compare

1 (default) | scalar

Number of characters to compare

Dependencies

This parameter is enabled when the **Compare Option** parameter is set to `First N characters`.

Data Types: `double`

Block Characteristics

Data Types	`Boolean` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

Topics

"Parse NMEA GPS Text Message"
"Simulink Strings"

Introduced in R2018a

String Concatenate

Concatenate input strings to form one output string

Library: Simulink / String

Description

The String Concatenate block concatenates multiple input strings, in order of their input, to form one output string. Use this block if you want to combine multiple strings into a single string.

Ports

Input

`Port_1` — First input string
scalar

First input string, specified as a scalar.

Data Types: `string`

`Port_2` — Second input string
scalar

Second input string, specified as a scalar.

Data Types: `string`

Output

`Port_1` — Concatenated string
scalar

Concatenated string, specified as a scalar.

Data Types: `string`

Parameters

`Number of Inputs` — Number of input strings
scalar

Number of input strings to concatenate, specified as a scalar. You can specify from 2 to 512 input ports.

`Output data type` — Output data type
`string` (default) | `scalar`

Output data type, specified using the string data type to specify a string with no maximum length.

To specify a string data type with a maximum length, specify `stringtype(N)`. For example, `stringtype(128)` creates a string data type with a maximum length of 128 characters.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

`Mode` — Category of data
`stringtype(128)` (default) | `scalar`

Use the `stringtype` function, for example, `stringtype(128)`.

Dependency

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

Block Characteristics

Data Types	`string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

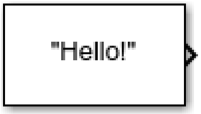
Topics

“Simulink Strings”

Introduced in R2018a

String Constant

Output specified string
Library: Simulink / String



Description

The String Constant block outputs a string specified by the **String** parameter. Use this block when you want a constant whose type is string.

Ports

Output

`Port_1` — Output string
scalar

Output string, specified as a scalar.

Data Types: `string`

Parameters

`String` — Input string
"Hello!" (default) | scalar

Input string, specified as a scalar.

`Output data type` — Output data type
`string` (default) | `scalar`

Output data type, specified using the string data type to specify a string with no maximum length.

To specify a string data type with a maximum length, specify `stringtype(`*`N`*`)`. For example, `stringtype(31)` creates a string data type with a maximum length of 31 characters.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Block Characteristics

Data Types	`string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

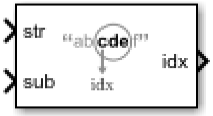
Topics

“Convert String to ASCII and Back to String”
“Simulink Strings”

Introduced in R2018a

String Find

Return index of first occurrence of pattern string
Library: Simulink / String



Description

The String Find block returns the index of the first occurrence of the pattern string **sub** in the text string **str**.

Ports

Input

`str` — String in which to find pattern
scalar

String in which to find pattern (`sub`), specified as a scalar.

Data Types: `string`

`sub` — Pattern
scalar

Pattern to be found in string (`str`), specified as a scalar.

Data Types: `string`

Output

`idx` — Position index of found pattern
scalar

Position index of the found pattern, specified as a positive integer scalar.

- If the block does not find the pattern, it returns `-1`.
- If the **sub** parameter is empty (`""`), the block returns `1`, indicating that it matched the beginning of the searched string.

Data Types: `int32`

Block Characteristics

Data Types	`base integer	string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

Topics

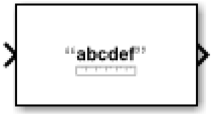
“Find Patterns in Strings”
“Simulink Strings”

Introduced in R2018a

String Length

Output number of characters in input string

Library: Simulink / String



Description

The String Length block outputs the number of characters in the input string. For example, you can use the String Length block to move focus of attention to a particular location in a string.

Ports

Input

`Port_1` — Input string
scalar

Input string, specified as a scalar.

Data Types: `string`

Output

`Port_1` — Number of characters
scalar

Number of characters in the input string, specified as a scalar.

Data Types: `uint32`

Block Characteristics

Data Types	`base integer	string`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

See Also

Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

Topics

"Get Text Following a Keyword"
"Simulink Strings"

Introduced in R2018a

String to ASCII

Convert string signal to uint8 vector
Library: Simulink / String

Description

The String To ASCII block converts a string signal to a uint8 vector. The block converts each character in the string to its corresponding ASCII value. For example, the block converts the input string `"Hello"` to `[72 101 108 108 111]`.

Ports

Input

`Port_1` — Input string signal
scalar

Input string signal, specified as a scalar.

Data Types: `string`

Output

`Port_1` — Converted uint8 vector signal
vector

Converted `uint8` vector signal of ASCII characters from input string signal, specified as a vector. The block converts each element in the string into its ASCII character equivalent and outputs the ASCII equivalents as a vector. If there are fewer characters than the maximum length, the block fills the remaining space with zeros at simulation. At code generation, the block fills the remaining space with null characters.

Data Types: `uint8`

Parameters

`Maximum length` — Maximum length of output vector
31 (default) | scalar

Maximum length of output string, specified as a scalar.

Block Characteristics

Data Types	`base integer` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To Enum | String to Double | String to Single | Substring | To String

Topics

“Convert String to ASCII and Back to String”

“Simulink Strings”

Introduced in R2018a

String to Double

Convert string signal to double signal
Library: Simulink / String

str
scan
"%lf"
lf

Description

Scan String scans an input string and converts it to signals per the format specified by the **Format** parameter. The block converts values to their decimal (base 10) representation and outputs the results as numeric or string signals. For example, if the **Format** parameter is set to `"%s is %f."`, the block outputs two parts, a string signal and a single signal. If the input is the string `"Pi is 3.14"`, the two outputs are `"Pi"` and `"3.14"`.

The Scan String, String to Double, and String to Single blocks are identical blocks. When configured for String to Double, the block converts the input string signal to a double numerical output. When configured for String to Single, the block converts the input string signal to a single numerical output.

For code generation, configure models that contain this block for non-finite number support by selecting the **Configuration Parameters** > **Code Generation** > **Interface** > **Support non-finite numbers** check box.

Ports

Input

`Port_1` — Input string
scalar

Input string, specified as a scalar.

Data Types: `string`

Output

d — Output data whose format matches %d format
scalar

Output data whose format matches specified format, defined as a scalar. Total maximum number of outputs is 128.

If the block cannot match an input string to a format operator specified in **Format**, it returns a warning and outputs an appropriate value (`0` or `""`) for each unmatched format operator.

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

`f` — Output data whose format matches %f format
scalar

Output data whose format matches the `%f` format, specified as a scalar. Total maximum number of outputs is 128.

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

`Port_N` — Output data whose format matches N format
scalar

Output data whose format matches *N* format, specified as a scalar. Total maximum number of outputs is 128.

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

Parameters

`Format` — Format operator for input
"%lf" (default) | scalar

Format operator for input, specified as a scalar. If the block cannot match the input string with the specified format, it returns `0`. The return of `0` differs from the `sscanf` function return, which is an empty matrix if the function cannot match the input with the specified format.

- For the String to Double block, this parameter has a default value of `%lf`.
- For the String to Single block, this parameter has a default value of `%f`.

For more information about acceptable format operators, see the Algorithms section.

Block Characteristics

Data Types	`double` | `single` | `base integer` | `string`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Algorithms

The Scan String block uses this format specifier prototype:

```
%[width][length]specifier
```

Numeric Fields

This table lists available conversion specifiers to convert text to numeric outputs. The block converts values to their decimal (base 10) representation.

Output Port Data Type	Conversion Specifier	Description
Integer, signed	%d	Base 10
Integer, unsigned	%u	Base 10

Output Port Data Type	Conversion Specifier	Description
Floating-point number	`%f`, `%e`, or `%g`	Floating-point values. Input fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`. Input fields that represents floating-point numbers can include leading `+` or `-` symbols and exponential notation using `e` or `E`. The conversion specifiers `%f`, `%e`, and `%g` all treat input fields the same way.

Character Fields

This table lists available conversion specifiers to convert text so that the output is a character array.

Character Field Type	Conversion Specifier	Description
String scalar	`%s`	Read the text until the block encounters whitespace.
	`%c`	Read any single character, including whitespace. To read multiple characters at a time, specify field width. For example, `%10c` reads 10 characters at a time.
Pattern-matching	`%[...]`	Read only characters in the brackets up to the first nonmatching character or whitespace. Example: `%[mus]` reads `'summer'` as `'summ'`.

Character Field Type	Conversion Specifier	Description
	%[^...]	Do not read characters in the brackets up to the first nonmatching character or whitespace. Example: %[m] reads 'summer' as 'su'.

Optional Operators

- Field Width — To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, %10s reads up to 10 characters at a time, including whitespace. %4f reads up to four digits at a time, including the decimal point.
- Literal Text to Ignore — This block must match the specified text immediately before or after the conversion specifier.

  Example: Hell%s reads "Hello!" as "o!".

Length Specifiers

The Scan String block supports the h and l length subspecifiers. These specifiers can change according to the **Configuration Parameters** > **Hardware Implementation** > **Number of bits** settings.

Length	i	u	f e g	s c
No length specifier	int	unsigned int	single	string
h	short	unsigned short	—	—
l	long	unsigned long	double	—

Notes for Specifiers that Specify Integer Data Types (d, u)

- Target int, long, short type sizes are controlled by settings in the **Configuration Parameters** > **Hardware Implementation** pane. For example, if the target int is 32 bits and the specifier is %u, then the expected input type will be uint32. For this example, the Scan String block requires that the output type be exactly `int32`. It cannot be any other data type.

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Single | Substring | To String | `sscanf`

Topics

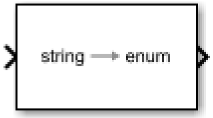
"Display and Extract Coordinate Data"
"Simulink Strings"

Introduced in R2018a

String to Enum

Input string signal to enumerated signal

Library: Simulink / String



Description

The String To Enum block converts the input string signal to an enumerated signal. To use this block, create an enumeration class in the current folder and use that class name in the **Output data type** parameter.

Ports

Input

`Port_1` — Input string signal
scalar

Input string signal, specified as a scalar.

Data Types: `string`

Output

`Output 1` — Enumerated number
scalar

Enumerated number associated with the input string, specified as a scalar.

Data Types: `enumerated`

Parameters

`Output data type` — Output data type

`SlDemoSign` (default) | `<data type expression>`

Use a data type object, for example, `Simulink.IntEnumType`.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. See "Specify Data Types Using Data Type Assistant" in the *Simulink User's Guide* for more information.

`Mode` — Category of data

`Enumerated` (default) | `<data type expression>`

Use a data type object, for example, `Simulink.IntEnumType`.

- `Enumerated` — Enumerated data class object.
- `<data type expression>` — Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

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

Block Characteristics

Data Types	`enumerated` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String to Double | String to Single | Substring | To String

Topics

"Convert String to Enumerated Data Type"
"Simulink Strings"

Introduced in R2018a

String to Single

Convert string signal to single signal
Library: Simulink / String

Description

Scan String scans an input string and converts it to signals per the format specified by the **Format** parameter. The block converts values to their decimal (base 10) representation and outputs the results as numeric or string signals. For example, if the **Format** parameter is set to `"%s is %f."`, the block outputs two parts, a string signal and a single signal. If the input is the string `"Pi is 3.14"`, the two outputs are `"Pi"` and `"3.14"`.

The Scan String, String to Double, and String to Single blocks are identical blocks. When configured for String to Double, the block converts the input string signal to a double numerical output. When configured for String to Single, the block converts the input string signal to a single numerical output.

For code generation, configure models that contain this block for non-finite number support by selecting the **Configuration Parameters** > **Code Generation** > **Interface** > **Support non-finite numbers** check box.

Ports

Input

`Port_1` — Input string
scalar

Input string, specified as a scalar.

Data Types: `string`

Output

d — Output data whose format matches %d format

scalar

Output data whose format matches specified format, defined as a scalar. Total maximum number of outputs is 128.

If the block cannot match an input string to a format operator specified in **Format**, it returns a warning and outputs an appropriate value (`0` or `""`) for each unmatched format operator.

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

f — Output data whose format matches %f format

scalar

Output data whose format matches the `%f` format, specified as a scalar. Total maximum number of outputs is 128.

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

`Port_N` — Output data whose format matches N format

scalar

Output data whose format matches *N* format, specified as a scalar. Total maximum number of outputs is 128.

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

Parameters

`Format` — Format operator for input

"%f" (default) | scalar

Format operator for input, specified as a scalar. If the block cannot match the input string with the specified format, it returns `0`. The return of `0` differs from the `sscanf` function return, which is an empty matrix if the function cannot match the input with the specified format.

- For the String to Double block, this parameter has a default value of `%lf`.
- For the String to Single block, this parameter has a default value of `%f`.

For more information about acceptable format operators, see the Algorithms section.

Block Characteristics

Data Types	`double` | `single` | `base integer` | `string`
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Algorithms

The Scan String block uses this format specifier prototype:

```
%[width][length]specifier
```

Numeric Fields

This table lists available conversion specifiers to convert text to numeric outputs. The block converts values to their decimal (base 10) representation.

Output Port Data Type	Conversion Specifier	Description
Integer, signed	%d	Base 10
Integer, unsigned	%u	Base 10

Output Port Data Type	Conversion Specifier	Description
Floating-point number	`%f`, `%e`, or `%g`	Floating-point values. Input fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`. Input fields that represents floating-point numbers can include leading `+` or `-` symbols and exponential notation using `e` or `E`. The conversion specifiers `%f`, `%e`, and `%g` all treat input fields the same way.

Character Fields

This table lists available conversion specifiers to convert text so that the output is a character array.

Character Field Type	Conversion Specifier	Description
String scalar	`%s`	Read the text until the block encounters whitespace.
	`%c`	Read any single character, including whitespace. To read multiple characters at a time, specify field width. For example, `%10c` reads 10 characters at a time.
Pattern-matching	`%[...]`	Read only characters in the brackets up to the first nonmatching character or whitespace. Example: `%[mus]` reads `'summer'` as `'summ'`.

Character Field Type	Conversion Specifier	Description
	`%[^...]`	Do not read characters in the brackets up to the first nonmatching character or whitespace. Example: `%[m]` reads `'summer'` as `'su'`.

Optional Operators

- Field Width — To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, `%10s` reads up to 10 characters at a time, including whitespace. `%4f` reads up to four digits at a time, including the decimal point.
- Literal Text to Ignore — This block must match the specified text immediately before or after the conversion specifier.

  Example: `Hell%s` reads `"Hello!"` as `"o!"`.

Length Specifiers

The Scan String block supports the `h` and `l` length subspecifiers. These specifiers can change according to the **Configuration Parameters** > **Hardware Implementation** > **Number of bits** settings.

Length	i	u	f e g	s c
No length specifier	`int`	`unsigned int`	`single`	`string`
`h`	`short`	`unsigned short`	—	—
`l`	`long`	`unsigned long`	`double`	—

Notes for Specifiers that Specify Integer Data Types (d, u)

- Target `int`, `long`, `short` type sizes are controlled by settings in the **Configuration Parameters** > **Hardware Implementation** pane. For example, if the target `int` is 32 bits and the specifier is `%u`, then the expected input type will be `uint32`. For this example, the Scan String block requires that the output type be exactly `int32`. It cannot be any other data type.

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | Substring | To String | `sscanf`

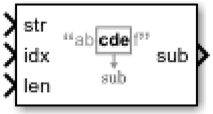
Topics

"Display and Extract Coordinate Data"
"Simulink Strings"

Introduced in R2018a

Substring

Extract substring from input string signal
Library: Simulink / String



Description

The Substring block extracts a substring from the input string signal. The block extracts the substring starting from the letter corresponding to **idx** and includes a **len** number of characters starting at **idx**. For example, if the input string is `"hello 123"`, input **idx** is `1`, and input **len** is `5`, the output is `"hello"`. The block extracts a substring starting at `1` and the next 4 characters for a total of 5 characters (`hello`).

Ports

Input

`str` — Input string signal
scalar

Input string signal, specified as a string.

Data Types: `string`

`idx` — Start of string to extract
scalar

Start of string to extract, specified as a positive scalar integer.

Data Types: `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`len` — Length of string to extract
scalar

Length of string to extract, specified as a scalar. If **len** causes the substring to extend beyond the end of the string, the output signal contains few than **len** characters.

Data Types: `uint8` | `uint16` | `uint32`

Output

`sub` — Extracted string

scalar

Extracted string, specified as a scalar.

Data Types: `string`

Parameters

`Inherit maximum length from input` — Use same maximum length as input string

`off` (default) | `on`

Use same maximum length as the input string source block.

☑ on

Use same maximum length. The substring includes the characters starting from the character at **idx** to the end of the string.

□ off

Do not use same maximum length.

`Output string from 'idx' to end` — Extract string from idx to end

`off` (default) | `on`

Extract string from `idx` to end of input string.

☑ on

Extract string from `idx` to end of input string.

□ off

Do not extract string from `idx` to end of input string.

Dependencies

Selecting this parameter removes the third input port.

Block Characteristics

Data Types	`base integer` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | To String

Topics

"Extract a String"
"Simulink Strings"

Introduced in R2018a

Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Group blocks to create model hierarchy
Library: Simulink / 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 and Models").
- Virtual subsystem – 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

☐ 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 Simulink Coder for code generation.
- 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"
- "Control Generation of Subsystem Functions" (Simulink Coder)
- "Generate Code and Executables for Individual Subsystems" (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 “Control Generation of Subsystem Functions” (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 “Control Generation of Subsystem Functions” (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 file name 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 “Control Generation of Subsystem Functions” (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 for Nonvirtual Subsystems" (Embedded Coder).
- For details on how to apply memory sections to atomic subsystems, see "Override Default Memory Placement for Subsystem Functions and Data" (Embedded Coder).

`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 for Nonvirtual Subsystems” (Embedded Coder)

`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” (Simulink Coder).
- 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 Memory Placement for Subsystem Functions and Data” (Embedded Coder).

`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” (Simulink Coder).

- 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 Memory Placement for Subsystem Functions and Data” (Embedded Coder).

`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 “Standard 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).
- 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 Memory Placement for Subsystem Functions and Data” (Embedded Coder).

`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 “Standard 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).
- 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 Memory Placement for Subsystem Functions and Data” (Embedded Coder).

`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 “Standard 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).
- 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 Memory Placement for Subsystem Functions and Data” (Embedded Coder).

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[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™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

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

AddSubtractSum of ElementsSum

Add or subtract inputs

Library: Simulink / Math Operations

Description

The Sum block performs addition or subtraction on its inputs. The Add, Subtract, Sum of Elements, and Sum blocks are identical blocks. This block can add or subtract scalar, vector, or matrix inputs. It can also collapse the elements of a signal and perform a summation.

You specify the operations of the block with the **List of signs** parameter with plus (+), minus (-), and spacer (|).

- The number of + and - characters equals the number of inputs. For example, +-+ requires three inputs. The block subtracts the second (middle) input from the first (top) input, and then adds the third (bottom) input.
- A spacer character creates extra space between ports on the block icon.
- If performing only addition, you can use a numerical value equal to the number of inputs.
- If only there is only one input port, a single + or - adds or subtracts the elements over all dimensions or in the specified dimension.

The Sum block first converts the input data type 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] ... + e[m]	`e[i]` is the ith element of input u
	The input port sign is –	y = 0.0 – e[0] – e[1] – e[2] ... – e[m]	
Two or more input ports	All input port signs are –	y = 0.0 – u[0] – u[1] – u[2] ... – u[n]	`u[i]` is the input to the ith input port
	The kth 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] ... (+/–) u[n]	

Ports

Inputs

The inputs can be of different data types, unless you select the **Require all inputs to have the same data type** parameter.

`Port_1` — First input operand signal

scalar | vector | matrix

Input signal to the addition or subtraction operation. If there is only one input signal, then addition or subtraction is performed on the elements over all dimensions or the specified dimension.

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

`Port_n` — nth input operand signal

scalar | vector | matrix

`n`th input signal to the operations. The number of inputs matches the number of signs in the **List of signs** parameter. The block applies the operations to the inputs in the order listed. You can also use a numerical value equal to the number of input ports as the **List of signs** parameter. The block creates the input ports and applies addition to all inputs. For example, if you assign 5 for the **List of signs** parameter, the block creates 5 input ports and adds them together to produce the output.

All nonscalar inputs must have the same dimensions. Scalar inputs are expanded to have the same dimensions as other inputs.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output signal resulting from addition and/or subtraction operations. The output signal has the same dimension as the input signals.

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

Parameters

Main

`Icon shape` — Block icon shape

rectangular (default) | **round**

Designate the icon shape of the block as rectangular or round.

For a rectangular block, the first input port is the top port. 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.

Programmatic Use
Block Parameter: `IconShape`
Type: character vector
Values: `'rectangular'` | `'round'`

Default: `'on'`

`List of signs` — Operations performed on inputs

++ (default) | + | - | | | `integer`

Enter addition and subtraction operations performed on the inputs. An input port is created for each operation. A spacer (`|`) creates extra space between the input ports on the block icon. Addition is the default operation. If you only want to add the inputs, enter the number of input ports. The operations are performed in the order listed.

When you enter only one element, the block enables the **Sum over** parameter. For a single vector input, `+` or `-` adds or subtracts the elements over all dimensions or in the specified dimension.

Tips

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.

Programmatic Use

Block Parameter: `Inputs`
Type: character vector
Values: `'+'` | `'-'` | `|` | integer
Default: `'++'`

`Sum over` — Dimensions for operations on a single vector input

All dimensions (default) | **Specified dimension**

Select the dimension over which the block performs the sum-over operation.

For **All dimensions**, all input elements are summed. When you select configuration parameter **Use algorithms optimized for row-major array layout**, Simulink enables row-major algorithms for simulation. To generate row-major code, set configuration parameter **Array layout** (Simulink Coder) to `Row-major` in addition to selecting **Use algorithms optimized for row-major array layout**. The column-major and row-major algorithms differ only in the summation order. In some cases, due to different operation order on the same data set, you might experience minor numeric differences in the outputs of column-major and row-major algorithms.

When you select **Specified dimensions**, another parameter **Dimension** appears. Choose the specific dimension for summing the vector input.

Dependency

Enabled when you list only one sign in the **List of signs** parameter.

Programmatic Use

Block Parameter: `CollapseMode`
Type: character vector
Values: `'All dimensions'` | `'Specified dimension'`
Default: `'All dimensions'`

`Dimension` — Dimension for summation on vector input

`1` (default) | `integer`

When you choose **Specified dimension** for the **Sum over** parameter, specify the dimension over which to perform the operation.

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.

Dependency

Enabled when you choose `Specified dimension` for the **Sum over** parameter.

Programmatic Use

Block Parameter: `CollapseDim`
Type: character vector
Value: `integer`
Default: `'1'`

Sample time — Specify sample time as a value other than -1

-1 (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use

Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '-1'

Signal Attributes

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Require all inputs to have the same data type — Require that all inputs have the same data type

off (default) | on

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

Programmatic Use

Block Parameter: InputSameDT
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Accumulator data type — Data type of the accumulator

Inherit: Inherit via internal rule (default) | Inherit: Same as first input | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Choose the data type of the accumulator. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. When you choose `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.

Programmatic Use
Block Parameter: `AccumDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule'` | `'Inherit: Same as first input'` | `'double'` `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'`, `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Output minimum` — Minimum output value for range checking

`[ ]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'` | scalar

Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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" (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`Inherit: Inherit via internal rule` (default) | `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)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `<data type expression>`

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

When you select an inherited option, the block behaves as follows:

- `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 the first input signal.
- `Inherit: Inherit same as accumulator`— Use data type of the accumulator.

Programmatic Use

Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'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)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select to lock 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).

Programmatic Use
Block Parameter: `LockScale`
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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 (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
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™.

For more information on HDL code generation, see Add.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Bias | Divide | Gain

Topics

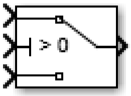
"Control Signal Data Types"

Introduced before R2006a

Switch

Combine multiple signals into single signal

Library: Simulink / Commonly Used Blocks
Simulink / Signal Routing



Description

The Switch block passes through the first input or the third input signal based on the value of the second input. The first and third inputs are data input. The second input is a control input. Specify the condition under which the block passes the first input by using the **Criteria for passing first input** and **Threshold** parameters.

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, set the **Threshold** parameter to a scalar value.

Limitations

- 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.

Ports

Input

`Port_1` — First data input signal

scalar | vector

First of two data inputs. The block propagates either the first or second data input to the output. The block selects which input to pass based on the control input. Specify the condition for the control input to pass the first input using the **Criteria for passing first input** and **Threshold** parameters.

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

`Port_2` — Control input signal

scalar | vector

Control signal the block uses to determine whether to pass the first or second data input to the output. If the control input meets the condition set in the **Criteria for passing first input** parameter, then the block passes the first data input. Otherwise, the block passes the second data input.

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

`Port_3` — Second data input signal

scalar | vector

Second of two data inputs. The block propagates either the first or second data input to the output. The block selects which input to pass based on the control input. Specify the condition for the control input to pass the first or second input using the **Criteria for passing first input** and **Threshold** parameters.

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

Output

Port_1 — Output signal

scalar | vector

Output signal propagated from either the first or second input signal, based on the control signal value.

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

Parameters

Main

Criteria for passing first input — Selection criteria to pass first data input

u2 >= Threshold (default) | u2 > Threshold | u2 ~= 0

Select the condition under which the block passes the first data 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 second data input signal from input Port_3.

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.

Tip

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.

Programmatic Use

Block Parameter: `Criteria`
Type: character vector
Value: `'u2 >= Threshold'` | `'u2 > Threshold'` | `'u2 ~=0'`
Default: `'u2 >= Threshold'`

`Threshold` — Threshold used in criteria

`0` (default) | `scalar`

Assign the threshold used in the **Criteria for passing first input** that determines which input the block passes to the output. **Threshold** must be greater than **Output minimum** and less than **Output maximum**.

To specify a nonscalar threshold, use brackets. For example, the following entries are valid:

- `[1 4 8 12]`
- `[MyColors.Red, MyColors.Blue]`

Dependencies

Setting **Criteria for passing first input** to `u2 ~=0` disables this parameter.

Programmatic Use

Block Parameter: `Threshold`
Type: character vector
Value: `'off' | 'on'`
Default: `'off'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | Boolean

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

Programmatic Use
Block Parameter: `ZeroCross`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

Signal Attributes

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Require all data port inputs to have the same data type` — Require data ports to have the same data type

`off` (default) | `on`

Require all data inputs to have the same data type.

Programmatic Use
Block Parameter: `InputSameDT`
Type: character vector
Value:
Default: `'0'`

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

`[ ]` (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector
Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class name>` | `<data type expression>`

Specify the output data type.

`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`.

Tip

When the output is of enumerated type, both data inputs should use the same enumerated type as the output.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Inherit via back propagation'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'`, `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `Enum: <class name>` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see "Lock the Output Data Type Setting" (Fixed-Point Designer).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Integer rounding mode` — Specify the rounding mode for fixed-point operations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Choose one of these rounding modes.

`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.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Default: `'Floor'`

See Also

For more information, see "Rounding" (Fixed-Point Designer).

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `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.
- `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.

Tip

- 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Allow different data input sizes` — Allow different data input sizes

`off` (default) | `on`

Select this check box to allow input signals with different sizes. The block 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.

Programmatic Use
Block Parameter: `AllowDiffInputSizes`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (string.h) under certain conditions.

HDL Code Generation

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

For information about HDL Code Generation, see Switch.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

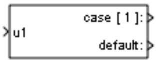
See Also

Manual Switch | Multiport Switch

Introduced before R2006a

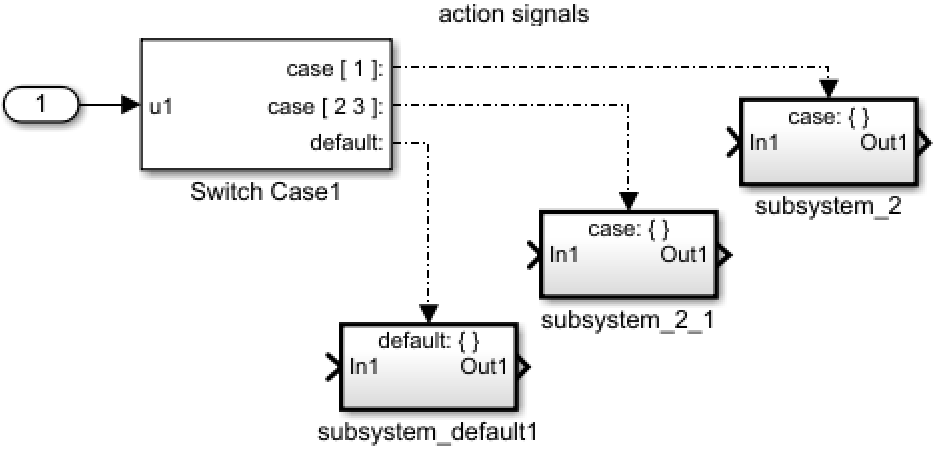
Switch Case

Select subsystem execution using logic similar to a switch statement
Library: Simulink / Ports & Subsystems



Description

The Switch Case block with Switch Case Action Subsystem blocks containing Action Port blocks, implements switch logic to control subsystem execution.



A Switch Case block has a single input. To select a case, define the input value using the **Case conditions** parameter. The cases are evaluated top down starting with the first case.

Each case is associated with an output port that is attached to a Switch Case Action Subsystem block. When a case is selected, the associated output port sends an action signal to execute the subsystem.

A `default` case is selected after all of the other case conditions evaluate to false. Providing a `default` case is optional, even if the other case conditions do not exhaust every possible input value.

Cases for the Switch Case block contain an implied break after a Switch Case Action Subsystem block is executed. Therefore, there is no fall through behavior for the Simulink Switch Case block as found in standard C `switch` statements.

Ports

Input

`u1 (logical operator)` — Value for case selection

scalar

Input to the port labeled **u1** of a Switch Case block can be:

- A scalar value with a built-in data type that Simulink supports. However, the Switch Case block does not support Boolean or fixed-point data types, and it truncates numeric inputs to 32-bit signed integers.
- A scalar value of any enumerated data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `enumerated`

Output

`case` — Action signal for a Switch Case Action Subsystem block

scalar

Output from the **Case** and **default** ports are action signals connected to Switch Case Action Subsystem blocks.

Parameters

`Case conditions` — Specify case values

`{1}` (default) | list of cases

Specify the cases values using MATLAB cell notation.

{1}

Specify the output port labeled `case[1]`outputs an action signal when the input port value is `1`.

list of ports with case assignments

Specify multiple cases and ports 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 from a long list of case conditions are displayed in shortened form on the face of the Switch Case block, using a terminating ellipsis (...).

You can use the `enumeration` function to specify case conditions that include a case for every value in an enumerated type.

Programmatic Use
Block Parameter: `CaseConditions`
Type: character vector
Values: `'{1}'` | `'<list of cases>'`
Default: `'{1}'`

`Show default case` — Control display of default output port

off (default) | on

Control display of default output port.

off

Hide default output port.

on

Display default output port as the last case on the Switch Case block. This allows you to specify a default case that executes when the input value does not match any of the other case values.

Programmatic Use
Block Parameter: `ShowDefaultCase`

Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`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
Values: `'on'` | `'off'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `enumerated`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`Yes`

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks

Action Port | Subsystem | Switch Case Action Subsystem

Topics

Select Subsystem Execution

Introduced before R2006a

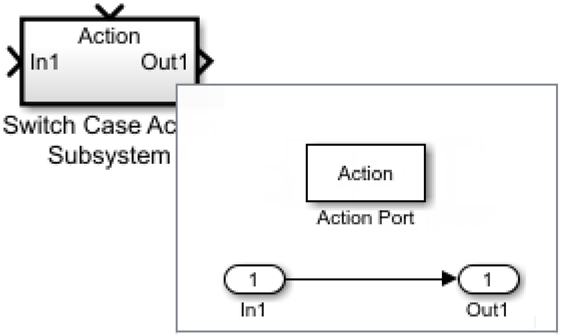
Switch Case Action Subsystem

Subsystem whose execution is enabled by a Switch Case block

Library: Simulink / Ports & Subsystems

Description

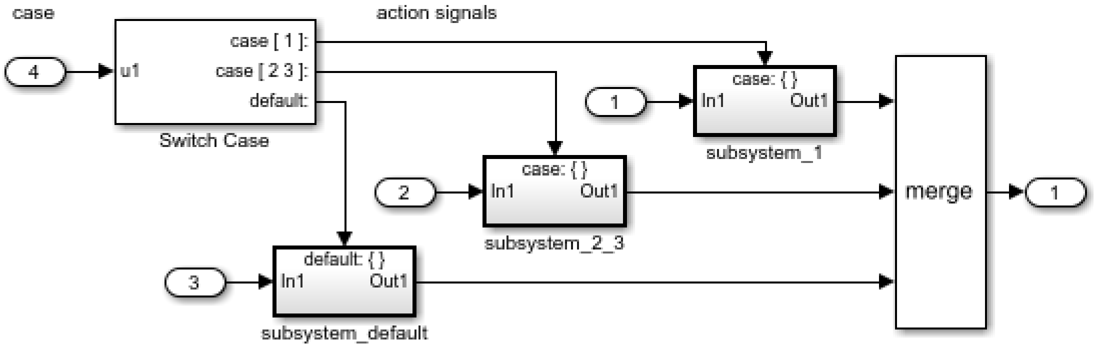
The Switch Case Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is controlled by a Switch Case block. The input port to a Switch Case block selects a case defined using the **Case conditions** parameter. Depending on input value and case selected, an action signal is sent to execute a Switch Case Action Subsystem 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 sample time parameter to be either inherited (`-1`) or the same value as the Switch Case block sample time.

Merge signals from Switch Case Subsystem blocks

This example shows how to create a one signal from multiple subsystem output signals. The Switch Case block selects the execution of one If Action Subsystem block from a set of subsystems. Regardless of which subsystem the Switch Case block selects, you can create a one resulting signal with a Merge block.



Ports

Input

`In1` — 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. An action signal is a control signal connected to the action port of an Switch Case Action Subsystem block. A message on the action signal initiates executition of the subsystem.

Data Types: `action`

Output

`Out1` — 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`

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[a] | `string`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`Yes`[a]
Zero-Crossing Detection	`No`

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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks
Action Port | Subsystem | Switch Case

Topics

Select Subsystem Execution

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

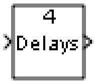
See Also

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: Simulink / 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 backpropagation. Each delay is equivalent to the z^{-1} discrete-time operator, which the Unit Delay block represents.

Ports

Input

`Port_1` — Input signal

scalar

Input signal to delay, specified as a scalar.

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

Output

`Port_1` — Delayed versions of input signal

scalar | vector

Output signal is all delayed versions of the input signal. Use the **Order output vector starting with** parameter to specify the order of delayed signals in the output vector.

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

Parameters

`Initial condition` — Initial output

`0.0` (default) | scalar | vector | matrix

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.

Limitations

The initial condition of this block cannot be `inf` or `NaN`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar | vector | matrix
Default: `'0.0'`

`Sample time` — Time between samples

`-1` (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `samptime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

`Number of delays` — Number of discrete-time operators

`4` (default) | positive scalar | `-1` (for inherited)

Specify the number of discrete-time operators as a positive scalar, or `-1` for inherited.

Programmatic Use
Block Parameter: `NumDelays`
Type: character vector
Values: positive scalar | `-1` (inherited)
Default: `'4'`

`Order output vector starting with` — Order of output

`Oldest` (default) | `Newest`

Specify whether to output the oldest delay version first, or the newest delay version first.

Programmatic Use
Block Parameter: `DelayOrder`
Type: character vector
Values: `'Oldest' | 'Newest'`
Default: `'Oldest'`

`Include current input in output vector` — Include current input in output vector

`off` (default) | `on`

Select this check box to include the current input in the output vector.

Programmatic Use
Block Parameter: `includeCurrent`
Type: character vector

Values: `'off'` | `'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
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™.

For more information about HDL code generation, see Tapped Delay.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Delay | Resettable Delay | Unit Delay | Variable Integer Delay

Topics

"Specify Sample Time"

Introduced before R2006a

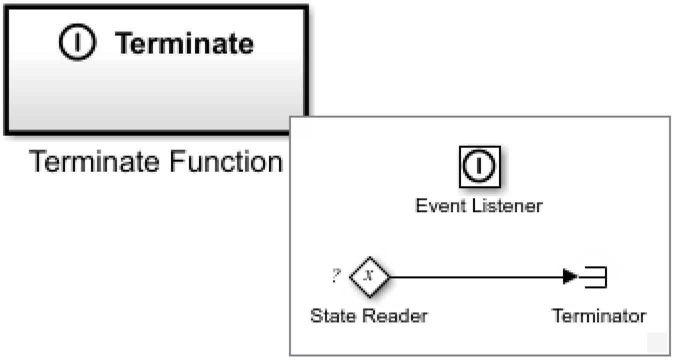
Terminate Function

Execute contents on a model terminate event

Library: Simulink / 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.

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	No
Zero-Crossing Detection	No

[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™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Event Listener | Initialize Function | Reset Function | State Reader | State Writer

Topics

"Customize Initialize, Reset, and Terminate Functions"
"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: Simulink / Commonly Used Blocks
Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix | n-D array | bus

Use this port to direct signals from output ports that are otherwise unconnected during a simulation. The port accepts real or complex signals of all data types.

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

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus` | `string`
Direct Feedthrough	`No`

Multidimensional Signals	Yes
Variable-Size Signals	No
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™.

When you use this block in your model, HDL Coder does not generate code for it.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Ground | “Unconnected block output ports”

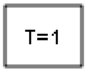
Topics

“Model Configuration Parameters: Connectivity Diagnostics”
“Systematic Diagnosis of Errors and Warnings”

Introduced before R2006a

Timed-Based Linearization

Generate linear models in base workspace at specific times
Library: Simulink / 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
`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

Field	Description
`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

Tip To generate models conditionally, use the Trigger-Based Linearization block.

Parameters

`Linearization time` — Time at which to generate a linear model

`1` (default) | scalar | vector

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.

Programmatic Use
Block Parameter: `LinearizationTime`
Type: character vector
Values: scalar | vector
Default: `'1'`

`Sample time (of linearized model)` — Sample time

`0` (default) | scalar | vector

Specify a sample time to create discrete-time linearizations of the model (see “Discrete-Time System Linearization” on page 2-54).

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'0'`

Block Characteristics

Data Types	
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

See Also

Trigger-Based Linearization | `dlinmod` | `linmod`

Topics

“Discrete-Time System Linearization” on page 2-54

Introduced in R2010a

To File

Write data to file
Library: Simulink / Sinks

Description

The To File block writes input signal data into a MAT-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 you pause the simulation or the simulation completes. If simulation terminates abnormally, the To File block saves the data it has logged up until the point of the abnormal termination.

The To File block icon shows the name of the output file.

Control Amount of Data Saved

If you specify data logging intervals with the **Configuration Parameters > Data Import/Export > Logging intervals** parameter, the To File block logs only data inside of the intervals. For example, the block logs no data if the intervals are empty (`[]`). The block stores the logged data in the file associated with the block instead of in the variable that you specify for the **Single simulation output** parameter.

For variable-step solvers, to control the amount of data available to the To File block, use the **Configuration Parameters > Data Import/Export > Additional parameters > Output options** parameter. For example, to write data at identical time points over multiple simulations, select the `Produce specified output only` option.

Block parameters also control the amount of data saved. See “Decimation” on page 1-0 and “Sample time” on page 1-0 .

Pause 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.

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

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 "Model Reuse" for more information.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix | N-D array

Signal to store in file. Each sample consists of a timestamp and an associated data value. The data can be in array format or MATLAB `timeseries` format. The To File block accepts real or complex signal data of any data type that Simulink software supports, except fixed-point data with a word length greater than 32 bits.

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

Parameters

`File name` — Path or file name
`untitled.mat` (default) | MAT-file path or name

Specify the path or file name of the MAT-file in which to store the output. On UNIX systems, the path name can start with a tilde (~) character signifying your home folder. If you specify a file name without path information, Simulink software stores the file in the

MATLAB working folder. (To determine the working folder, at the MATLAB command line, enter pwd.) If the file exists, Simulink software overwrites it.

Programmatic Use
Block Parameter: `FileName`
Type: character vector
Values: MAT-file path or name
Default: `'untitled.mat'`

`Variable name` — Matrix name

`ans` (default) | character vector

Specify the name of the matrix contained in the file.

Programmatic Use
Block Parameter: `MatrixName`
Type: character vector
Values: character vector
Default: `'ans'`

`Save format` — Data format

`Timeseries` (default) | `Array`

Specify the data format that the To File block uses for writing data.

Use the `Array` format only for vector, double, noncomplex signals.

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 virtual and nonvirtual 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 software writes one column to the matrix for each data sample. The first element of the column contains the timestamp. The remainder of the column contains data for the corresponding output values.

- Supports writing data that is one-dimensional, double, and noncomplex.

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.

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.

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

Programmatic Use
Block Parameter: `SaveFormat`
Type: character vector

Values: `'Timeseries' | 'Array'`
Default: `'Timeseries'`

`Decimation` — Decimation factor that determines when data writes

1 (default) | scalar | vector

Specify the decimation factor, *n*, that writes data at every *n*th time that the block executes. The default value has this block writing data at every time step.

Programmatic Use
Block Parameter: `Decimation`
Type: character vector
Values: scalar | vector
Default: `'1'`

`Sample time` — Sample period and offset

-1 (default) | scalar | vector

Specifies the sample period and offset at which to collect data points. This parameter is useful when you are using a variable-step solver where the interval between time steps is not constant. The default value causes the block to inherit the sample time from the driving block. See "Specify Sample Time".

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`[a] | `enumerated` | `bus`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`

Variable-Size Signals	No
Zero-Crossing Detection	No

a. Supports up to 32-bit fixed-point data types.

Tips

- If MATLAB encounters memory issues when you log many 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”.
- 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.

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Code generation for RSim target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.

To generate code for a To File block, on the **Code Generation** > **Interface** pane, select the configuration parameter “MAT-file logging” (Simulink Coder).

HDL Code Generation

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

When you use this block in your model, HDL Coder™ does not generate HDL code for it. See To File.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Supports up to 32-bit fixed-point data types.

See Also

From File | From Workspace | To Workspace

Topics

“Save Runtime Data from Simulation”
“Convert Logged Data to Dataset Format”
“Limit Amount of Exported Data”

Introduced before R2006a

To Workspace

Write data to workspace

Library: Simulink / Sinks

Description

The To Workspace block writes input signal data to a workspace. During simulation, the block writes data to an internal buffer. When you pause the simulation or the simulation completes, that data is written to the workspace. Data is not available until the simulation pauses or stops.

The To Workspace block typically writes data to the MATLAB base workspace. For a `sim` command in a MATLAB function, the To Workspace block sends data 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.

```
function myfunc
    a = sim('mTest','SimulationMode','normal');
    b = a.get('simout')
    assignin('base','b',b);
end
```

The To Workspace block icon shows the name of the variable to which the data is written.

Control Amount of Data Saved

If you specify data logging intervals with the **Configuration Parameters > Data Import/Export > Logging intervals** parameter, the To Workspace block does not log data outside of the intervals. For example, the block logs no data if the intervals are empty (`[]`). The block stores the logged data in the variable that you specify for the **Single simulation output** parameter.

For variable-step solvers, to control the amount of data available to the To Workspace block, use the **Configuration Parameters > Data Import/Export > Additional parameters > Output options** parameter. For example, to write data at identical time points over multiple simulations, select the `Produce specified output only` option.

Block parameters also control the amount of data saved. See “Limit data points to last” on page 1-0 , “Decimation” on page 1-0 , and “Sample time” on page 1-0 .

Log to MAT-File

When you enable the **MAT-file logging** parameter in **Configuration Parameters**, the To Workspace block logs its data to a MAT-file. For information about this parameter, see “MAT-file logging” (Simulink Coder).

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix | N-D array

Workspace data created from input signal. The To Workspace block can save real or complex inputs of any data type that Simulink supports, including fixed-point and enumerated data types, and bus objects.

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

Parameters

`Variable name` — Name of variable for saved data
`simout` (default) | character vector

Specify the name of the variable for the saved data.

Programmatic Use
Block Parameter: `VariableName`
Type: character vector
Values: character vector
Default: `'simout'`

`Limit data points to last` — Maximum number of input samples to save

`inf` (default) | scalar | vector

Specify the maximum number of input samples to save. If the simulation generates more data points than the specified maximum, the simulation saves only the most recently generated samples. The default value causes the block to write all data.

Programmatic Use
Block Parameter: `MaxDataPoints`
Type: character vector
Values: scalar | vector
Default: `'inf'`

`Decimation` — Decimation factor that determines when data writes

`1` (default) | scalar | vector

Specify the decimation factor, *n*, which writes data at every *n*th time that the block executes. The default value causes the block to write data at every time step.

Programmatic Use
Block Parameter: `Decimation`
Type: character vector
Values: scalar | vector
Default: `'1'`

`Save format` — Format for saving simulation output

`Timeseries` (default) | `Structure With Time` | `Structure` | `Array`

Specify the format for saving simulation output to the workspace.

The default `Timeseries` format saves nonbus signals as a MATLAB `timeseries` object and bus signals as a structure of MATLAB `timeseries` objects.

The `Array` format saves 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 signal is a scalar or a vector, each input sample is output as a row of the array. 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. Suppose again that `simout` is the name of the resulting workspace array. Then, `simout(:,:,1)` is the input signal value at the first sample point, `simout(:,:,2)` is the input signal value at the second sample point, and so on.

The `Structure` format consists of a structure with three fields:

- `time` — Empty field for this format.
- `signals` — 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.

The `Structure With Time` format is the same as `Structure`, except that the time field contains a vector of simulation time hits.

If you select `Array` or `Structure`, the **Save 2-D signals as** parameter appears.

To read the To Workspace block output directly with a From Workspace block, use either the `Timeseries` or `Structure with Time` format. The From Workspace block can read sample-based data from a To Workspace block that was saved in a previous simulation. For details, see “Comparison of Signal Loading Techniques”.

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

Simulation Mode	`Timeseries`	`Array`, `Structure`, or `Structure With Time`
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

Programmatic Use
Block Parameter: `SaveFormat`
Type: character vector
Values: `'Timeseries'` | `'Structure with Time'` | `'Structure'` | `'Array'`
Default: `'Timeseries'`

`Save 2-D signal as` — Format for saving 2-D signals

`3-D array (concatenate along third dimension)` (default) | `2-D array (concatenate along first dimension)` | `Inherit from input (this choice will be removed - see release notes)`

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 2-by-4-by-10 array.

- `2-D array (concatenate along first dimension)`

  This setting is well-suited for frame-based signals. Data is concatenated along the first dimension. For example, 2-by-4 matrix input for 10 samples is stored as a 20-by-4 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)`	*K*-by-*N* matrix. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of samples acquired in each column by the end of simulation. This setting is equivalent to multiplying the input frame size (*M*) by the total number of *M*-by-*N* inputs acquired by the block.
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 `inf`, *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

Input Signal Dimensions	Save 2-D Signals As ...	Signal to Workspace Output Dimension
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 `inf`, *K* represents the total number of *M*-by-*N* inputs acquired by the end of simulation

Dependencies

To enable the **Save 2-D signals as** parameter, set the **Save format** to `Array` or `Structure`.

Programmatic Use
Block Parameter: `Save2DSignal`
Type: character vector
Values: `'2-D array (concatenate along first dimension)'` | `'3-D array (concatenate along third dimension)'` | `'Inherit from input (this choice will be removed - see release notes)'`
Default: `'3-D array (concatenate along third dimension)'`

`Log fixed-point data as a fi object` — Log fixed-point data as a fi object or double

`off` (default) | `on`

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, Simulink software logs fixed-point data to the workspace as `double`.

Programmatic Use
Block Parameter: `FixptAsFi`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Sample time` — Sample period and offset

-1 (default) | scalar | vector

Specifies the sample period and offset at which to collect data points. This parameter is useful when you are using a variable-step solver where the interval between time steps is not constant. The default value causes the block to inherit the sample time from the driving block. See “Specify Sample Time”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Block Characteristics

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

Tips

To make post-processing easier, you can convert data saved by this block to `Dataset` format. This conversion is useful when post processing this data with other logged data that can use `Dataset` format (for example, logged states). See “Convert Logged Data to Dataset Format”. You can also use signal logging with a variable-size signal exception.

Extended Capabilities

HDL Code Generation

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

When you use this block in your model, HDL Coder™ does not generate HDL code for it. See To Workspace.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

From File | From Workspace | To File

Topics

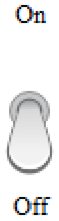
"Export Simulation Data"
"Limit Amount of Exported Data"

Introduced before R2006a

Toggle Switch

Toggle parameter between two values

Library: Simulink / Dashboard



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.

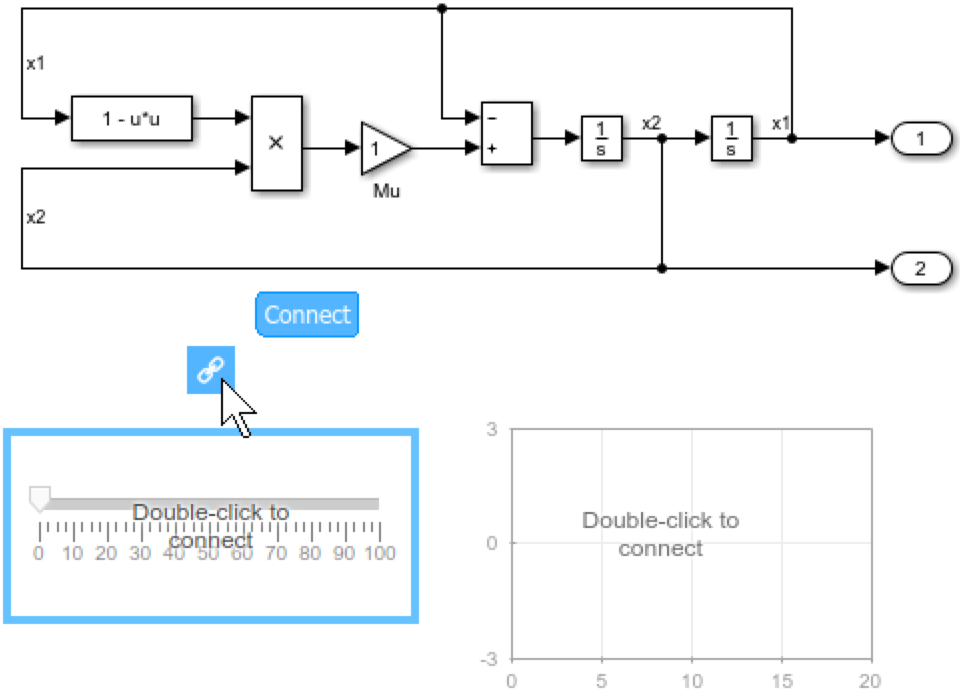
Double-clicking the Toggle Switch block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connecting Dashboard Blocks

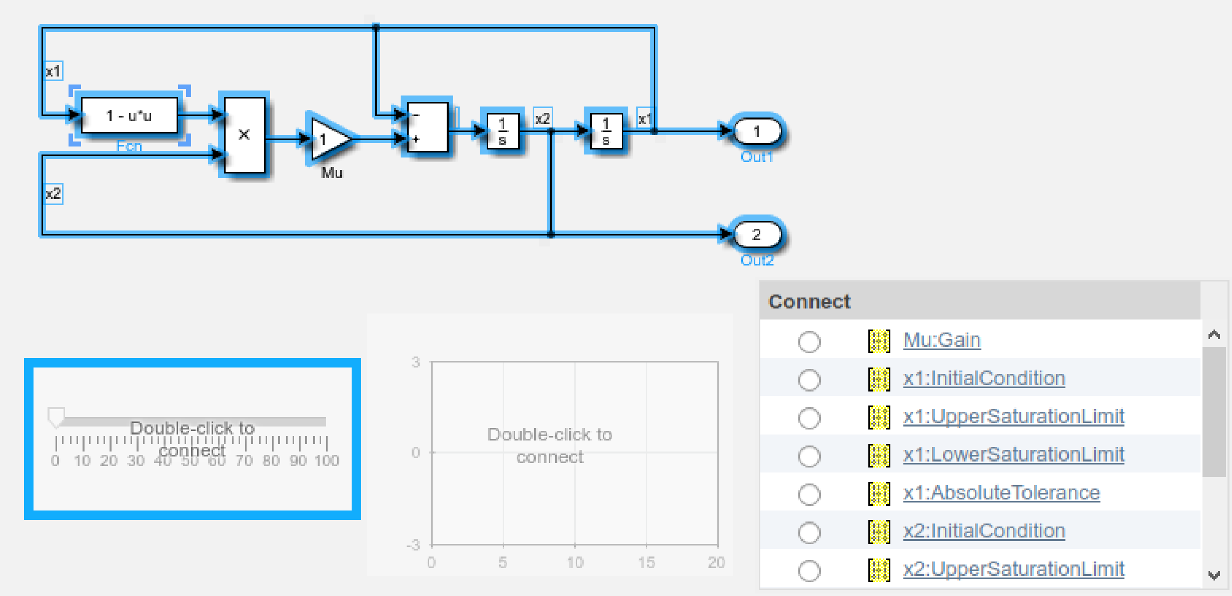
Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.



In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.



Connect

○ Mu:Gain
○ x1:InitialCondition
○ x1:UpperSaturationLimit
○ x1:LowerSaturationLimit
○ x1:AbsoluteTolerance
○ x2:InitialCondition
○ x2:UpperSaturationLimit

When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using

connect mode, the Dashboard block does not display the connected value until the you uncomment the block.

- Dashboard blocks cannot connect to signals inside referenced models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` 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 variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio button next to the variable or parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram using **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.

See Also

Rocker Switch | Rotary Switch | Slider Switch

Topics

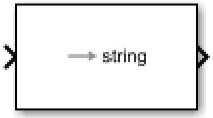
"Tune and Visualize Your Model with Dashboard Blocks"
"Decide How to Visualize Simulation Data"

Introduced in R2015a

To String

Convert input signal to string signal

Library: Simulink / String



Description

The To String block creates a string signal from an input signal. For example, consider using this signal to convert a logical value `1` or `0` to its string equivalent `"false"` or `"true"`.

When a MinGW compiler compiles code generated from the block, running the compiled code may produce nonstandard results for floating-point inputs. For example, a numeric input of `501.987` returns the string `"5.019870e+002"` instead of the expected string `"5.019870e+02"`.

Ports

Input

`Port_1` — Input signal
scalar

Input signal, specified as a scalar.

Output

`Port_1` — Output string
scalar

Output string, specified as a scalar. This block returns the output as a string, surrounded by double quotes.

- If the input is a Boolean, the output is a logical value (`1` or `0`) and the block returns its textual equivalent (`true` or `false`).
- If the input is a numeric data type, such as an integer, single, double, or fixed point, the block returns the number as a string. For example, an input of `1` converts to `"1"` and an input of `0` converts to `"0"`.

Note The output string might not contain all the digits of the numeric value from the input port.

Data Types: `string`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `string`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring

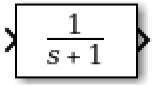
Topics

"Get Text Following a Keyword"
"Simulink Strings"

Introduced in R2018a

Transfer Fcn

Model linear system by transfer function
Library: Simulink / 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

  $$H(s)=\frac{y(s)}{u(s)}=\frac{num(s)}{den(s)}=\frac{num(1)s^{nn-1}+num(2)s^{nn-2}+\ldots+num(nn)}{den(1)s^{nd-1}+den(2)s^{nd-2}+\ldots+den(nd)},$$

  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.

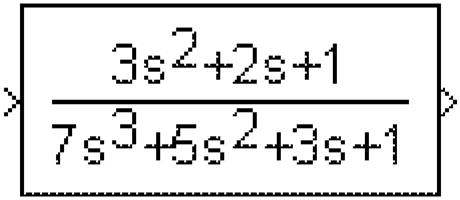
For more information, type `help tf2ss` or see the Control System Toolbox™ documentation.

Transfer Function Display on the Block

The Transfer Fcn block displays the transfer function depending on how you specify the numerator and denominator parameters.

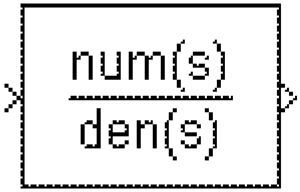
- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified coefficients and powers of *s*. If you specify a variable in parentheses, the block evaluates the variable.

  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:

- 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:

Ports

Input

`Port_1` — Input signal

scalar

Input signal, specified as a scalar with data type `double`.

Data Types: `double`

Output

`Port_1` — Output signal

scalar | vector

Output signal, provided as a scalar or vector with data type `double`.

- For a single-output system, the input and output of the block are scalar time-domain signals.
- For a multiple-output system, the input is a scalar, and the output is a vector, where each element is an output of the system.

Data Types: `double`

Parameters

`Numerator coefficients` — Vector or matrix of numerator coefficients

`[1]` (default) | vector | matrix

Define the numerator coefficients of the transfer function.

- 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.

Programmatic Use
Block Parameter: `Numerator`
Type: character vector, string
Values: vector | matrix
Default: `'[1]'`

`Denominator coefficients` — Row vector of denominator coefficients

`[1 1]` (default) | vector

Define the row vector of denominator coefficients.

- 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.

Programmatic Use
Block Parameter: `Denominator`
Type: character vector, string
Values: vector
Default: `'[1 1]'`

`Absolute tolerance` — Absolute tolerance for computing block states

`auto` (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify `auto` or `-1`.

- 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.
- 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.

Programmatic Use
Block Parameter: `AbsoluteTolerance`
Type: character vector, string
Values: `'auto'` | `'-1'` | any positive real-valued scalar or vector
Default: `'auto'`

`State Name (e.g., 'position')` — Assign unique name to each state

`' '` (default) | `'position'` | `{'a', 'b', 'c'}` | `a` | ...

Assign a unique name to each state. If this field is blank (`' '`), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'position'`.

- 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 in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

Limitations

- 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.

Programmatic Use
Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select **Analysis > Control Design > Model Discretizer**. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Discrete Transfer Fcn | State-Space

Topics

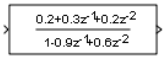
“States”

Introduced before R2006a

Transfer Fcn Direct Form II

Implement Direct Form II realization of transfer function

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal

scalar | vector

Input signal, specified as a scalar or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal

scalar | vector

Output signal, specified as a scalar or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Numerator coefficients` — Numerator coefficients

`[0.2 0.3 0.2]` (default) | vector

Specify the numerator coefficients as a vector.

Programmatic Use
Block Parameter: `NumCoefVec`
Type: character vector
Values: vector
Default: `'[0.2 0.3 0.2]'`

`Denominator coefficients excluding lead` — Denominator coefficient

`[-0.9 0.6]` (default) | vector

Specify the denominator coefficients, excluding the leading coefficient, which must be `1.0`.

Programmatic Use
Block Parameter: `DenCoeffVec`
Type: character vector
Values: vector
Default: `'[-0.9 0.6]'`

`Initial condition` — Initial condition

`0.0` (default) | scalar

Specify the initial condition as a scalar.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar
Default: `'0.0'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`

Zero-Crossing Detection	No

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.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Discrete Transfer Fcn | Transfer Fcn Direct Form II Time Varying

Topics

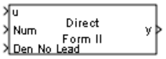
“Fixed-Point Numbers”

Introduced before R2006a

Transfer Fcn Direct Form II Time Varying

Implement time varying Direct Form II realization of transfer function

Library: Simulink / 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, to use a denominator of `[1 -1.7 0.72]`, specify 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.

Ports

Input

u — Input signal
scalar | vector

Input signal, specified as a scalar or vector.

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

`Num` — Numerator coefficients

scalar | vector

Numerator coefficients of the transfer function, specified as a scalar or vector.

Dependencies

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.

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

`Den No Lead` — Denominator coefficients

scalar | vector

Denominator coefficients of the transfer function, specified as a scalar or vector, without the leading coefficient of one. The full denominator has a leading coefficient of one, but it is excluded from the input signal.

Example: For a denominator coefficient of `[1 -1.7 0.72]`, specify a signal with value `[-1.7 0.72]`.

Dependencies

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.

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

Output

y — Output signal

scalar | vector

Output signal, specified as a scalar or vector.

The block automatically selects the data types and scalings of the output and any temporary variables.

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

Parameters

`Initial condition` — Initial condition

`0.0` (default) | scalar

Specify the initial condition as a scalar.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar
Default: `'0.0'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations. For more information, see "Rounding" (Fixed-Point Designer).

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean`[a] | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

[a] This block is not recommended for use with Boolean signals.

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.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Discrete Transfer Fcn | Transfer Fcn Direct Form II

Topics

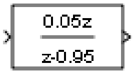
“Fixed-Point Numbers”

Introduced before R2006a

Transfer Fcn First Order

Implement discrete-time first order transfer function

Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector

Input signal to the first order transfer function algorithm.

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

Output

`Port_1` — Transfer function output signal
scalar | vector

Output signal that is the discrete-time first order transfer function of the input with a unity DC gain.

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

Parameters

`Pole (in Z plane)` — Pole

`0.95` (default) | scalar

Specify the pole.

Programmatic Use
Block Parameter: `PoleZ`
Type: character vector
Value: real scalar
Default: `'0.95'`

`Initial condition for previous output` — Initial condition for previous output

`0.0` (default) | scalar

Specify the initial condition for the previous output.

Programmatic Use
Block Parameter: `ICPrevOutput`
Type: character vector
Value: real scalar
Default: `'0.0'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations. For more information, see "Rounding" (Fixed-Point Designer).

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action
`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

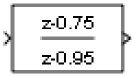
Transfer Fcn | Transfer Fcn Lead or Lag

Introduced before R2006a

Transfer Fcn Lead or Lag

Implement discrete-time lead or lag compensator

Library: Simulink / 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 1, 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`.

Ports

Input

`Port_1` — Input signal
scalar | vector

Input signal that the block applies the discrete-time lead or lag compensation to.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector

Output signal that is discrete-time lead or lag compensation of the input sign.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Pole of compensator (in Z plane)` — Pole

`0.95` (default) | scalar

Specify the pole of the compensator.

Programmatic Use
Block Parameter: `PoleZ`
Type: character vector
Value: real scalar
Default: `'0.95'`

`Zero of compensator (in Z plane)` — Zero of compensator

`0.75` (default) | scalar

Specify the zero of compensator in the Z plane.

Programmatic Use
Block Parameter: `ZeroZ`
Type: character vector
Value: real scalar
Default: `'0.75'`

`Initial condition for previous output` — Initial condition for previous output

`0.0` (default) | scalar

Specify the initial condition for the previous output.

Programmatic Use
Block Parameter: `ICPrevOutput`
Type: character vector
Value: real scalar
Default: `'0.0'`

`Initial condition for previous input` — Initial condition for previous input

`0.0` (default) | scalar

Specify the initial condition for the previous input.

Programmatic Use
Block Parameter: `ICPrevInput`
Type: character vector
Value: real scalar
Default: `'0.0'`

`Initial condition for previous output` — Initial condition for previous output

`0.0` (default) | scalar

Specify the initial condition for the previous output.

Programmatic Use
Block Parameter: `ICPrevOutput`
Type: character vector
Value: real scalar
Default: `'0.0'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate to max or min when overflows occur` — Method of overflow action

`off` (default) | `on`

When you select this check box, 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.

Programmatic Use
Block Parameter: `DoSatur`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean`[a] | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

[a] This block is not recommended for use with Boolean signals.

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

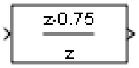
See Also

Transfer Fcn | Transfer Fcn First Order

Introduced before R2006a

Transfer Fcn Real Zero

Implement discrete-time transfer function that has real zero and no pole
Library: Simulink / Discrete



Description

The Transfer Fcn Real Zero block implements a discrete-time transfer function that has a real zero and effectively no pole.

Ports

Input

`Port_1` — Input signal
scalar | vector

Input signal to the discrete-time transfer function algorithm.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector

Output signal that is the discrete-time transfer function with a real zero and effectively no pole of the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Zero (in Z plane)` — Zero

`0.75` (default) | scalar

Specify the zero in the Z plane.

Programmatic Use
Block Parameter: `ZeroZ`
Type: character vector
Value: real scalar
Default: `'0.75'`

`Initial condition for previous input` — Initial condition for previous input

`0.0` (default) | scalar

Specify the initial condition for the previous input.

Programmatic Use
Block Parameter: `ICPrevInput`
Type: character vector
Value: real scalar
Default: `'0.0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

Saturate to max or min when overflows occur — Method of overflow action
off (default) | on

When you select this check box, 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.

Programmatic Use
Block Parameter: DoSatur
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

Data Types	double | single | Boolean[a] | base integer | fixed point
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No

[a] This block is not recommended for use with Boolean signals.

Extended Capabilities

C/C++ Code Generation

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

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Transfer Fcn | Transfer Fcn Lead or Lag

Introduced before R2006a

Transport Delay

Delay input by given amount of time
Library: Simulink / 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”.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal to delay, specified as a scalar, vector, or matrix.

Data Types: `double`

Output

`Port_1` — Delayed signal

scalar | vector | matrix

Input signal, delayed by specified amount of time. Output has the same dimensions and data type as the input signal.

Data Types: `double`

Parameters

`Time delay` — Time delay

`1` (default) | `scalar` | vector | matrix

Specify the amount of simulation time to delay the input signal before propagation to the output as a non-negative scalar, vector, or matrix.

Programmatic Use:
Block Parameter: `DelayTime`
Type: character vector, string
Values: non-negative scalar, vector, or matrix
Default: `'1'`

`Initial output` — Initial output value

`0` (default) | scalar | vector | matrix

Specify the output that the block generates until the simulation time first exceeds the time delay input as a scalar, vector, or matrix.

Limitations

The initial output of this block cannot be `inf` or `NaN`.

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.

Programmatic Use

Block Parameter: `InitialOutput`
Type: character vector, string
Values: scalar | vector | matrix
Default: `'0'`

`Initial buffer size` — Initial memory allocation

`1024` (default) | positive integer scalar

Define the initial memory allocation for the number of input points to store.

- 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.

Tips

- 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.

Programmatic Use

Block Parameter: `BufferSize`
Type: character vector, string
Value: positive integer scalar
Default: `'1024'`

`Use fixed buffer size` — Use fixed-size buffer

`off` (default) | `on`

Select this check box to use a fixed-size buffer to save input data from previous time steps.

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.

Tips

- 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.

Programmatic Use

Block Parameter: `FixedBuffer`
Type: character vector, string
Value: `'off'` | `'on'`
Default: `'off'`

`Direct feedthrough of input during linearization` — Enable direct feedthrough

`off` (default) | `on`

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

Tips

- 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.

Programmatic Use
Block Parameter: TransDelayFeedthrough
Type: character vector, string
Value: 'off' | 'on'
Default: 'off'

Pade order (for linearization) — Order of Pade approximation
0 (default) | scalar | vector | matrix

Set the order of the Pade approximation for linearization routines as a scalar, vector, or matrix of nonnegative integers.

- 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.

Programmatic Use
Block Parameter: PadeOrder
Type: character vector, string
Values: scalar | vector | matrix
Default: '0'

Block Characteristics

Data Types	double
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select **Analysis > Control Design > Model Discretizer**. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Unit Delay | Variable Time Delay

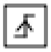
Topics

"Model Discretizer"

Introduced before R2006a

Trigger

Add trigger port to subsystem or model
Library: Simulink / 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
Values: `'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
Values: `'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
Values: `'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
Values: `'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
Values: `'(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
Values: `'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
Values: `'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 by Using Bind Actions” (Stateflow).
- 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
Values: `'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
Values: `'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
Values: `'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
Values: `'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
Values: `'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
Values: `'-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
Values: `'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
Values: `'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
Values: `'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
Values: `'-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” (Simulink Coder).

`[]`

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
Values: `'[]'|'<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” (Simulink Coder).

`[]`

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
Values: `'[]'` | `'<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)`

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
Values: `'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
Values: `'on'` | `'off'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`

Variable-Size Signals	No
Zero-Crossing Detection	Yes

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™.

For more information about HDL code generation, see Trigger.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Triggered Subsystem

Topics

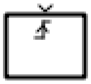
“Conditionally Executed Subsystems Overview”
“Using Triggered Subsystems”
“Using Enabled and Triggered Subsystems”

“Using Function-Call Subsystems”
“Export-Function Models”

Introduced before R2006a

Trigger-Based Linearization

Generate linear models in base workspace when triggered
Library: Simulink / 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
`c`	The C matrix of the linearization
`d`	The D matrix of the linearization
`StateName`	Names of the model's states

Field	Description
`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 (`OperPoint.x`) and inputs (`OperPoint.u`) at the operating point time (`OperPoint.t`).
`Ts`	The sample time of the linearization for a discrete linearization

Tip Use the Timed-Based Linearization block to generate linear models at predetermined times.

Ports

Input

`Port_1` — Input signal

scalar

Input trigger signal specified as a scalar. You specify the type of event that triggers generation of a linear model using the **Trigger type** parameter.

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

Parameters

`Trigger type` — Signal type that triggers generation of a linear model

`rising` | `falling` | `either` | `function-call`

Type of event on the trigger input signal that triggers generation of a linear model. You can select:

- `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.

Programmatic Use
Block Parameter: `TriggerType`
Type: character vector
Values: `'rising'` | `'falling'` | `'either'` | `'function-call'`
Default: `'rising'`

Sample time (of linearized model) — Sample time

`0` (default) | scalar | vector

Specify a sample time to create a discrete-time linearization of the model (see “Discrete-Time System Linearization” on page 2-54).

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'0'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`

Zero-Crossing Detection	No

Extended Capabilities

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Timed-Based Linearization | `dlinmod` | `linmod`

Topics

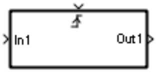
"Discrete-Time System Linearization" on page 2-54

Introduced before R2006a

Triggered Subsystem

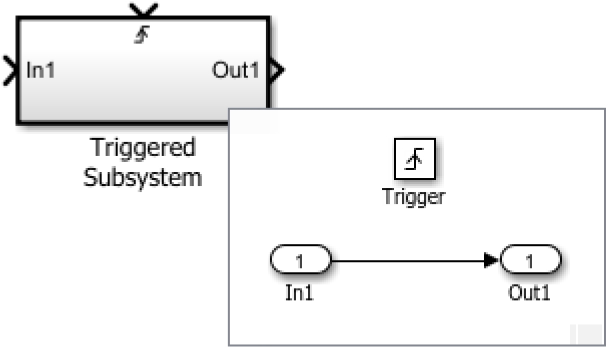
Subsystem whose execution is triggered by external input

Library: Simulink / 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`

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[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 code generation support depends on block implementation.

HDL Code Generation

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

For more information about HDL code generation, see Triggered Subsystem.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks

Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Trigger

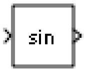
Topics

“Conditionally Executed Subsystems Overview”
“Using Enabled Subsystems”
“Using Triggered Subsystems”
“Using Enabled and Triggered Subsystems”
“Using Function-Call Subsystems”

Introduced before R2006a

Trigonometric Function

Specified trigonometric function on input
Library: Simulink / Math Operations



Description

The Trigonometric Function block performs common trigonometric functions and outputs the result in rad.

Supported Functions

You can select one of these 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	asin(*u*)	`asin`
`acos`	Inverse cosine of the input	acos(*u*)	`acos`
`atan`	Inverse tangent of the input	atan(*u*)	`atan`
`atan2`	Four-quadrant inverse tangent of the input	atan2(*u*)	`atan2`

Function	Description	Mathematical Expression	MATLAB Equivalent
sinh	Hyperbolic sine of the input	sinh(*u*)	sinh
cosh	Hyperbolic cosine of the input	cosh(*u*)	cosh
tanh	Hyperbolic tangent of the input	tanh(*u*)	tanh
asinh	Inverse hyperbolic sine of the input	asinh(*u*)	asinh
acosh	Inverse hyperbolic cosine of the input	acosh(*u*)	acosh
atanh	Inverse hyperbolic tangent of the input	atanh(*u*)	atanh
sincos	Sine of the input; cosine of the input	—	—
cos + jsin	Complex exponential of the input	—	—

CORDIC Approximation Method

If you use the CORDIC approximation method (see “Definitions” on page 1-2095), the block input has some further requirements.

When you set **Function** to sin, cos, sincos, or cos + jsin, and set the **Approximation method** to CORDIC, the block has these limitations:

- When you use signed fixed-point types, the input angle must fall within the range [–2π, 2π) rad.
- When you use unsigned fixed-point types, the input angle must fall within the range [0, 2π) rad.

When you set **Function** to atan2 and the **Approximation method** to CORDIC, the block has these limitations:

- 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	

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input specified as a scalar, vector, or matrix. The block accepts input signals of the following data types:

Functions	Input Data Types
• `sin` • `cos` • `sincos` • `cos + jsin` • `atan2`	• Floating point • Fixed point (only when **Approximation method** is `CORDIC`)

Functions	Input Data Types
• `tan` • `asin` • `acos` • `atan` • `sinh` • `cosh` • `tanh` • `asinh` • `acosh` • `atanh`	• Floating point

Dependencies

- When you set **Function** to `atan2`, the block shows two input ports. The first input (**Port_1**) is the *y*-axis or imaginary part of the function argument. The second input (**Port_2**) is the *x*-axis or real part of the function argument.
- 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.

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.

Limitations

Complex input signals are supported for all functions in this block, except `atan2`.

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	`sin`, `cos`, `sincos`, and `cos + jsin`	`fixdt(1, WL, WL - 2)` where *WL* is the input word length This fixed-point type provides the best precision for the CORDIC algorithm.
Fixed point, signed	`atan2`	`fixdt(1, WL, WL – 3)`
Fixed point, unsigned	`atan2`	`fixdt(1, WL, WL – 2)`

When you set **Function** to `sin`, `cos`, `sincos`, or `cos + jsin`, and set the **Approximation method** to `CORDIC`, the block has these limitations:

- When you use signed fixed-point types, the input angle must fall within the range [–2π, 2π) rad.
- When you use unsigned fixed-point types, the input angle must fall within the range [0, 2π) rad.

When you set **Function** to `atan2` and the **Approximation method** to `CORDIC`, the block has these limitations:

- 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Port_2` — *x*-axis or real part of the function argument for `atan2`

scalar | vector | matrix

Input the *x*-axis or real part of the function argument for `atan2`. When you set **Function** to `atan2`, the block shows two input ports. The first input (**Port_1**) is the *y*-axis or imaginary part of the function argument. The second input (**Port_2**) is the *x*-axis or real part of the function argument. (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.)

Dependencies

To enable this port, set **Function** to `atan2`.

Limitations

- Fixed-point input signals are supported only when you set **Approximation method** to `CORDIC`.
- When you set **Function** to `atan2` and **Approximation method** to `CORDIC`:
  - 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Specified trigonometric function of input

scalar | vector | matrix

Result of applying the specified trigonometric function to one or more inputs in rad. Each function supports:

- Scalar operations
- Element-wise vector and matrix operations

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`sin` — Sine of input signal

scalar | vector | matrix

Sine of the input signal, in rad.

Dependencies

To enable this port, set **Function** to `sincos`.

Limitations

Fixed-point input signals are supported only when you set **Approximation method** to `CORDIC`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`cos` — Cosine of input signal

scalar | vector | matrix

Cosine of the input signal, in rad.

Dependencies

To enable this port, set **Function** to `sincos`.

Limitations

Fixed-point input signals are supported only when you set **Approximation method** to `CORDIC`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Function` — Trigonometric function

`sin` (default) | `cos` | `tan` | `asin` | `acos` | `atan` | `atan2` | `sinh` | `cosh` | `tanh` | `asinh` | `acosh` | `atanh` | `sincos` | `cos + jsin`

Specify the trigonometric function. The name of the function on the block icon changes to match your selection.

Limitations

When you set **Function** to `sin`, `cos`, `sincos`, or `cos + jsin`, and set the **Approximation method** to `CORDIC`, the block has these limitations:

- When you use signed fixed-point types, the input angle must fall within the range [–2π, 2π) rad.
- When you use unsigned fixed-point types, the input angle must fall within the range [0, 2π) rad.

When you set **Function** to `atan2` and the **Approximation method** to `CORDIC`, the block has these limitations:

- 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.

Programmatic Use

Block Parameter: `Operator`
Type: character vector
Values: `'sin'` | `'cos'` | `'tan'` | `'asin'` | `'acos'` | `'atan'` | `'atan2'` | `'sinh'` | `'cosh'` | `'tanh'` | `'asinh'` | `'acosh'` | `'atanh'` | `'sincos'` | `'cos + jsin'`
Default: `'sin'`

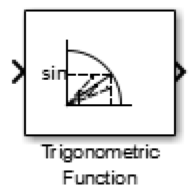
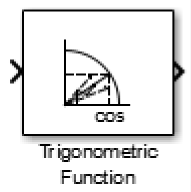
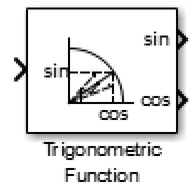
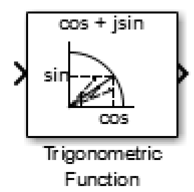
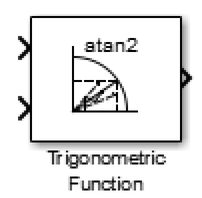
Approximation method — CORDIC or none

`None` (default) | `CORDIC`

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.

If you select `CORDIC` and enlarge the block from the default size, the block icon changes:

Function	Block Icon
`sin`	
`cos`	
`sincos`	
`cos + jsin`	
`atan2`	

Dependencies

To enable this parameter, set **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`.

To use fixed-point input signals, you must set **Approximation method** to `CORDIC`.

Limitations

When you set **Function** to `sin`, `cos`, `sincos`, or `cos + jsin`, and set the **Approximation method** to `CORDIC`, the block has these limitations:

- When you use signed fixed-point types, the input angle must fall within the range [–2π, 2π) rad.
- When you use unsigned fixed-point types, the input angle must fall within the range [0, 2π) rad.

When you set **Function** to `atan2` and the **Approximation method** to `CORDIC`, the block has these limitations:

- 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.

Programmatic Use

Block Parameter: `ApproximationMethod`
Type: character vector
Values: `'None' | 'CORDIC'`
Default: `'None'`

`Number of iterations` — Number of iterations for CORDIC algorithm

`11` (default) | positive integer, less than or equal to word length of fixed-point input

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.

Dependencies

To enable this parameter, you must set the **Function** and **Approximation method** parameters as follows:

- Set **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`.
- Set **Approximation method** to `CORDIC`.

Programmatic Use
Block Parameter: `NumberOfIterations`
Type: character vector
Values: positive integer, less than or equal to word length of fixed-point input
Default: `'11'`

`Output signal type` — complexity of output signal

`auto` (default) | `real` | `complex`

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

Dependencies

Setting **Approximation method** to `CORDIC` disables this parameter.

Note When **Function** is `atan2`, complex input signals are not supported for simulation or code generation.

Programmatic Use
Block Parameter: `OutputSignalType`
Type: character vector
Values: `'auto' | 'real' | 'complex'`
Default: `'auto'`

`Sample time` — Specify sample time as a value other than `-1`

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

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.

References

[1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers* EC-8 (1959); 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): 191–200.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference (1971): 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* 90, no. 5 (1983): 317–325.

Extended Capabilities

C/C++ Code Generation

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

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.

HDL Code Generation

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

For more information about HDL code generation, see Trigonometric Function.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

This block supports fixed-point and base integer data types when you set the **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`, and set the **Approximation method** to `CORDIC`.

See Also

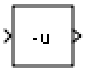
Math Function | Sine, Cosine | Sqrt

Introduced before R2006a

Unary Minus

Negate input

Library: Simulink / Math Operations



Description

The Unary Minus block negates the input.

Ports

Input

`Port_1` — Signal to negate
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

Output

`Port_1` — Negation of input signal
scalar | vector | matrix | N-D array

Negation of the input signal. The output has the same data type and dimensions as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

Parameters

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

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.

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:

Parameter Setting	Block Behavior	Examples
Saturate on integer overflow = `on`	Values saturate to the most positive value of the integer data type	• 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.
Saturate on integer overflow = `off`	Values wrap to the most negative value of the integer data type	• For 8-bit signed integers, -128 remains -128. • For 16-bit signed integers, -32768 remains -32768. • For 32-bit signed integers, -2147483648 remains -2147483648.

Programmatic Use

Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Specify sample time as a value other than -1

`-1` (default) | scalar

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

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

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'-1'`

Block Characteristics

Data Types	`double` | `single` | `base integer`[a] | `fixed point`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

a. This block only supports signed fixed-point data types.

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™.

For more information about HDL code generation, see Unary Minus.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

This block only supports signed fixed-point data types.

See Also

`uminus`

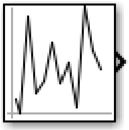
Topics

"Operator Precedence" (MATLAB)

Introduced before R2006a

Uniform Random Number

Generate uniformly distributed random numbers
Library: Simulink / 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.

Ports

Output

`Port_1` — Random number output signal
scalar | vector

Output signal of generated uniformly distributed random numbers over the interval you specify.

Data Types: `double`

Parameters

`Minimum` — Minimum interval

`-1` (default) | scalar | vector

Specify the minimum of the interval.

Programmatic Use
Block Parameter: `Minimum`
Type: character vector
Values: scalar
Default: `'-1'`

`Maximum` — Maximum interval

`1` (default) | scalar | vector

Specify the maximum of the interval.

Programmatic Use
Block Parameter: `Maximum`
Type: character vector
Values: scalar
Default: `'1'`

Seed — Random number seed

`0` (default) | scalar

Specify the starting seed for the random number generator.

The seed must be 0 or a positive integer. Output is repeatable for a given seed.

Programmatic Use
Block Parameter: See
Type: character vector
Values: scalar
Default: `'0'`

`Sample time` — Sample time

`0.1` (default) | scalar

Specify the time interval between samples. See “Specify Sample Time” in the Simulink documentation for more information.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'0.1'`

`Interpret vector parameters as 1-D` — Treat vectors as 1-D

`on` (default) | `off`

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.

- When you select this check box, the block 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`.
- When you clear this check box, the block does not output a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector.

Programmatic Use
Block Parameter: `VectorParams1D`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`

Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

See Also

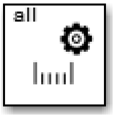
Random Number | Repeating Sequence

Introduced before R2006a

Unit System Configuration

Configure units

Library: Simulink / 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` — Disallowed unit systems

`SI | English | SI (extended) | CGS`

Displays a list of the disallowed unit systems. By default, the **Allow all unit systems** check box is selected, and all unit systems are allowed.

To designate a unit system as disallowed, select it in the **Allowed unit systems** column and click **<< Disallow**.

Dependencies

To enable changes to this parameter, you must first clear the **Allow all unit systems** check box.

`Allowed unit systems` — Allowed unit systems

`SI | English | SI (extended) | CGS`

Displays a list of the allowed unit systems. By default, the **Allow all unit systems** check box is selected, and all unit systems are allowed.

To designate a unit system as allowed, use the **Allow >>** and **<< Disallow** buttons to move unit systems between the **Disallowed unit systems** and **Allowed unit systems** columns.

Dependencies

To enable changes to this parameter, you must first clear the **Allow all unit systems** check box.

Programmatic Use
Block Parameter: `UnitSystems`
Type: cell array of character vectors
Values: cell array of the following character vectors: `'SI'` | `'English'` | `'SI (extended)'` | `'CGS'`
Default: `{'SI', 'English', 'SI (extended)', 'CGS'}`

`Allow all unit systems` — Allow all unit systems

`on` (default) | `off`

When you select this check box, all unit systems are allowed. To restrict the allowed unit systems to only the ones specified in the **Allowed unit systems** column, clear this check box.

Dependencies

Selecting the **Allow all unit systems** check box disables the **Disallowed unit systems** and **Allowed unit systems** parameters.

Programmatic Use
Block Parameter: `AllowAllUnitSystems`
Type: character vector
Values: `'on' | 'off'`
Default: `'on'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated` | `bus`
Direct Feedthrough	`No`

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Inport | Outport | Unit Conversion

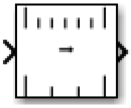
Topics

“Update an Existing Model to Use Units”
“Units in Simulink”
“Unit Specification in Simulink Models”
“Restricting Unit Systems”

Introduced in R2016a

Unit Conversion

Convert units
Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal to convert units on, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector | matrix

Output signal with converted units, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

`Output data type` — Output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation`

Specify the output data type.

- `Inherit: Inherit via internal rule` —Simulink chooses intermediate and output data types to balance numerical accuracy, performance, and generated code size, while accounting for 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.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit via internal rule' | 'Inherit via back propagation'`
Default: `'Inherit via internal rule'`

Block Characteristics

Data Types	`double` | `single` | `base integer` | `fixed point`
Direct Feedthrough	Yes
Multidimensional Signals	Yes

Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

"Converting Units"

Topics

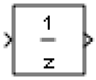
"Units in Simulink"
"Converting Units"

Introduced in R2016a

Unit Delay

Delay signal one sample period

Library: Simulink / 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

The Memory, Unit Delay, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different.

This table shows 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.	• `sldemo_enginewc` (Compression subsystem)
Memory on page 1-1222	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.	• `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.	• `sldemo_radar_eml` • `aero_dap3dof`

Each block has the following capabilities.

Capability	Memory	Unit Delay	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	No, because the block can only inherit sample time from the driving block or the solver used for the entire model.	Yes	Yes
Support for frame-based signals	No	Yes	Yes
Support for state logging	No	Yes	No

Ports

Input

`Port_1` — Input signal

scalar | vector

Input signal that the block delays by one sample period.

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

Output

`Port_1` — Output signal

scalar | vector

Output signal that is the input delayed by one sample period.

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

Parameters

Main

`Initial condition` — First sample period output

`0` (default) | scalar | vector

Specify the output of the simulation for the first sampling period, during which the output of the Unit Delay block is otherwise undefined.

Programmatic Use
Block Parameter: `InitialCondition`
Type: character vector
Value: scalar | vector
Default: `'0'`

`Input processing` — Specify sample- or frame-based processing

`Elements as channels (sample based)` (default) | `Columns as channels (frame based)`

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use

Block Parameter: `InputProcessing`
Type: character vector
Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Sample time (-1 for inherited)` — Discrete interval between sample time hits

`-1` (default) | `scalar`

Enter the discrete interval between sample time hits or specify `-1` to inherit the sample time.

See also “Specify Sample Time”.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Value: real scalar
Default: `'-1'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: `StateName`
Type: character vector
Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state names resolve to signal object

`Off` (default) | Boolean

Specify if requiring that state name resolve to Simulink signal objects or not. If selected, the software generates an error at run time if you specify a state name that does not match the name of a Simulink signal object.

Dependency

Enabled when you give the parameter **State name** a value and set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Programmatic Use
Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default)

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use
Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`

Default: `'Simulink.Signal'`

`Code generation storage class` — Storage class for code generation
`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `Bitfield (Custom)` | `Volatile (Custom)` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation. If you do not need to interface to external code, select `Auto`.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use
Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto'` | `'Model default'` | `'ExportedGlobal''` | `'ImportedExtern'` | `'ImportedExternPointer'` | `'Custom'`
Default: `'Auto'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on memcpy or memset functions (strong.h) under certain conditions.

HDL Code Generation

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

For information about HDL code generation, see Unit Delay.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

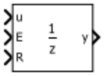
Delay | Resettable Delay

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



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	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

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

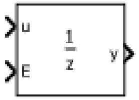
State Control | Unit Delay | Unit Delay Resettable Synchronous | Unit Delay Enabled Synchronous

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



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

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

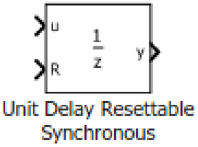
State Control | Unit Delay | Unit Delay Resettable Synchronous | Unit Delay Enabled Resettable Synchronous

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

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

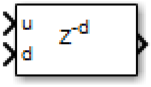
State Control | Unit Delay | Unit Delay Enabled Resettable Synchronous | Unit Delay Enabled Synchronous

Introduced in R2017b

Variable Integer Delay

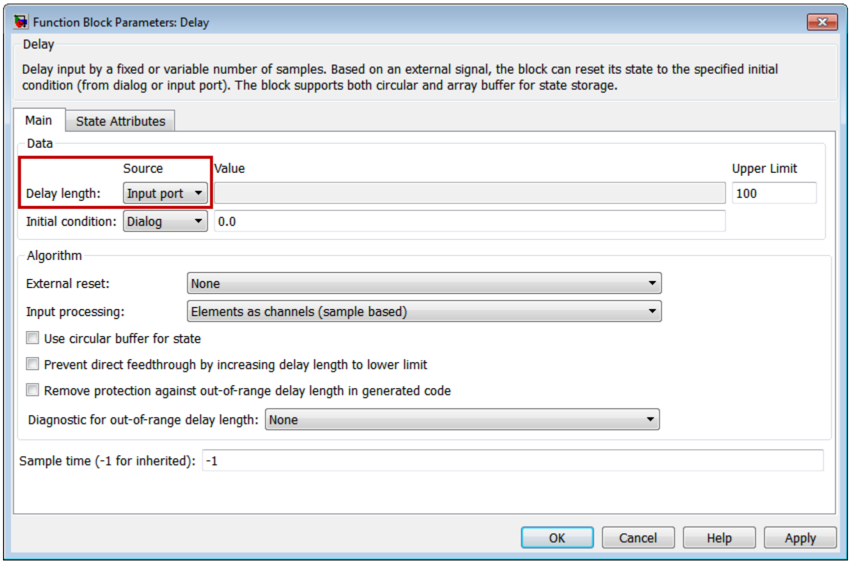
Delay input signal by variable sample period

Library: Simulink / 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.



Function Block Parameters: Delay

Delay

Delay input by a fixed or variable number of samples. Based on an external signal, the block can reset its state to the specified initial condition (from dialog or input port). The block supports both circular and array buffer for state storage.

Data

	Source	Value	Upper Limit
Delay length:	Input port		100
Initial condition:	Dialog	0.0	

Algorithm

External reset: None

Input processing: Elements as channels (sample based)

□ Use circular buffer for state

□ Prevent direct feedthrough by increasing delay length to lower limit

□ Remove protection against out-of-range delay length in generated code

Diagnostic for out-of-range delay length: None

Sample time (-1 for inherited): -1

Ports

Input

u — Data input signal

scalar | vector

Input data signal delayed according to parameters settings.

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

d — Delay length

scalar

Delay length specified as inherited from an input port. Enabled when you select the **Delay length: Source** parameter as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`Enable` — External enable signal

scalar

Enable signal that enables or disables execution of the block. To create this port, select the **Show enable port** parameter.

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

`External reset` — External reset signal

scalar

External signal that resets execution of the block to the initial condition. To create this port, select the **External reset** parameter.

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

`x0` — Initial condition

scalar | vector

Initial condition specified as inherited from an input port. Enabled when you select the **Initial Condition: Source** parameter as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

`Port_1` — Output signal
scalar | vector

Output signal that is the input signal delayed by the length of time specified by the parameter **Delay length**. The initial value of the output signal depends on several conditions. See “Initial Block Output” on page 1-340.

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

Parameters

Main

`Delay length` — Delay length
`Dialog` (default) | `Input port`

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.

Programmatic Use

Block Parameter: `DelayLengthSource`
Type: character vector
Values: `'Dialog'` | `'Input port'` |
Default: `'Dialog'`
Block Parameter: `DelayLength`
Type: character vector
Values: scalar
Default: `'2'`
Block Parameter: `DelayLengthUpperLimit`
Type: character vector
Values: scalar
Default: `'100'`

`Initial condition` — Initial condition

`Dialog` (default) | `Input port`

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.

Programmatic Use

Block Parameter: `InitialConditionSource`
Type: character vector
Values: `'Dialog'` | `'Input port'` |
Default: `'Dialog'`
Block Parameter: `InitialCondition`
Type: character vector

Values: scalar
Default: '0.0'

Input processing — Specify sample- or frame-based processing

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-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).

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

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).

Programmatic Use

Block Parameter: InputProcessing
Type: character vector

Values: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`
Default: `'Elements as channels (sample based)'`

`Use circular buffer for state` — Circular buffer for storing state

`off` (default) | `on`

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.

Programmatic Use

Block Parameter: `UseCircularBuffer`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Prevent direct feedthrough` — Prevent direct feedthrough

`off` (default) | `on`

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.

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use

Block Parameter: `PreventDirectFeedthrough`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Remove delay length check in generated code` — Remove delay length out-of-range check

`off` (default) | `on`

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

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use

Block Parameter: `RemoveDelayLengthCheckInGeneratedCode`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Diagnostic for delay length` — Diagnostic checks for delay length

`None` (default) | `Warning` | `Error`

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.

Dependency

To enable this parameter, set **Delay length: Source** to `Input port`.

Programmatic Use

Block Parameter: `DiagnosticForDelayLength`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

`Show enable port` — Create enable port

`off` (default) | `on`

Select to control execution of this block with an enable port. 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.

Programmatic Use

Block Parameter: `ShowEnablePort`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`External reset` — External state reset

`None` (default) | `Rising` | `Falling` | `Either` | `Level` | `Level hold`

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.

Reset Mode	Behavior
`Level`	Reset in either of these cases: • 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

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector
Values: `'None'` | `'Rising'` | `'Falling'` | `'Either'` | `'Level'` | `'Level hold'`
Default: `'None'`

`Sample time (-1 for inherited)` — Discrete interval between sample time hits

`-1` (default) | `scalar`

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.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Value: real scalar
Default: `'-1'`

State Attributes

`State name` — Unique name for block state

`''` (default) | alphanumeric string

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use

Block Parameter: `StateName`
Type: character vector
Values: unique name
Default: `''`

`State name must resolve to Simulink signal object` — Require state name resolve to a signal object

`off` (default) | `on`

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, specify a value for **State name**. 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**.

Programmatic Use

Block Parameter: `StateMustResolveToSignalObject`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Signal object class` — Custom storage class package name

`Simulink.Signal` (default) | `<StorageClass.PackageName>`

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 “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Programmatic Use
Block Parameter: `StateSignalObject`
Type: character vector
Values: `'Simulink.Signal' | '<StorageClass.PackageName>'`
Default: `'Simulink.Signal'`

`Code generation storage class` — State storage class for code generation

`Auto` (default) | `Model default` | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | `BitField (Custom)` | `Model default` | `ExportToFile (Custom)` | `ImportFromFile (Custom)` | `FileScope (Custom)` | `AutoScope (Custom)` | `Struct (Custom)` | `GetSet (Custom)` | `Reusable (Custom)`

Select state storage class for code generation.

- `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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

To enable this parameter, specify a value for **State name**.

Programmatic Use
Block Parameter: `StateStorageClass`
Type: character vector
Values: `'Auto' | 'SimulinkGlobal' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...`
Default: `'Auto'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Consider using the Model Discretizer to map these continuous blocks into discrete equivalents that support code generation. From a model, select **Analysis** > **Control Design** > **Model Discretizer**.

Not reommended for production code.

HDL Code Generation

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

For information about HDL code generation, see Delay.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Delay | Resettable Delay | Tapped Delay | Unit Delay

Topics

“Using Enabled Subsystems”

Introduced in R2012b

Variable Time Delay

Delay input by variable amount of time

Library: Simulink / Continuous

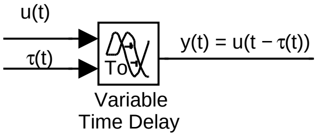
Description

The Variable Transport Delay and Variable Time Delay blocks appear as two blocks in the Simulink block library. However, they are the same Simulink block with different settings for the **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” 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 must 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

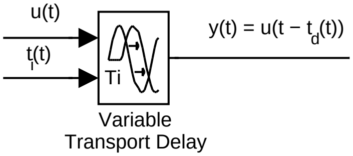
In this mode, the block output at the current time step is equal to the value of its data (top, or left) input at an earlier time step equal to the current time minus a transportation delay.

$$y(t) = u(t - t_d(t))$$

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 that 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.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix

Input signal specified as a scalar, vector, or matrix.

Data Types: double

t_0 — Time delay input

scalar | vector | matrix

Time delay input specified as a scalar, vector, or matrix. When the block is in Variable time delay mode, this value specifies the time delay. For more information about that calculation, see "Variable Time Delay" on page 1-2145.

Dependencies

To enable this port, set **Select delay type** to Variable time delay.

Data Types: double

t_i — Instantaneous time delay input

scalar | vector | matrix

Instantaneous time delay input specified as a scalar, vector, or matrix. When the block is in Variable transport delay mode, this value is used to calculate the transportation delay. For more information about that calculation, see "Variable Transport Delay" on page 1-2146.

Dependencies

To enable this port, set **Select delay type** to Variable transport delay.

Data Types: double

Output

Port_1 — Delayed signal
scalar | vector | matrix

Output signal specified as a scalar, vector, or matrix.

Data Types: double

Parameters

Select delay type — Type of delay
Variable time delay | Variable transport delay

Specify the type of delay as Variable time delay or Variable transport delay.

The default value of this parameter depends on the block implementation: Variable time delay for the Variable Time Delay block, and Variable transport delay for the Variable Transport Delay block.

Dependencies

- 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.

Programmatic Use
Block Parameter: VariableDelayType
Type: character vector, string
Values: 'Variable transport delay' | 'Variable time delay'

Maximum delay — Maximum value of time delay input
10 (default) | scalar | vector

Set the maximum value of the time delay input. 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.

Programmatic Use

Block Parameter: `MaximumDelay`
Type: character vector, string
Value: scalar | vector
Default: `'10'`

`Initial output` — Initial output

`0` (default) | scalar | vector

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Dependencies

- The initial output of this block cannot be `inf` or `NaN`.
- A `Run-to-run tunable parameter` cannot be changed during simulation run time. However, changing it before a simulation begins does not cause Accelerator or Rapid Accelerator to regenerate code.

Programmatic Use

Block Parameter: `InitialOutput`
Type: character vector, string
Values: scalar | vector
Default: `'0'`

`Initial buffer size` — Initial memory allocation

`1024` (default) | scalar

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.

- 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 must be allocated.

Tips

- 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.

Programmatic Use

Block Parameter: `MaximumPoints`
Type: character vector, string
Values: scalar | vector
Default: `'1024'`

`Use fixed buffer size` — Use fixed-size buffer

`off` (default) | `on`

Selecting this check box uses a fixed-size buffer to save input data from previous time steps. When you clear this check box, 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.

Tips

- 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.

Programmatic Use

Block Parameter: `FixedBuffer`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Handle zero delay` — Use direct feedthrough

`off` (default) | `on`

Selecting this check box converts this block to a direct feedthrough block. When you clear this check box, the block does not use direct feedthrough.

Dependencies

To enable this parameter, set **Select delay type** to `Variable time delay`.

Programmatic Use

Block Parameter: `ZeroDelay`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Direct feedthrough of input during linearization` — Use direct feedthrough during linearization

`off` (default) | `on`

When you selec this parameter, the block outputs its input during linearization and trim, which sets the block mode to direct feedthrough. To disable direct feedthrough, clear this check box.

Tips

- 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.

Programmatic Use

Block Parameter: `TransDelayFeedthrough`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Pade order (for linearization)` — Order of Pade approximation

`0` (default) | scalar

Set the order of the Pade approximation for linearization routines.

- 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.

Programmatic Use
Block Parameter: PadeOrder
Type: character vector, string
Values: scalar
Default: '0'

Absolute tolerance — Absolute tolerance for computing block state

auto (default) | positive, real, scalar or vector

Specify the absolute tolerance for computing the block state.

Dependencies

To enable this parameter, set **Select delay type** to Variable transport delay.

Programmatic Use
Block Parameter: AbsoluteTolerance
Type: character vector, string
Values: 'auto' | '-1' | any positive real scalar or vector
Default: 'auto'

State Name (e.g., 'position') — Unique name for each state

' ' (default) | character vector, string

Assign a unique name to each state. If this field is blank, no name assignment occurs.

Tips

- 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, string, cell array, or structure.

Dependencies

To enable this parameter, set **Select delay type** to `Variable transport delay`.

Programmatic Use

Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined character vector, user-defined string
Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select `Analysis > Control Design > Model Discretizer`. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Transport Delay | Variable Transport Delay

Topics

“States”

Introduced in R2007a

Variable Transport Delay

Delay input by variable amount of time

Library: Simulink / Continuous

Description

The Variable Transport Delay and Variable Time Delay blocks appear as two blocks in the Simulink block library. However, they are the same Simulink block with different settings for the **Select delay type** parameter. Use this parameter to specify the mode in which the block operates.

Variable Transport Delay

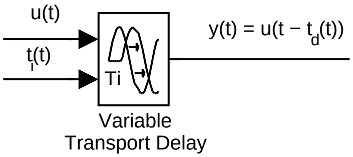
In this mode, the block output at the current time step is equal to the value of its data (top, or left) input at an earlier time step equal to the current time minus a transportation delay.

$$y(t) = u(t - t_d(t))$$

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 that 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

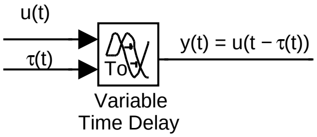
$$t_i(t) = \frac{L}{v_i(t)}$$

where L is the length of the pipe and $v_i(t)$ is the speed of the fluid.

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” 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 must 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.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Input signal specified as a scalar, vector, or matrix.

Data Types: `double`

t_0 — Time delay input

scalar | vector | matrix

Time delay input specified as a scalar, vector, or matrix. When the block is in `Variable time delay` mode, this value specifies the time delay. For more information about that calculation, see "Variable Time Delay" on page 1-2156.

Dependencies

To enable this port, set **Select delay type** to `Variable time delay`.

Data Types: `double`

t_i — Instantaneous time delay input

scalar | vector | matrix

Instantaneous time delay input specified as a scalar, vector, or matrix. When the block is in `Variable transport delay` mode, this value is used to calculate the transportation delay. For more information about that calculation, see "Variable Transport Delay" on page 1-2155.

Dependencies

To enable this port, set **Select delay type** to `Variable transport delay`.

Data Types: `double`

Output

`Port_1` — Delayed signal

scalar | vector | matrix

Output signal specified as a scalar, vector, or matrix.

Data Types: `double`

Parameters

`Select delay type` — Type of delay

`Variable time delay` | `Variable transport delay`

Specify the type of delay as `Variable time delay` or `Variable transport delay`.

The default value of this parameter depends on the block implementation: `Variable time delay` for the Variable Time Delay block, and `Variable transport delay` for the Variable Transport Delay block.

Dependencies

- 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.

Programmatic Use

Block Parameter: `VariableDelayType`
Type: character vector, string
Values: `'Variable transport delay'` | `'Variable time delay'`

`Maximum delay` — Maximum value of time delay input

`10` (default) | scalar | vector

Set the maximum value of the time delay input. 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.

Programmatic Use

Block Parameter: `MaximumDelay`
Type: character vector, string
Value: scalar | vector
Default: `'10'`

`Initial output` — Initial output

`0` (default) | scalar | vector

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Dependencies

- The initial output of this block cannot be `inf` or `NaN`.
- A `Run-to-run tunable parameter` cannot be changed during simulation run time. However, changing it before a simulation begins does not cause Accelerator or Rapid Accelerator to regenerate code.

Programmatic Use

Block Parameter: `InitialOutput`
Type: character vector, string
Values: scalar | vector
Default: `'0'`

`Initial buffer size` — Initial memory allocation

`1024` (default) | scalar

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.

- 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 must be allocated.

Tips

- 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.

Programmatic Use

Block Parameter: `MaximumPoints`
Type: character vector, string
Values: scalar | vector
Default: `'1024'`

`Use fixed buffer size` — Use fixed-size buffer

`off` (default) | `on`

Selecting this check box uses a fixed-size buffer to save input data from previous time steps. When you clear this check box, 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.

Tips

- 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.

Programmatic Use

Block Parameter: `FixedBuffer`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Handle zero delay` — Use direct feedthrough

`off` (default) | `on`

Selecting this check box converts this block to a direct feedthrough block. When you clear this check box, the block does not use direct feedthrough.

Dependencies

To enable this parameter, set **Select delay type** to `Variable time delay`.

Programmatic Use

Block Parameter: `ZeroDelay`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Direct feedthrough of input during linearization` — Use direct feedthrough during linearization

`off` (default) | `on`

When you selec this parameter, the block outputs its input during linearization and trim, which sets the block mode to direct feedthrough. To disable direct feedthrough, clear this check box.

Tips

- 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.

Programmatic Use

Block Parameter: `TransDelayFeedthrough`
Type: character vector, string
Values: `'off'` | `'on'`

Default: 'off'

Pade order (for linearization) — Order of Pade approximation

0 (default) | scalar

Set the order of the Pade approximation for linearization routines.

- 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.

Programmatic Use
Block Parameter: PadeOrder
Type: character vector, string
Values: scalar
Default: '0'

Absolute tolerance — Absolute tolerance for computing block state

auto (default) | positive, real, scalar or vector

Specify the absolute tolerance for computing the block state.

Dependencies

To enable this parameter, set **Select delay type** to Variable transport delay.

Programmatic Use
Block Parameter: AbsoluteTolerance
Type: character vector, string
Values: 'auto' | '-1' | any positive real scalar or vector
Default: 'auto'

State Name (e.g., 'position') — Unique name for each state

' ' (default) | character vector, string

Assign a unique name to each state. If this field is blank, no name assignment occurs.

Tips

- 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, string, cell array, or structure.

Dependencies

To enable this parameter, set **Select delay type** to `Variable transport delay`.

Programmatic Use
Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined character vector, user-defined string
Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select `Analysis > Control Design > Model Discretizer`. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Transport Delay | Variable Time Delay

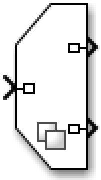
Topics

“States”

Introduced in R2007a

Variant Sink

Route amongst multiple outputs using Variants
Library: Simulink / 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.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix, to be connected to the active output port.

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

Output

Port_1 — Output from first Variant

scalar | vector | matrix

Output signal from the first Variant. The Variant control that evaluates to `true`, determines which output port is active.

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

Port_N — Output from Nth Variant

scalar | vector | matrix

Output signal from the Nth Variant. The Variant control that evaluates to `true`, determines which output port is active.

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

Parameters

Variant control mode — Name of the Variant control mode

`Expression` (default) | `Label`

To choose the active Variant based on the evaluation of the Variant conditions, use the `Expression` mode else select `Label` mode. When you select the **Variant control mode** as `Label`, the **Label mode active choice** option is available. In `Label` mode, Variant control need not be created in the global workspace. In `Label` mode, the Variant control is a string that is not evaluated and the choice used in simulation is determined by the **Label mode active choice** parameter. You can select an active Variant choice from **Label mode active choice** options.

When you select `Label` option, the Variant badge indicates the change.

Note When you promote **Label mode active choice** parameter to a mask, the **Variant control mode** is disabled.

- If the block is in `Expression` mode while promoting **Label mode active choice** parameter to mask, you can change the **Variant control mode** to `Label` by changing the promoted **Label mode active choice** parameter from the Mask dialog box.

- If the block is in `Label` mode while promoting **Label mode active choice** parameter to mask, you cannot change the **Variant control mode** to `Expression` mode.

For information about promoting parameters to mask, see “Promote Parameter to Mask”.

`Port` — Number of connected input port

no default

Number of the input port that is connected to one Variant choice upstream of the Variant Sink block. This value is read-only.

Click ➕ to add a port or ❌ to delete an existing one.

`Variant control expression` — Variant controls available in the global workspace

`'Variant'` (default) | boolean condition expression | `Simulink.Variant` object | `Simulink.Parameter` object | enum

Displays the Variant controls available in the global 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 MATLAB variables.

To enter non-numeric Variant control values, use enumerated data. For information about using enumerated data, see “Use Enumerated Data in Simulink Models”

To enter a Variant name, double-click a **Variant control expression** 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.

Programmatic Use
Block Parameter: `VariantControls`
Type: cell array of character vectors
Values: Variant control that is associated with the Variant choice
Default: `'Variant'`

`Condition (read-only)` — Condition for Variant controls

no default

Displays the **Condition** for the Variant controls that are `Simulink.Variant` objects. Create or change a Variant condition in the `Simulink.Variant` parameter dialog box or in the global workspace.

For more information, see “Create Variant Controls Programmatically” and `Simulink.Variant`.

`Label mode active choice` — Name of Variant to use if Label control mode is selected

`Variant_1` (default) | name of variant control

When you select the **Variant control mode** as `Label`, the **Label mode active choice** option is available. You can select an active Variant choice from **Label mode active choice** options. You can also right-click the badge on the Variant Sink block and select **Label Mode Active Choice**.

The **Label mode active choice** drop-down list displays all Variant controls that are currently defined in the global workspace or a data dictionary. Use valid MATLAB identifiers to specify Variant controls. For more information, see `Simulink.Variant`.

Note **Label mode active choice** option is not available in `Expression` mode.

Dependencies

To enable this parameter, select `Label` mode.

Programmatic Use

Block Parameter: `LabelModeActivechoice`
Type: character vector
Values: Specified by the Variant control expression
Default: `Variant_1`

`Allow zero active variant controls` — Simulate model without active Variant choice

`off` (default) | `on`

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.

Dependencies

`Expression` option from **Variant control mode** is selected.

Programmatic Use

Block Parameter: `AllowZeroVariantControls`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Show variant condition on block` — Annotate block ports

`off` (default) | `on`

When you select this option, Simulink annotates each Variant control (condition expression) on the Variant Sink block ports.

Programmatic Use

Block Parameter: `ShowConditionOnBlock`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Analyze all choices during update diagram and generate preprocessor conditionals` — Analyze all Variant choices during update diagram or simulation

`off` (default) | `on`

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 you select this option, the preprocessor conditionals (`#if`) are generated in the code with ERT-based targets.

When you select this option, the Variant badge indicates the change.

For more information, see “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)

Dependencies

- `Expression` option from **Variant control mode** is selected.
- The check box is available for generating ERT targets only.

Programmatic Use
Block Parameter: `GeneratePreprocessorConditionals`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Manual Variant Sink | Manual Variant Source | Model | `Simulink.Variant` | Variant Source | Variant Subsystem

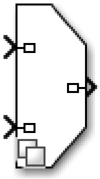
Topics

"Introduction to Variant Controls"
"Define, Configure, and Activate Variants"
"Working with Variant Choices"
"Variant Systems" (Embedded Coder)
"Represent Variant Source and Sink Blocks in Generated Code" (Embedded Coder)
"Variants Example Models"

Introduced in R2016a

Variant Source

Route among multiple inputs using Variants
Library: Simulink / 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.

When the **Analyze all choices during update diagram and generate preprocessor conditionals** option in the block dialog box is cleared, then during simulation Simulink connects the active choice directly to the output port of the Variant Source block and ignores the inactive choices.

Ports

Input

`Port_1` — Input port associated with first Variant control
scalar | vector | matrix

Input port associated with the first Variant control. The Variant control that evaluates to `true`, determines which input port is active.

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

`Port_N` — Input port associated with Nth Variant control

scalar | vector | matrix

Input port associated with the Nth Variant control. The Variant control that evaluates to `true`, determines which input port is active.

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

Output

`Port_1` — Output of active Variant

scalar | vector | matrix

Output signal from the active Variant.

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

Parameters

`Variant control mode` — Name of the Variant control mode

`Expression` (default) | `Label`

To choose the active Variant based on the evaluation of the Variant conditions, use the `Expression` mode else select `Label` mode. When you select the **Variant control mode** as `Label`, the **Label mode active choice** option is available. In `Label` mode, Variant control need not be created in the global workspace. You can select an active Variant choice from **Label mode active choice** options.

When you select `Label` option, the Variant badge indicates the change.

Note When you promote **Label mode active choice** parameter to a mask, the **Variant control mode** is disabled.

- If the block is in `Expression` mode while promoting **Label mode active choice** parameter to mask, you can change the **Variant control mode** to `Label` by changing the promoted **Label mode active choice** parameter from the Mask dialog box.
- If the block is in `Label` mode while promoting **Label mode active choice** parameter to mask, you cannot change the **Variant control mode** to `Expression` mode.

For information about promoting parameters to mask, see “Promote Parameter to Mask”.

Port — Number of connected input port

no default

Number of the input port that is connected to one Variant choice upstream of the Variant Source block. This value is read-only.

Click ✚ to add a port or ✖ to delete an existing one.

`Variant control expression` — Variant controls available in the global workspace

`'Variant'` (default) | boolean condition expression | `Simulink.Variant` object | `Simulink.Parameter` object | enum

Displays the Variant controls available in the global 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.

To enter non-numeric Variant control values, use enumerated data. For information about using enumerated data, see “Use Enumerated Data in Simulink Models”

To edit a Variant name, double-click a **Variant control expression** cell 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.

The **Variant control** that evaluates to `true` determines which input port must be active.

Programmatic Use

Block Parameter: `VariantControls`
Type: cell array of character vectors
Value: Variant control that is associated with the Variant choice

Default: `'Variant'`

`Condition (read-only)` — Condition for Variant controls

no default

Displays the **Condition** for the Variant controls that are `Simulink.Variant` objects. Create or change a Variant condition in the `Simulink.Variant` parameter dialog box or in the global workspace.

For more information, see "Create Variant Controls Programmatically" and `Simulink.Variant`.

`Label mode active choice` — Name of Variant to use if Label control mode is selected

`Variant_1` (default) | name of Variant control

When you select the **Variant control mode** as `Label`, the **Label mode active choice** option is available. You can select an active Variant choice from **Label mode active choice** options. You can also right-click the badge on the Variant Source block and select **Label Mode Active Choice**.

The **Label mode active choice** drop-down list displays all Variant controls that are currently defined in the global workspace or a data dictionary. Use valid MATLAB identifiers to specify Variant controls. For more information, see `Simulink.Variant`.

Note **Label mode active choice** option is not available in `Expression` mode.

Dependencies

To enable this parameter, select `Label` mode.

Programmatic Use

Block Parameter: `LabelModeActivechoice`
Type: character vector
Value: Specified by the Variant control expression.
Default: `Variant_1`

`Allow zero active variant controls` — Simulate model without active Variant choice

`off` (default) | `on`

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.

Dependencies

`Expression` option from **Variant control mode** is selected.

Programmatic Use

Block Parameter: `AllowZeroVariantControls`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Show variant condition on block` — Annotate block ports

`off` (default) | `on`

When you select this option, Simulink annotates each Variant control (condition expression) on the Variant Source block ports.

Programmatic Use

Block Parameter: `ShowConditionOnBlock`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Analyze all choices during update diagram and generate preprocessor conditionals` — Analyze all Variant choices during update diagram or simulation

`off` (default) | `on`

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.

For more information, see “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)

Dependencies

- `Expression` option from **Variant control mode** is selected.
- The check box is available for generating ERT targets only.

Programmatic Use

Block Parameter: `GeneratePreprocessorConditionals`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Manual Variant Sink | Manual Variant Source | Model | `Simulink.Variant` | Variant Sink | Variant Subsystem

Topics

“Introduction to Variant Controls”
“Define, Configure, and Activate Variants”
“Create Variant Controls Programmatically”
“Variant Systems” (Embedded Coder)
“Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
“Variants Example Models”

Introduced in R2016a

Variant Subsystem

Template subsystem containing Subsystem blocks as Variant choices

Library: Simulink / Ports & Subsystems

Description

The Variant Subsystem block can have at most one active choice for simulation. The Variant Subsystem block is a template preconfigured to contain two Subsystem blocks to use as Variant Subsystem choices. The Variant Subsystem block can contain a mixture of Subsystem and Model blocks as Variant systems. The Variant Subsystem blocks can also include Inport, Outport, and Connection Port blocks. There are no drawn connections inside the Variant Subsystem blocks.

Each Variant system is associated with a Variant control that is created in the global 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 block, you can specify a non-ground value as its output.

Note You must specify the correct data type in the **Signal Attributes** section of the Outport block dialog box.

Ports

During simulation, Simulink disables the inactive ports in a Variant Subsystem 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 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 block, then Variant system blocks can have different numbers of outports than the Variant Subsystem block has.

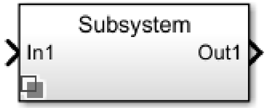
Parameters

`Variant control mode` — Name of the Variant control mode

`Expression` (default) | `Label`

To choose the active Variant based on the evaluation of the Variant conditions, use the `Expression` mode else select `Label` mode. When you select the **Variant control mode** as `Label`, the **Label mode active choice** option is available. In `Label` mode, Variant control need not be created in the global workspace. You can select an active Variant choice from **Label mode active choice** options.

When you select `Label` mode, the Variant badge indicates the change.



Note When you promote **Label mode active choice** parameter to a mask, the **Variant control mode** is disabled. While promoting **Label mode active choice** parameter to a mask in a nested model, ensure that you promote **Label mode active choice** parameter to the immediate parent Variant Subsystem block mask.

- If the block is in `Expression` mode while promoting **Label mode active choice** parameter to mask, you can change the **Variant control mode** to `Label` by changing the promoted **Label mode active choice** parameter from the Mask dialog box.
- If the block is in `Label` mode while promoting **Label mode active choice** parameter to mask, you cannot change the **Variant control mode** to `Expression` mode.

For information about promoting parameters to mask, see “Promote Parameter to Mask”.

`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 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 block.	
Create/Edit selected variant object: Create a `Simulink.Variant` object in the global 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.	

To...	Click...
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 global 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 expression` — Variant control in global workspace

`Variant` (default) | boolean condition expression | a Simulink.Variant object representing a boolean condition expression | a Simulink.Parameter object (required for code generation) | enum

To enter a Variant name, double-click a **Variant control** cell in a new row and type in the Variant control expression.

To enter non-numeric Variant control values, use enumerated data. For information about using enumerated data, see "Use Enumerated Data in Simulink Models"

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: `'variant'`

`Condition (read-only)` — Condition for Variant controls

`''` (default)

This read-only field is based on the condition for the associated Variant control in the global workspace. Create or change a Variant condition in the `Simulink.Variant` parameter dialog box or in the global workspace.

`Label mode active choice` — Name of Active choice if Label mode is selected

`off` (default) | `on`

When you select the **Variant control mode** as `Label`, the **Label mode active choice** option is available. You can select an active Variant choice from **Label mode active**

choice options. You can also right-click the badge on the Variant Subsystem block and select **Label Mode Active Choice**.

For **Label mode active choice** 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 **Label mode active choice** drop-down list to see a list of Variant controls that are specified in the Variant choice section.

Note **Label mode active choice** option is not available in `Expression` mode.

Dependencies

To enable this parameter, select `Label` option from **Variant control mode** parameter.

Programmatic Use

Parameter: `LabelModeActivechoice`
Type: character vector
Value: `''` if no Label mode active choice is specified, the value is empty or the name of the Label mode active choice.
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 if there is no active Variant choice, Simulink disables all the blocks connected to the input and output stream of Variant Subsystem 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

- The (default) option of Variant is not selected

- `Expression` option from **Variant control mode** is selected.

Programmatic Use
Parameter: `AllowZeroVariantControls`
Type: character vector
Value: `'off'` `'on'`
Default: `'off'`

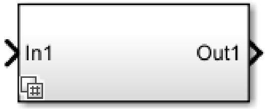
`Analyze all choices during update diagram and generate preprocessor conditionals` — Generate preprocessor conditionals

`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.
- `Expression` option from **Variant control mode** is selected.

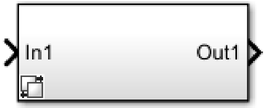
Programmatic Use
Parameter: `GeneratePreprocessorConditionals`
Type: character vector
Value: `'off'` `'on'`
Default: `'off'`

`Propagate conditions outside of Variant Subsystem` — Propagate Variant conditions outside of Variant Subsystem block

`off` (default) | `on`

When you select this option, Simulink propagates the Variant conditions outside of the Variant Subsystem 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'`

Block Characteristics

Data Types	`double`[a] | `single`[a] | `Boolean`[a] | `base integer`[a] | `fixed point`[a] | `enumerated`[a] | `bus`[a] | `string`[a]
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`[a]
Variable-Size Signals	`Yes`[a]
Zero-Crossing Detection	`No`

[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™.

Actual data type or capability support depends on block implementation.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

See Also

Inport | Outport | `Simulink.Variant`

Topics

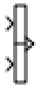
“Variant Systems” (Embedded Coder)
“Define, Configure, and Activate Variants”
“Working with Variant Choices”
“Introduction to Variant Controls”
“Create Variant Controls Programmatically”

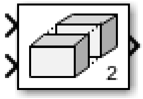
Introduced in R2010b

Vector Concatenate, Matrix Concatenate

Concatenate input signals of same data type to create contiguous output signal

Library: Simulink / Commonly Used Blocks
Simulink / Math Operations
Simulink / Signal Routing





Description

The Concatenate block concatenates the input signals 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

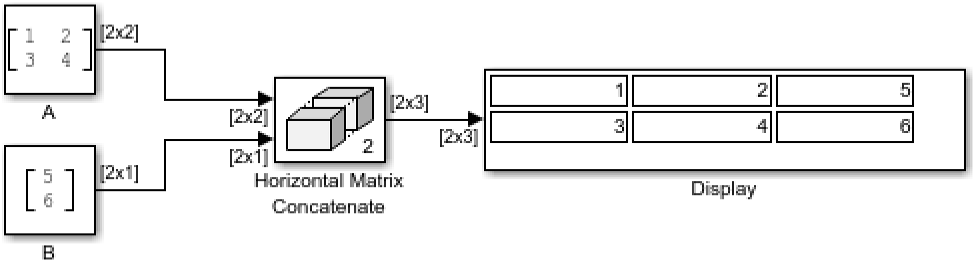
In vector mode, all input signals must be either vectors or row vectors (1-by-M matrices) or column vectors (M-by-1 matrices) or a combination of vectors and either row or column vectors. When all inputs are vectors, the output is a vector.

If any of the inputs are row or column vectors, the output is a row or column vector, respectively.

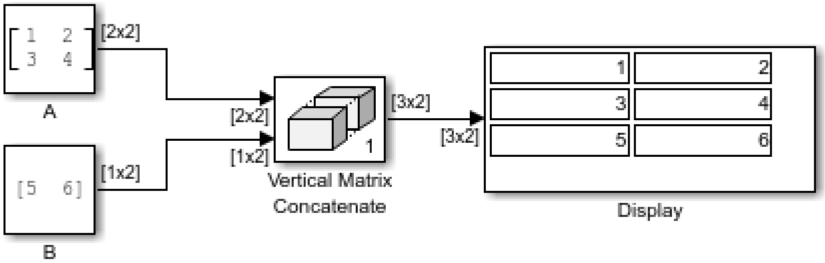
Multidimensional Array Mode

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 `[2x3]` (2-D), this block treats the input as `[2x3x1x1]`. The output is always an array. The **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, see the ex_concatenate_horizontal model:

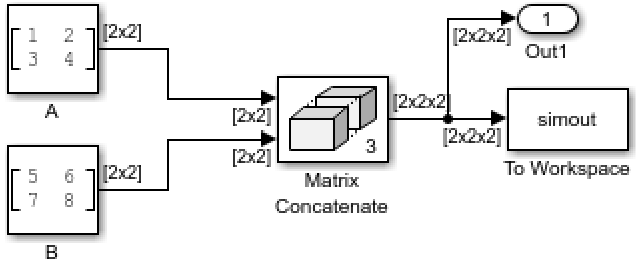


If you set the **Concatenate dimension** parameter to `1` and inputs are 2-D matrices, the block performs vertical matrix concatenation and stacks the input matrices on top of each other to create the output matrix. For example, see the ex_concatenate_vertical model:



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, see the ex_concatenate_multidims model:



Ports

Input

`Port_1` — First input to concatenate

scalar | vector | matrix | N-D array

First input to concatenate, specified as a scalar, vector, matrix, or N-D array.

Dependencies

- Inputs must be of the same data type.
- Matrix and N-D array inputs are supported only when you set **Mode** to `Multidimensional array`.

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

`Port_N` — Nth input to concatenate

scalar | vector | matrix | N-D array

Nth input to concatenate, specified as a scalar, vector, matrix, or N-D array.

Dependencies

- To enable this port, set **Number of inputs** to an integer greater than or equal to 2.
- Inputs must be of the same data type.
- Matrix and N-D array inputs are supported only when you set **Mode** to `Multidimensional array`.

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

Output

`Port_1` — Concatenation of input signals

scalar | vector | matrix | N-D array

Concatenation of input signals, along specified dimension. Outputs have the same data type as the input.

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

Parameters

`Number of inputs` — Number of input ports

2 (default) | positive integer

Specify the number of inputs for the block as a real-valued, positive integer, less than or equal to 65536.

Programmatic Use

Block Parameter: `NumInputs`
Type: character vector
Values: positive integer
Default: `'2'`

`Mode` — Type of concatenation

`Vector` | `Multidimensional array`

Select the type of concatenation that this block performs. The default **Mode** of the Vector Concatenate block is `Vector`. The default **Mode** of the Matrix Concatenate block is `Multidimensional array`.

- When you select `Vector` the block performs vector concatenation (see “Vector Mode” on page 1-2187 for details).
- When you select `Multidimensional array`, the block performs matrix concatenation (see “Multidimensional Array Mode” on page 1-2188 for details).

Programmatic Use

Block Parameter: `Mode`
Type: character vector
Values: `'Vector' | 'Multidimensional array'`
Default: `'Vector'`

`Concatenate dimension` — Output dimension along which to concatenate input arrays

1 (default) | scalar integer

Specify the output dimension along which to concatenate the input arrays.

- To concatenate input arrays vertically, enter `1`.
- To concatenate input arrays horizontally, enter `2`.
- To perform multidimensional concatenation on the inputs, specify an integer greater than 2.

Dependencies

To enable this parameter, set **Mode** to `Multidimensional array`.

Programmatic Use

Block Parameter: `ConcatenateDimension`
Type: character vector
Values: scalar integer
Default: `'1'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point` | `enumerated`
Direct Feedthrough	`No`

Multidimensional Signals	`No`
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™.

For more information about HDL code generation, see Vector Concatenate.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Mux | `cat`

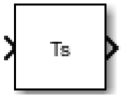
Topics

"Combine Buses into an Array of Buses"
"Creating, Concatenating, and Expanding Matrices" (MATLAB)

Introduced in R2009b

Weighted Sample Time

Support calculations involving sample time
Library: Simulink / Signal Attributes



Description

The Weighted Sample Time block outputs the weighted sample time or weighted sample rate. Because the Weighted Sample Time block is an implementation of the Weighted Sample Time Math, you can also add, subtract, multiply, or divide the input signal, `u`, by a weighted sample time, `Ts`. If the input signal is continuous, `Ts` is the sample time of the Simulink model. Otherwise, `Ts` 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. The block can output just a weighted sample time (`Ts Only`) or a weighted sample rate (`1/Ts Only`).

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.

Tip You can use the Weighted Sample Time and Weighted Sample Time Math blocks to extract the sample time from a Simulink signal. To do so, set the **Operation** parameter to `Ts` and the **Weight value** to `1.0`. In this configuration, the block outputs the sample time of the input signal.

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 + (Ts * w)
```

However, if the **Operation** parameter specifies `/`, the block calculates output using this equation:

```
(u / Ts) / w
```

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Specify the input signal, `u` as a scalar, vector, or matrix. Depending on the value of the **Operation** parameter, the block can add, subtract, multiply or divide the input signal by weighted sample time or just output the weighted sample time or weighted sample rate.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output the weighted sample time or sample rate of the input signal, or output the input signal adjusted by the weighted sample time, `Ts`. If the input signal is continuous, `Ts` is the sample time of the Simulink model. Otherwise, `Ts` is the sample time of the discrete input signal. When the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

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

Parameters

Main

`Operation` — Math operation

`Ts Only` (default) | `+` | `-` | `*` | `/` | `1/Ts Only`

Specify the operation to use for adjusting the input signal. You can select: `+`, `-`, `*`, `/`, `Ts Only`, or `1/Ts Only`.

Programmatic Use
Block Parameter: `TsamMathOp`
Type: character vector
Values: `'+' | '-' | '*' | '/' | 'Ts Only' | '1/Ts Only'`
Default: `'Ts Only'`

`Weight value` — Weight of sample time

`1.0` (default) | real-valued scalar

Enter the weight of the sample time as a real-valued scalar.

Programmatic Use
Block Parameter: `weightValue`
Type: character vector
Values: real-valued scalar
Default: `'1.0'`

`Implement using` — Method for adjusting sample time

`Online Calculations` (default) | `Offline Scaling Adjustment`

Select one of two modes: online calculations or offline scaling adjustment.

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.

Dependencies

To enable this parameter, set **Operation** to `*` or `/`.

Programmatic Use

Block Parameter: `TsampMathImp`
Type: character vector
Values: `'Online Calculations' | 'Offline Scaling Adjustment'`
Default: `'Online Calculations'`

Signal Attributes

`Output data type` — Data type of output signal

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `<data type expression>`

Specify the data type for the output.

Programmatic Use

Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Inherit via back propagation'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Dependencies

To enable this parameter, set **Operation** to `+`, `-`, `*`, or `/`. If you set the **Operation** parameter to `*` or `/`, you must also set **Implement using** to `Online Calculations`.

Programmatic Use

Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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 (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Dependencies

To enable this parameter, set **Operation** to `+`, `-`, `*`, or `/`. If you set the **Operation** parameter to `*` or `/`, you must also set **Implement using** to `Online Calculations`.

Programmatic Use

Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`Yes`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Probe | Weighted Sample Time Math

Topics

“What Is Sample Time?”
“Specify Sample Time”
“View Sample Time Information”

Introduced before R2006a

Waveform Generator

Output waveforms using signal notations

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 “Algorithms” on page 1-2210).
- 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 “Algorithms” on page 1-2210.

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Supported Operators

Operation	Operator
Absolute value	`abs()`
Addition	`+`
Division	`/`
Multiplication	`*`
Parentheses	`()`
Subtraction	`-`
Unary minus	`-`

The Waveform block observes the following rules of operator precedence:

1 `( )`

2 `+ -` (unary)

3 `* /`

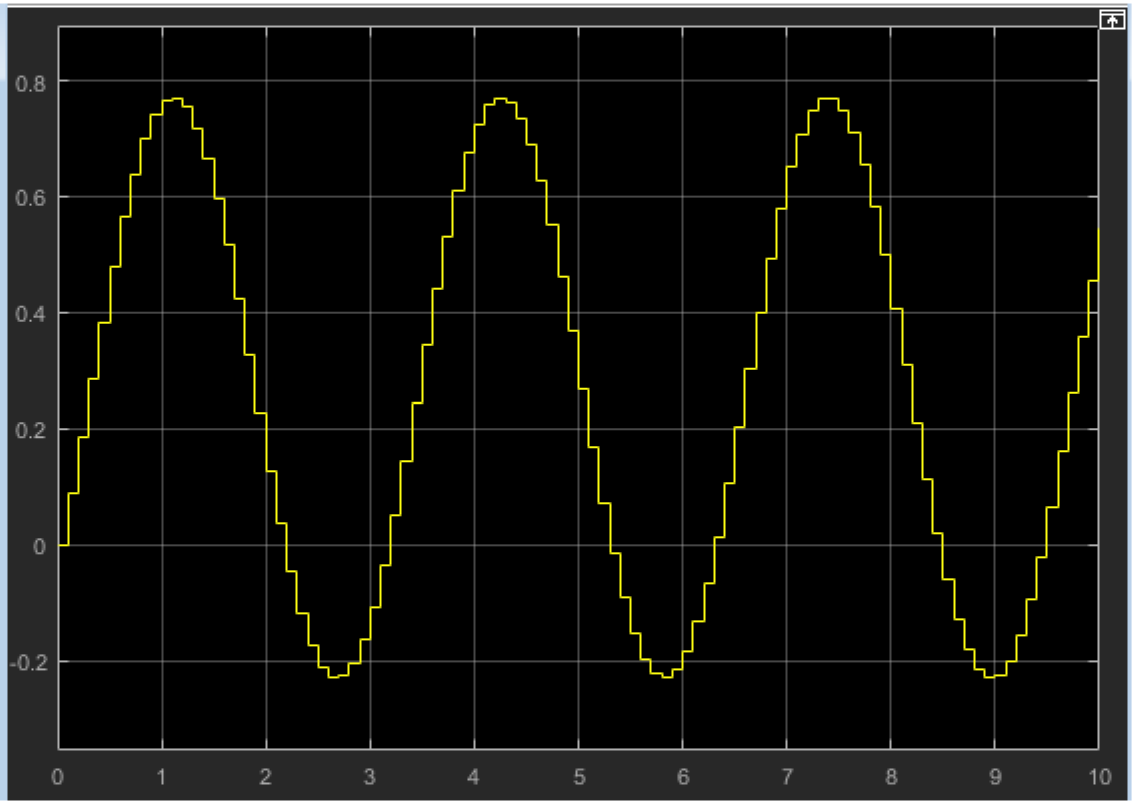
4 `+ -`

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 the Algorithms section.

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.

Limitations

- You cannot tune the parameters of a waveform, such as frequency or amplitude, during execution of the code that you generate by using Simulink Coder. Instead, you can generate code that enables you to switch between waveform variants that you specify. For more information, see “Switch Between Output Waveforms During Code Execution for Waveform Generator Block” (Simulink Coder).

Ports

Output

`Port_1` — Generated output signal

scalar | vector

Output signal specified by an entry in the **Waveform Definition** table.

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

Parameters

Main

`Output Signal` — Waveform for output signal

1 (default) | integer

Select waveform definition to specify the output signal. The number corresponds to the line item in the **Waveform Definition** table. You can change this parameter while a simulation is running.

Programmatic Use
Block Parameter: `SelectedSignal`
Type: character vector
Values: scalar
Default: `'1'`

`Waveform Definition` — Waveform signal notations

constant | `gaussian(mean,variance,seed)` | `pulse(amplitude,trigger_time,duration)` | `sawtooth(amplitude,frequence,phase_offset)` | `sin(amplitude,frequence,phase_offset)` | `square(amplitude,frequence,phase_offset)` | `step(step_time,initial_value,final_value)`

Enter signal notations in the **Waveform Definition** table, one waveform definition per line. For syntax details, see Algorithms.

Signal Attributes

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

`Output minimum` — Minimum output value for range checking

`[]` (default) | scalar

Lower value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMin`
Type: character vector
Values: `'[ ]'`| scalar
Default: `'[ ]'`

`Output maximum` — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

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” (Simulink Coder).

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: `OutMax`
Type: character vector

Values: `'[ ]'` | scalar
Default: `'[ ]'`

`Output data type` — Specify the output data type

`double` (default) | `Inherit: Inherit via back propagation` | `single` | `int8` | `int32` | `uint32` | `fixdt(1,16,2^0,0)` | `<data type expression>` | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: , `'Inherit: Inherit via back propagation'`, `'single'`, `'int8'`, `'uint8'`, `int16`, `'uint16'`, `'int32'`, `'uint32'`, `fixdt(1,16,0)`, `fixdt(1,16,2^0,0)`, `fixdt(1,16,2^0,0)`. `'<data type expression>'`
Default: `'Double'`

`Lock output data type setting against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `LockScale`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select to lock 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).

Programmatic Use
Block Parameter: `LockScale`

Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Specify the rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Choose one of these rounding modes.

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.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

See Also

For more information, see “Rounding” (Fixed-Point Designer).

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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	Rationale	Impact on Overflows	Example
Do not select this check box (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

`Sample time` — Time interval between samples

`0.1` (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar

Default: `'0.1'`

Algorithms

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

Constant values can be:

- Numbers
- Workspace variables
  - Scalar, real variables only
- Built-in MATLAB constant, `pi`

- `1`
- `1.1`
- `x`
- `pi`

Gaussian Noise

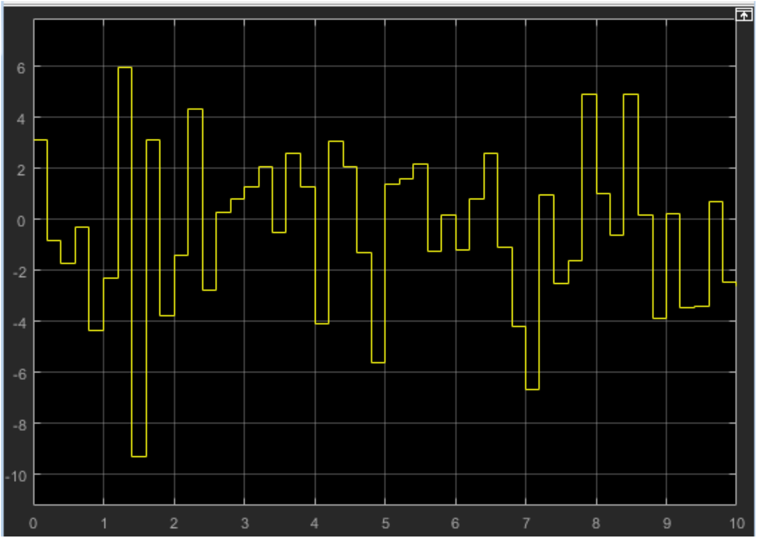
```
gaussian(mean,variance,seed)
```

```
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

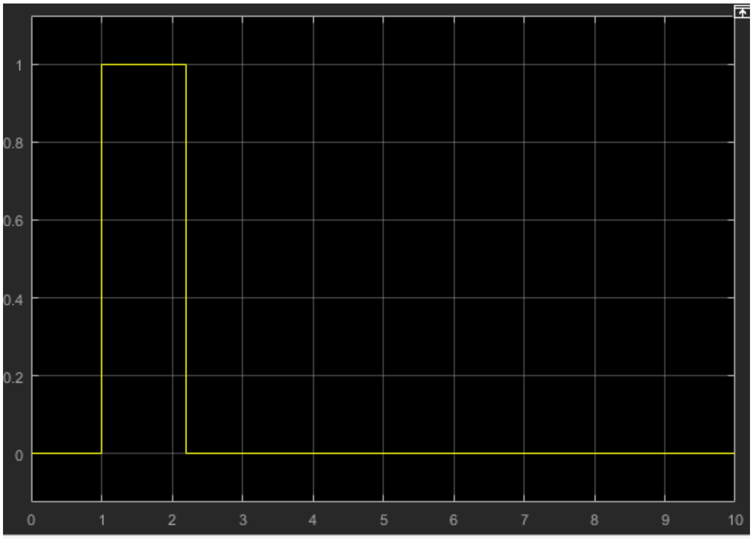
```
pulse(amplitude,trigger_time,duration)
```

```
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)
```



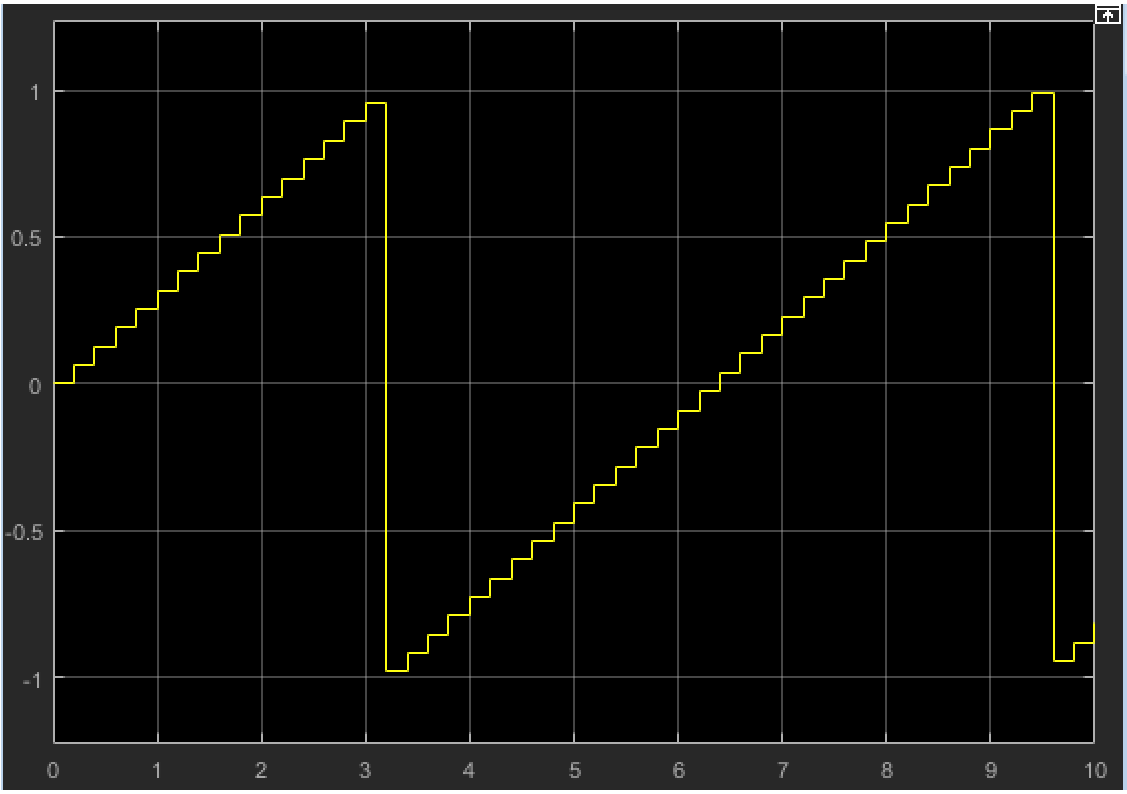
Sawtooth

```
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)
```



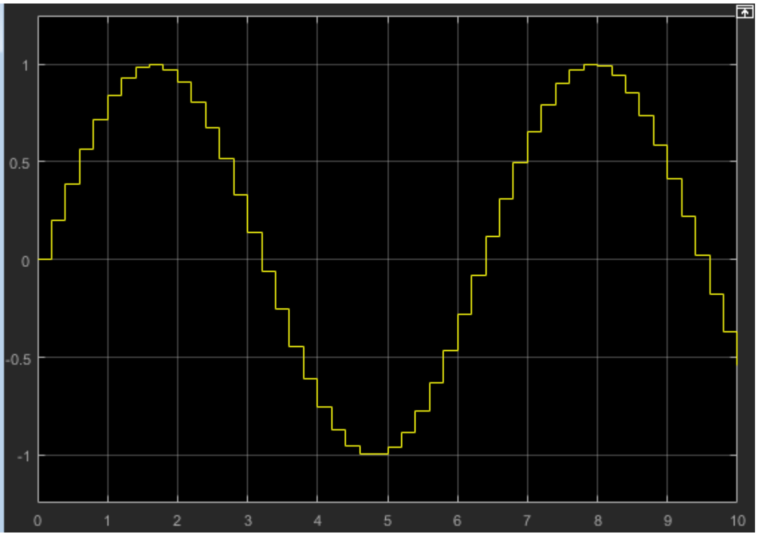
Sine Wave

```
sin(amplitude,frequency,phase_offset)
```

```
sin('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_offset)
```

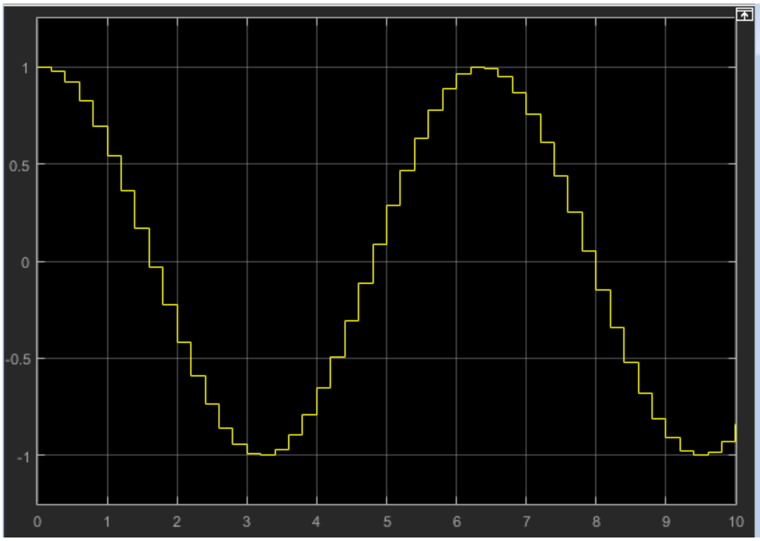
- `amplitude` — Amplitude of sine wave.
  - Default: 1
- `frequency` — Waveform frequency, in rad/s.
  - Default: 1
- `phase_offset` — Phase offset, in rads.
  - Default: 0

```
sin('Amplitude',1,'Frequency',1,'Phase',0)
```



To get the cosine waveform:

```
sin('Amplitude',1,'Frequency',1,'Phase',pi/2)
```

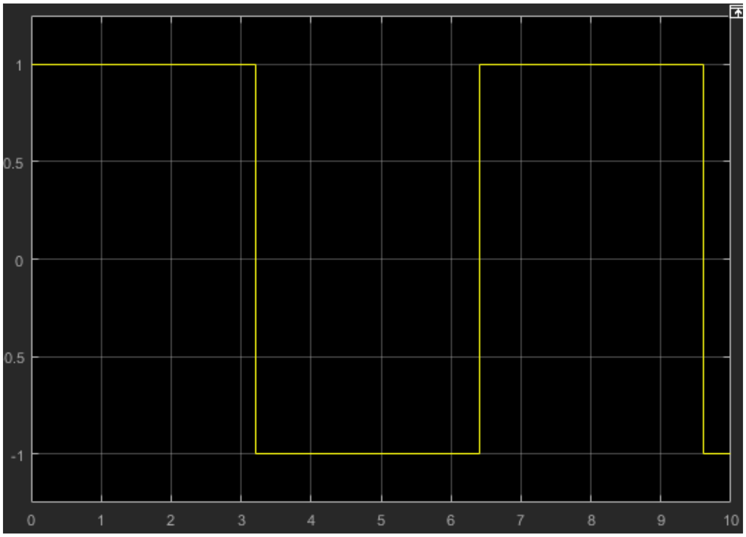


Square

```
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

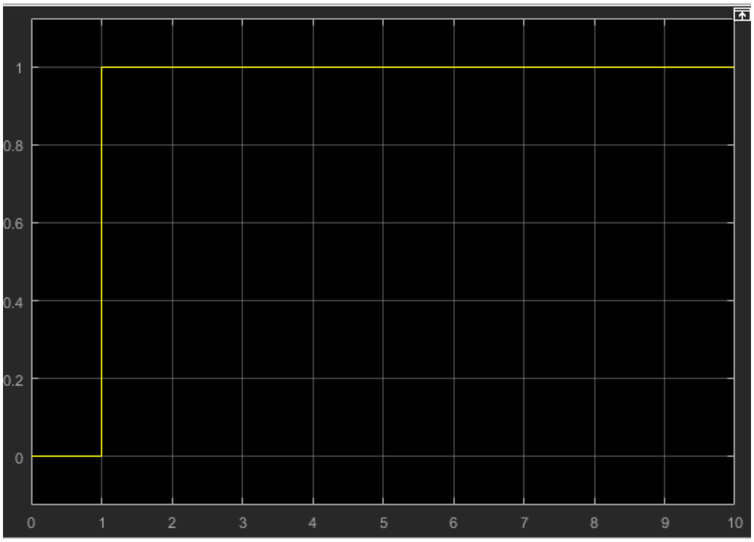
```
step(step_time,initial_value,final_value)
```

```
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)
```



See Also

Repeating Sequence | Signal Builder

Topics

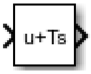
"Signal Basics"

Introduced in R2015b

Weighted Sample Time Math

Support calculations involving sample time

Library: Simulink / Math Operations



Description

The Weighted Sample Time Math block adds, subtracts, multiplies, or divides its input signal, `u`, by a weighted sample time, `Ts`. If the input signal is continuous, `Ts` is the sample time of the Simulink model. Otherwise, `Ts` 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. The block can output just a weighted sample time (`Ts Only`) or a weighted sample rate (`1/Ts Only`).

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.

Tip You can use the Weighted Sample Time and Weighted Sample Time Math blocks to extract the sample time from a Simulink signal. To do so, set the **Operation** parameter to `Ts` and the **Weight value** to `1.0`. In this configuration, the block outputs the sample time of the input signal.

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 + (Ts * w)
```

However, if the **Operation** parameter specifies `/`, the block calculates output using this equation:

```
(u / Ts) / w
```

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

Specify the input signal, `u` as a scalar, vector, or matrix. Depending on the value of the **Operation** parameter, the block can add, subtract, multiply or divide the input signal by weighted sample time or just output the weighted sample time or weighted sample rate.

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

Output

`Port_1` — Output signal

scalar | vector | matrix

Output the weighted sample time or sample rate of the input signal, or output the input signal adjusted by the weighted sample time, `Ts`. If the input signal is continuous, `Ts` is the sample time of the Simulink model. Otherwise, `Ts` is the sample time of the discrete input signal. When the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

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

Parameters

Main

`Operation` — Math operation

`+` (default) | `-` | `*` | `/` | `Ts Only` | `1/Ts Only`

Specify the operation to use for adjusting the input signal. You can select: `+`, `-`, `*`, `/`, `Ts Only`, or `1/Ts Only`.

Programmatic Use

Block Parameter: `TsamMathOp`
Type: character vector
Values: `'+'` | `'-'` | `'*'` | `'/'` | `'Ts Only'` | `'1/Ts Only'`
Default: `'+'`

`Weight value` — Weight of sample time

`1.0` (default) | real-valued scalar

Enter the weight of the sample time as a real-valued scalar.

Programmatic Use

Block Parameter: `weightValue`
Type: character vector
Values: real-valued scalar
Default: `'1.0'`

`Implement using` — Method for adjusting sample time

`Online Calculations` (default) | `Offline Scaling Adjustment`

Select one of two modes: online calculations or offline scaling adjustment.

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.

Dependencies

To enable this parameter, set **Operation** to `*` or `/`.

Programmatic Use
Block Parameter: `TsampMathImp`
Type: character vector
Values: `'Online Calculations' | 'Offline Scaling Adjustment'`
Default: `'Online Calculations'`

Signal Attributes

`Output data type` — Data type of output signal

`Inherit: Inherit via internal rule` (default) | `Inherit: Inherit via back propagation` | `<data type expression>`

Specify the data type for the output.

Programmatic Use
Block Parameter: `OutDataTypeStr`
Type: character vector
Values: `'Inherit: Inherit via internal rule` | `'Inherit: Inherit via back propagation'` | `'<data type expression>'`
Default: `'Inherit: Inherit via internal rule'`

`Integer rounding mode` — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

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

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.

Dependencies

To enable this parameter, set **Operation** to `+`, `-`, `*`, or `/`. If you set the **Operation** parameter to `*` or `/`, you must also set **Implement using** to `Online Calculations`.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

`Saturate on integer overflow` — Method of overflow action

`off` (default) | `on`

Specify whether overflows saturate or wrap.

Action	Rationale	Impact on Overflows	Example
Select this check box (`on`).	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 `int8` (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 (`off`).	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (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 `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, 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.

Dependencies

To enable this parameter, set **Operation** to `+`, `-`, `*`, or `/`. If you set the **Operation** parameter to `*` or `/`, you must also set **Implement using** to `Online Calculations`.

Programmatic Use
Block Parameter: `SaturateOnIntegerOverflow`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Probe | Weighted Sample Time

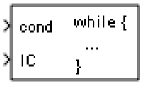
Topics

“What Is Sample Time?”
“Specify Sample Time”
“View Sample Time Information”

Introduced before R2006a

While Iterator

Repeat execution of a subsystem while a logical expression is true
Library: Ports & Subsystems



Description

The While Iterator block, when placed in a Subsystem block, repeatedly executes the contents of the subsystem during the current time step while the value of the input condition is true or `1`. Use this block to implement the block diagram equivalent of a `while` loop in a programming language.

The While Iterator Subsystem block is preconfigured with a While Iterator block. 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
Values: `'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
Values: `'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 and before the first iteration loop.

Programmatic Use
Block Parameter: `ResetStates`
Type: character vector
Values: `'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.

Dependencies

Selecting this parameter enables the **Output data type** parameter.

Programmatic Use
Block Parameter: `ShowIterationPort`
Type: character vector
Values: `'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.

Dependencies

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'`

See Also

Blocks

Subsystem | While Iterator Subsystem

Topics

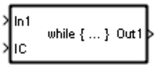
Iterator Subsystem Execution

Introduced before R2006a

While Iterator Subsystem

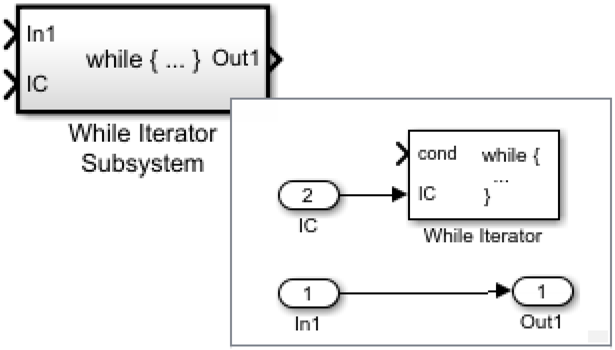
Subsystem that repeats execution during a simulation time step

Library: Simulink / 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 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

`In1` — 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`

Block Characteristics

Data Types	double[a] | single[a] | Boolean[a] | base integer[a] | fixed point[a] | enumerated[a] | bus[a] | string[a]
Direct Feedthrough	No
Multidimensional Signals	Yes[a]
Variable-Size Signals	Yes[a]
Zero-Crossing Detection	No

[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.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks
Subsystem | While Iterator

Topics

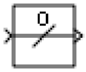
Iterator Subsystem Execution

Introduced before R2006a

Width

Output width of input vector

Library: Simulink / 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”.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix | N-D array

Input signal specified as a scalar, vector, matrix, or N-D array.

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

Output

`Port_1` — Width of input signal

scalar

Output is the width of the input signal, specified as a scalar.

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

Parameters

Output data type mode — Output data type mode
Choose intrinsic data type (default) | Inherit via back propagation | All ports same datatype

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.

Programmatic Use
Block Parameter: OutputDataTypeScalingMode
Type: character vector
Values: 'Choose intrinsic data type' | 'Inherit via back propagation' | 'All ports same datatype'
Default: 'Choose intrinsic data type'

Output data type — Output data type
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32

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.

Programmatic Use
Block Parameter: DataType
Type: character vector
Values: 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32'
Default: 'double'

Block Characteristics

Data Types	double | single | Boolean | base integer | fixed point | enumerated | bus | string

Direct Feedthrough	`Yes`
Multidimensional Signals	`Yes`
Variable-Size Signals	`Yes`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Probe

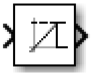
Topics

“Variable-Size Signal Basics”

Introduced before R2006a

Wrap To Zero

Set output to zero if input is above threshold
Library: Simulink / Discontinuities



Description

The Wrap To Zero block sets the output to zero when the input is above the **Threshold** value. When the input is less than or equal to the **Threshold**, then the output is equal to the input.

Ports

Input

`Port_1` — Input signal
scalar | vector

Example:

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

Output

`Port_1` — Output signal
scalar | vector

Output signal set to the input signal value or zero. The data type of the output is the same data type as the input.

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`.

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

Parameters

To edit the parameters for the Wrap to Zero block, double-click the block icon.

`Threshold` — Threshold for outputting zero

`255` (default) | scalar

Threshold value for setting the output value to zero.

Programmatic Use

Block Parameter: `Threshold`
Type: character vector
Values: `scalar`
Default: `'255'`

Block Characteristics

Data Types	`double` | `single` | `Boolean` | `base integer` | `fixed point`
Direct Feedthrough	`No`
Multidimensional Signals	`Yes`
Variable-Size Signals	`No`
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™.

For more information on HDL code generation, see Wrap To Zero.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

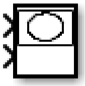
See Also

Saturation | Saturation Dynamic

Introduced before R2006a

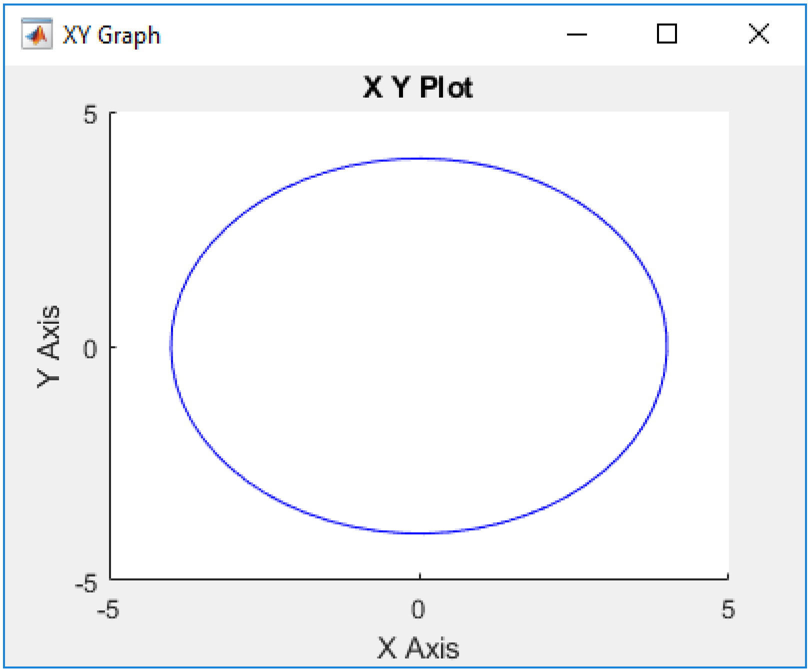
XY Graph

Display X-Y plot of signals using MATLAB figure window
Library: Simulink / 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 from the first input (the *x* direction) against data from 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.

Note The XY Graph block does not support stepping back in a simulation.

Ports

Input

`Port_1` — X-axis values

scalar

Plot input as *x* values on an X-Y plot. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

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

`Port_2` — Y-axis values

scalar

Plot input as *y* values on an X-Y plot. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

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

Parameters

`X-min` — Minimum x

`-1` (default) | real number

Specify the minimum *x*-axis value. Data below the minimum *x* is ignored.

Programmatic Use
Block Parameter: `xmin`

Type: character vector
Values: real number
Default: `'-1'`

`X-max` — Maximum x

`1` (default) | real number

Specify the maximum *x*-axis value. Data above the maximum *x* is ignored.

Programmatic Use
Block Parameter: `xmax`
Type: character vector
Values: real number
Default: `'1'`

`Y-min` — Minimum y

`-1` (default) | real number

Specify the minimum *y*-axis value. Data below the minimum *y* is ignored.

Programmatic Use
Block Parameter: `ymin`
Type: character vector
Values: real number
Default: `'-1'`

`Y-max` — Maximum y

`1` (default) | real number

Specify the maximum *y*-axis value. Data above the maximum *y* is ignored.

Programmatic Use
Block Parameter: `ymax`
Type: character vector
Values: real number
Default: `'1'`

`Sample time` — Sample time

`-1` (default) | positive number

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: `st`
Type: character vector
Values: `'-1'` (for inherited) | positive number
Default: `'-1'`

Block Characteristics

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

Extended Capabilities

HDL Code Generation

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

For more information about HDL code generation, see XY Graph.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Scope

Topics

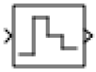
“Decide How to Visualize Simulation Data”

Introduced before R2006a

Zero-Order Hold

Implement zero-order hold sample period

Library: Simulink / Discrete



Description

The Zero-Order Hold block holds its input for the sample period you specify. 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.

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”.

Comparison with Similar Blocks

The Memory, Unit Delay, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different.

This table shows 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.	• `sldemo_enginewc` (Compression subsystem)
Memory on page 1-1222	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.	• `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.	• `sldemo_radar_eml` • `aero_dap3dof`

Each block has the following capabilities.

Capability	Memory	Unit Delay	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	No, because the block can only inherit sample time from the driving block or the solver used for the entire model.	Yes	Yes
Support for frame-based signals	No	Yes	Yes
Support for state logging	No	Yes	No

Ports

Input

`Port_1` — Input signal

scalar | vector

Input signal that the block holds by one sample period.

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

Output

`Port_1` — Output signal

scalar | vector

Output signal that is the input held by one sample period.

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

Parameters

`Sample time (-1 for inherited)` — Discrete interval between sample time hits

`-1` (default) | `scalar`

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.

Block Characteristics

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

Extended Capabilities

C/C++ Code Generation

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

Generated code relies on `memcpy` or `memset` functions (`strong.h`) under certain conditions.

HDL Code Generation

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

For information about HDL code generation, see Zero-Order Hold.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

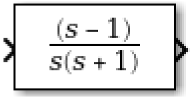
Memory | Unit Delay

Introduced before R2006a

Zero-Pole

Model system by zero-pole-gain transfer function

Library: Simulink / 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

  $$H(s)=K\frac{Z(s)}{P(s)}=K\frac{(s-Z(1))(s-Z(2))\ldots(s-Z(m))}{(s-P(1))(s-P(2))\ldots(s-P(n))},$$

  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**.

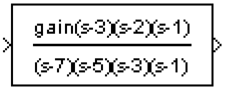
Each element of the output vector corresponds to a column in **Zeros**.

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.

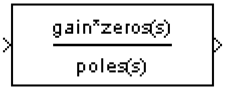
- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified zeros, poles, and gain. If you specify a variable in parentheses, the block evaluates the variable.

  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.



- 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.



Ports

Input

`Port_1` — Input signal

scalar

Input signal, specified as a scalar with data type `double`.

Data Types: `double`

Output

`Port_1` — Output signal

scalar | vector

System modeled by a zero-pole gain transfer function, provided as a scalar or vector signal with data type `double`.

- When modeling a single-output system, the block outputs a scalar time-domain signal. For more information, see “Modeling a Single-Output System” on page 1-2251.
- When modeling a multiple-output system, the block outputs a vector, where each element is an output of the system. For more information, see “Modeling a Multiple-Output System” on page 1-2251.

Data Types: `double`

Parameters

`Zeros` — Matrix of zeros

`[1]` (default) | vector | matrix

Define the matrix of zeros.

- 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.

Programmatic Use
Block Parameter: `Zeros`
Type: character vector, string
Value: vector | matrix
Default: `'[1]'`

`Poles` — Vector of poles

`[0 -1]` (default) | vector

Define the vector of poles.

- 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.

Programmatic Use
Block Parameter: `Poles`
Type: character vector, string
Value: vector
Default: `'[0 -1]'`

`Gain` — Vector of gains

`[1]` (default) | vector

Define the vector of gains.

- 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**.

Programmatic Use

Block Parameter: `Gain`
Type: character vector, string
Value: vector
Default: `'[1]'`

`Absolute tolerance` — Absolute tolerance for computing block states

`auto` (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify `auto` or `-1`.

- 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.
- 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.

Programmatic Use

Block Parameter: `AbsoluteTolerance`
Type: character vector, string
Values: `'auto'` | `'-1'` | any positive real-valued scalar or vector
Default: `'auto'`

`State Name (e.g., 'position')` — Assign unique name to each state

`' '` (default) | `'position'` | `{'a', 'b', 'c'}` | `a` | ...

Assign a unique name to each state. If this field is blank (`' '`), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'position'`.
- 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 in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

Limitations

- 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.

Programmatic Use
Block Parameter: `ContinuousStateAttributes`
Type: character vector, string
Values: `' '` | user-defined
Default: `' '`

Block Characteristics

Data Types	`double`
Direct Feedthrough	`No`
Multidimensional Signals	`No`
Variable-Size Signals	`No`
Zero-Crossing Detection	`No`

Extended Capabilities

C/C++ Code Generation

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

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select **Analysis > Control Design > Model Discretizer**. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also

Discrete Zero-Pole

Topics

"States"

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 1-968.

`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 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 line 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])
```

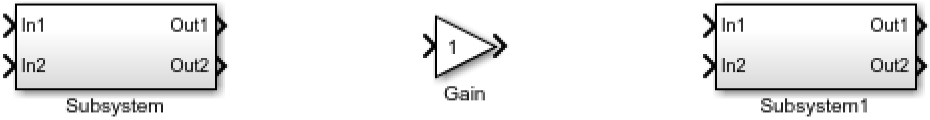
Connect Blocks Using Autorouting Options

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')
```

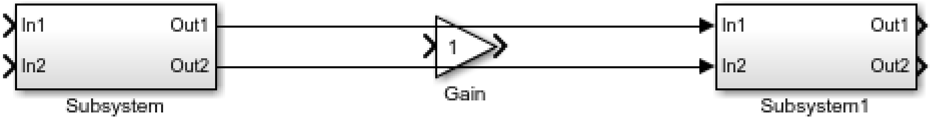
Add blocks as shown. Add an inport and outport to each subsystem.



Add lines to connect the outputs from Subsystem to the inputs of Subsystem1.

```
add_line('route',{'Subsystem/1','Subsystem/2'},...
      {'Subsystem1/1','Subsystem1/2'})
```

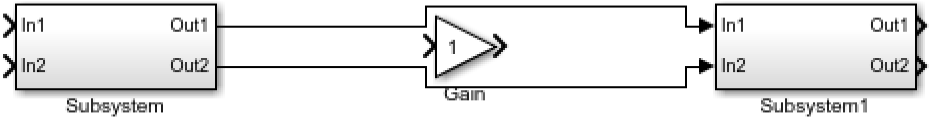
Because you did not use the autorouting options, the function draws straight lines, which pass through the Gain block.



Delete the lines. Add lines again, this time using the autorouting option set to `'on'`.

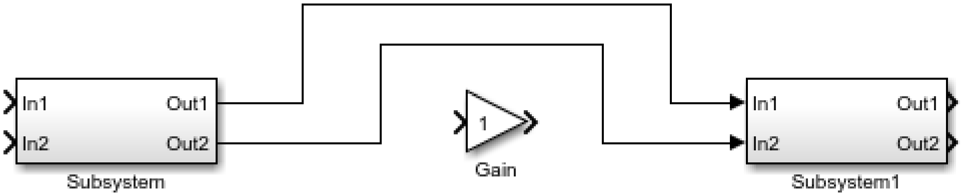
```
add_line('route',{'Subsystem/1','Subsystem/2'},...
      {'Subsystem1/1','Subsystem1/2'},'autorouting','on')
```

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,fileorfolder)
```

Description

`addFile(proj,fileorfolder)` 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.

`fileorfolder` — Path of file or folder

character vector | cell array of character vectors | string array

Path of the file or folder to add relative to the project root folder, specified as a character vector, string, or array. Files must include the file extension. The file or folder 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 | string

Path of the folder to add relative to the project root folder, specified as a character vector or string. 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

batchsim

Offload simulations to run on a compute cluster

Syntax

```
simJob = batchsim(in)
simJob = batchsim(myCluster,in)
simJob = batchsim(...,Name1,Value1,...NameN,ValueN)
```

Description

`simJob = batchsim(in)` runs a batch job on a single worker to simulate a model using the inputs specified in the `SimulationInput` object, `in`.

`simJob = batchsim(myCluster,in)` runs a batch job on the cluster identified by the cluster object `myCluster`. If a cluster profile is not specified, `batchsim` uses a default cluster profile as set up in the parallel preferences. For more information, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).

`simJob = batchsim(...,Name1,Value1,...NameN,ValueN)` runs a batch job that simulates a model using the inputs specified in the `SimulationInput` object and the options specified as `Name,Value` pair.

`batchsim` offloads simulations to a compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later. Use the `'Pool'` argument to run simulations in parallel.

The `batchsim` command uses the Parallel Computing Toolbox™ and MATLAB Distributed Computing Server™ licenses to run the simulations on compute cluster. `batchsim` runs the simulations in serial if a parallel pool cannot be created. If Parallel Computing Toolbox license is not available, `batchsim` errors out.

Examples

Run Parallel Simulations with batchsim

This example shows how to run parallel simulations in batch. batchsim offloads simulations to a compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

This example uses the sldemo_househeat model and runs simulations in batch to observe the model behavior for different temperature set points.

Open the model.

```
open_system('sldemo_househeat');
```

Define a set of values for different temperatures.

```
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

Using the setPointValues, initialize an array of Simulink.SimulationInput objects.

```
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',...
        'Value',num2str(setPointValues(i)));
end
```

Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, let's assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. NumWorkers tells you how many workers are running the simulations. NumWorkers is always the number of workers specified in the 'Pool' argument and an additional head worker.

```
simJob = batchsim(in,'Pool',3)

                  ID: 1
                Type: pool
          NumWorkers: 4
            Username: #####
               State: running
      SubmitDateTime: ##-###-#### ##:##:##
       StartDateTime: 
    Running Duration: 0 days 0h 0m 0s
```

Access the results of the batch job using the `fetchOutputs` method. `fetchOutputs` returns an array of `Simulink.SimulationOuput` objects.

```
out = fetchOutputs(job)
```

```
1x11 Simulink.SimulationOutput array
```

Input Arguments

`in` — `Simulink.SimulationInput` object used to simulate the model
object, array

Specified as a `Simulink.SimulationInput` object or an array of `Simulink.SimulationInput` objects that is used to specify changes to the model for simulation.

Example: `in = Simulink.SimulationInput('vdp')`

`myCluster` — parallel.Cluster object
object

Cluster object that is used to specify the cluster in which the batch job runs.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Pool'`, 5

`AdditionalPaths` — Files to attach to parallel pool
character vector | cell array

Specified as a character vector, a cell array or an array of character vector to define paths to be added to the MATLAB search path of the workers before the simulations execute. The default search path might not be the same on the workers as it is on the client; the path difference could be the result of different current working folders (pwd), platforms, or network file system access. The '`AdditionalPaths`' property can assure that workers are looking in the correct locations for necessary code files, data files, model files, etc.

`AttachedFiles` — Files to attach to parallel pool
cell array

Specified as a cell array of additional files to attach to the parallel pool.

`AutoAddClientPath` — Whether user-added entries on client path are added to each worker path
true (default) | false

Specified as true or false to control whether user-added entries on the client path are added to each worker path.

`AutoAttachedFiles` — Whether code files should be automatically attached to the job
true (default) | false

Specified as true or false to control whether code files are automatically attached to the job.

`CaptureDiary` — Whether `diary` is collected
true (default) | false

Specified as true or false to indicate collection of the `diary`.

`CleanupFcn` — Function handle to run once per worker after running simulations
function handle

Specify a `function handle` to `'CleanupFcn'` to run once per worker after the simulations are completed.

`EnvironmentVariables` — Names of environment variables copied from client session to workers
character vector | cell array

Specifies the names of environment variables copied from the client session to the workers. The names specified here are appended to the '`EnvironmentVariables`' property specified in the applicable parallel profile to form the complete list of environment variables. Any variables listed which are not set are not copied to the workers. These environment variables will be set on the workers for the duration of the batch job.

`ManageDependencies` — Manage model dependencies
'on' (default) | 'off'

When `ManageDependencies` is set to `'on'`, model dependencies are automatically sent to the parallel workers if necessary. If `ManageDependencies` is set to `'off'`, explicitly attach model dependencies to the parallel pool.

`Pool` — Size of the number of workers for a parallel pool

integer

An integer specifying the number of workers to make into a parallel pool for the job *in addition* to the worker running the batch job itself. The simulations use this pool for execution. Because the pool requires N workers in addition to the worker running the batch, there must be at least N+1 workers available on the cluster.

`Profile` — Cluster profile name

profile name

The name of a cluster profile used to identify the cluster. If this option is omitted, the default profile is used to identify the cluster and is applied to the job and task properties.

`SetupFcn` — Function handle to run once per worker

function handle

Specify a `function handle` to `'SetupFcn'` to run once per worker before the start of the simulations.

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.

`ShowProgress` — Show the progress of the simulations in `diary`

'on' | 'off'

Set to `'on'`, to copy the progress of the simulations in the command window to `diary` of `Simulink.Simulation.Job` object. The progress is hidden when set to `'off'`.

`TransferBaseWorkspaceVariables` — Transfer variables to the parallel workers

'off' (default) | 'on'

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.

`UseFastRestart` — Use fast restart

'off' (default) | 'on'

When `UseFastRestart` is set to `true`, simulations run on the workers using fast restart.

Note When using `batchsim`, use the `UseFastRestart` option and not the `FastRestart` option. See “Get Started with Fast Restart” for more information.

Output Arguments

`simJob` — `Simulink.Simulation.Job` job object

object

An object containing metadata of submitted batch job. Poll job object using it's ID to check the status of simulations or to access outputs on completion of the job.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Using `batchsim` with Parallel Computing Toolbox installed, MATLAB automatically opens a worker and runs the job in the background on another session on your local machine. Specifying a pool size runs the simulations on the number of workers specified. Control parallel behavior with the parallel preferences, including scaling up to a cluster.

For details, see “Run Multiple Simulations”.

See Also

Functions
batch | batchsim | cancel | diary | fetchOutputs | getSimulationJobs | listAutoAttachedFiles | parcluster | parsim | wait

Classes
Simulink.Simulation.Job | Simulink.SimulationInput

Topics

"Multiple Simulation Workflows"
"Run Multiple Simulations"
"Batch Processing" (Parallel Computing Toolbox)
"Job Monitor" (Parallel Computing Toolbox)

Introduced in R2018b

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 =

  1×2 logical array

   0   0
```

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 Models and Other Project 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 `hydlib`, 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

Top-level model of current system

Syntax

```
model = bdroot
model = bdroot(elements)
```

Description

`model = bdroot` returns the top-level model of the current system. The current system is the currently active Simulink Editor window or the system in which a block is selected.

`model = bdroot(elements)` returns the top-level model of the specified model elements. Before using `bdroot`, make sure the top-level model of each element in `elements` is loaded.

Examples

Get Top-Level Model of Current System

Open the system Controller in the model `f14`.

```
load_system('f14')
open_system('f14/Controller')
```

Get the top-level model of the current system.

```
bdroot
```

```
ans =

    'f14'
```

Get Top-Level Model of a System

Open the system Controller in the model `f14`.

```
load_system('f14')
open_system('f14/Controller')
```

Get the top-level model of the current system.

```
bdroot(gcs)
```

```
ans =

    'f14'
```

Input Arguments

`elements` — Model elements whose top-level models to return

model name | block path name | handle | cell array of character vectors | string array | numeric array

Model elements whose top-level model to return, specified as the model name, block or system path name, handle, cell array of character vectors or string array of system names, or numeric array of handles.

Tip Use `bdroot` with `gcs`, `gcb`, and `gcbh` to get the top-level model of the current system or block.

Output Arguments

`model` — Top-level model

character vector | cell array | string array

Top-level model, returned as a character vector of the model name. If the input was an array, `model` is returned as an array of the same type as the input.

See Also

`gcb` | `gcbh` | `gcs`

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

sys Name of the Simulink system from which the linear model is extracted.

x, *u* State (*x*) and the input (*u*) 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 `x`. To extract the `x` 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'`.

Ts 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 linmodv5.

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 (t) 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_struc = 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 Ts 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 Ts 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 Ts, provided that

- Ts 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 Ad provides an indication of the stability of the system. The system is stable if Ts>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 Ts = 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 Ad and Bd matrices, which can be complex. The eigenvalues of the Ad 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 *`saveflag`*s 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. Provide the external source, library, and header files to the code generator.

`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`.

```
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`. Provide the source and header files to the code generator in the function.

```
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
    coder.updateBuildInfo('addSourceFiles','foo.c');
    coder.cinclude('foo.h');
    y = coder.ceval('foo', 10, 20);
end
end
```

Generate C library code for function `callfoo`. The `codegen` function generates C code in the `\codegen\lib\callfoo` subfolder.

```
codegen -config:lib callfoo -report
```

Call a C Library Function

Call a C library function from MATLAB code.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)
%#codegen
y = abs(u);
```

Generate a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `\codegen\lib\myabsval`. (The library file extension can change depending on your platform.) It 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(y)
%#codegen
% 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
  % add the required include statements to generated function code
  coder.updateBuildInfo('addIncludePaths','$(START_DIR)\codegen\lib\myabsval');
  coder.cinclude('myabsval_initialize.h');
  coder.cinclude('myabsval.h');
  coder.cinclude('myabsval_terminate.h');

  % 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
```

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

Rather than providing the library at the command line, you can use `coder.updateBuildInfo` to specify the library within the function. Use this option to preconfigure the build. Add this line to the `else` block:

```
coder.updateBuildInfo('addLinkObjects','myabsval.lib','$(START_DIR)\codegen\lib\myabsva
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex(-2.75)
```

```
ans =

    2.7500
```

Call the MATLAB function `callmyabsval`.

```
callmyabsval(-2.75)
```

```
ans =

    2.7500
```

The `callmyabsval` function exhibits the desired behavior for execution in MATLAB and in code generation.

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.ExternalDependency` | `coder.extrinsic` | `coder.opaque` | `coder.ref` | `coder.rref` | `coder.target` | `coder.updateBuildInfo` | `coder.wref`

Topics

“Integrate C Code Using the MATLAB Function Block”
“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`.

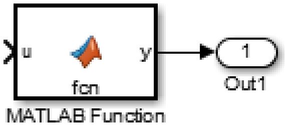
Write a C function `myMult2.c` that doubles its input. Save it in `mycfiles`.

```
#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);
% Specify the locations of myMult2.h and myMult2.c
coder.extrinsic('pwd', 'fullfile');
customDir = coder.const(fullfile(pwd, 'mycfiles'));
coder.updateBuildInfo('addIncludePaths', customDir);
coder.updateBuildInfo('addSourcePaths', customDir);
coder.updateBuildInfo('addSourceFiles', 'myMult2.c');
end
```

Open the Configuration Parameters dialog box.

On the **Solver** pane, select a fixed-step solver.

Save the model as `mymodel`.

Build the model.

The file `mymodel.h` contains this statement:

```
#include "myMult2.h"
```

To read more about integrating custom code in a MATLAB Function block, see “Integrate C Code Using the MATLAB Function Block”.

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

- When possible, the code generator constant-folds expressions automatically. 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.
- When constant-folding computationally intensive function calls, to reduce code generation time, make the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

```
function j = fcn(z)
zTable = coder.const(0:0.01:100);
jTable = coder.const(feval('besselj',3,zTable));
j = interp1(zTable,jTable,z);
end
```

See “Use coder.const with Extrinsic Function Calls” (MATLAB Coder).

- If `coder.const` is unable to constant-fold a function call, try to force constant-folding by making the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

  ```
  function yi = fcn(xi)
  y = coder.const(feval('rand',1,100));
  yi = interp1(y,xi);
  end
  ```

  See “Use coder.const with Extrinsic Function Calls” (MATLAB Coder).

See Also

Topics

“Fold Function Calls into Constants” (MATLAB Coder)
“Use coder.const with Extrinsic Function Calls” (MATLAB Coder)

Introduced in R2013b

coder.cstructname

Package: coder

Name C structure type in generated code

`coder.cstructname` names the generated or externally defined C structure type to use for MATLAB variables that are represented as structures in generated code.

Syntax

```
coder.cstructname(var,structName)
coder.cstructname(var,structName,'extern','HeaderFile',headerfile)
coder.cstructname(var,structName,'extern','HeaderFile',
headerfile,'Alignment',alignment)

outtype = coder.cstructname(intype,structName)
outtype = coder.cstructname(intype,structName,'extern','HeaderFile',
headerfile)
outtype = coder.cstructname(intype,structName,'extern','HeaderFile',
headerfile,'Alignment',alignment)
```

Description

`coder.cstructname(var,structName)` names the C structure type generated for the MATLAB variable `var`. The input `var` can be a structure or a cell array. Use this syntax in a function from which you generate code. Place `coder.cstructname` after the definition of `var` and before the first use of `var`. If `var` is an entry-point (top-level) function input argument, place `coder.cstructname` at the beginning of the function, before any control flow statements.

`coder.cstructname(var,structName,'extern','HeaderFile',headerfile)` specifies that the C structure type to use for `var` has the name `structName` and is defined in the external file, `headerfileName`.

It is possible to use the `'extern'` option without specifying the header file. However, it is a best practice to specify the header file so that the code generator produces the `#include` statement in the correct location.

`coder.cstructname(var,structName,'extern','HeaderFile', headerfile,'Alignment',alignment)` also specifies the run-time memory alignment for the externally defined structure type `structName`. If you have Embedded Coder and use custom Code Replacement Libraries (CRLs), specify the alignment so that the code generator can match CRL functions that require alignment for structures. See “Data Alignment for Code Replacement” (Embedded Coder).

`outtype = coder.cstructname(intype,structName)` returns a structure or cell array type object `outtype` that specifies the name of the C structure type to generate. `coder.cstructname` creates `outtype` with the properties of the input type `intype`. Then, it sets the `TypeName` property to `structName`. Use this syntax to create a type object that you use with the `codegen -args` option. You cannot use this syntax in a function from which you generate code.

You cannot use this syntax in a MATLAB Function block.

`outtype = coder.cstructname(intype,structName,'extern','HeaderFile', headerfile)` returns a type object `outtype` that specifies the name and location of an externally defined C structure type. The code generator uses the externally defined structure type for variables with type `outtype`.

You cannot use this syntax in a MATLAB Function block.

`outtype = coder.cstructname(intype,structName,'extern','HeaderFile', headerfile,'Alignment',alignment)` creates a type object `outtype` that also specifies the C structure type alignment.

You cannot use this syntax in a MATLAB Function block.

Examples

Name the C Structure Type for a Variable in a Function

In a MATLAB function, `myfun`, assign the name `MyStruct` to the generated C structure type for the variable `v`.

```
function y = myfun()
%#codegen
v = struct('a',1,'b',2);
coder.cstructname(v, 'myStruct');
y = v;
end
```

Generate standalone C code. For example, generate a static library.

```
codegen -config:lib myfun -report
```

To see the generated structure type, open `codegen/lib/myfun/myfun_types.h` or view `myfun_types.h` in the code generation report. The generated C structure type is:

```
typedef struct {
  double a;
  double b;
} myStruct;
```

Name the C Structure Type Generated for a Substructure

In a MATLAB function, `myfun1`, assign the name `MyStruct` to the generated C structure type for the structure `v`. Assign the name `mysubStruct` to the structure type generated for the substructure `v.b`.

```
function y = myfun()
%#codegen
v = struct('a',1,'b',struct('f',3));
coder.cstructname(v, 'myStruct');
coder.cstructname(v.b, 'mysubStruct');
y = v;
end
```

The generated C structure type `mysubStruct` is:

```
typedef struct {
  double f;
} mysubStruct;
```

The generated C structure type `myStruct` is:

```
typedef struct {
  double a;
  mysubStruct b;
} myStruct;
```

Name the Structure Type Generated for a Cell Array

In a MATLAB function, `myfun2`, assign the name `myStruct` to the generated C structure type for the cell array`c`.

```
function z = myfun2()
c = {1 2 3};
coder.cstructname(c,'myStruct')
z = c;
```

The generated C structure type for `c` is:

```
typedef struct {
  double f1;
  double f2;
  double f3;
} myStruct;
```

Name an Externally Defined C Structure Type

Specify that a structure passed to a C function has a structure type defined in a C header file.

Create a C header file `mycadd.h` for the function `mycadd` that takes a parameter of type `mycstruct`. Define the type `mycstruct` in the header file.

```
#ifndef MYCADD_H
#define MYCADD_H

typedef struct {
    double f1;
    double f2;
} mycstruct;

double mycadd(mycstruct *s);
#endif
```

Write the C function `mycadd.c`.

```
#include <stdio.h>
#include <stdlib.h>

#include "mycadd.h"

double mycadd(mycstruct *s)
{
  return  s->f1 + s->f2;
}
```

Write a MATLAB function `mymAdd` that passes a structure by reference to `mycadd`. Use `coder.cstructname` to specify that in the generated code, the structure has the C type `mycstruct`, which is defined in `mycadd.h`.

```
function y = mymAdd
%#codegen
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'mycstruct', 'extern', 'HeaderFile', 'mycadd.h');
y = 0;
y = coder.ceval('mycadd', coder.ref(s));
```

Generate a C static library for function `mymAdd`.

```
codegen -config:lib mymAdd mycadd.c
```

The generated header file `mymadd_types.h` does not contain a definition of the structure `mycstruct` because `mycstruct` is an external type.

Create a Structure Type Object That Names the Generated C Structure Type

Suppose that the entry-point function `myFunction` takes a structure argument. To specify the type of the input argument at the command line:

1. Define an example structure `S`.
2. Create a type `T` from `S` by using `coder.typeof`.
3. Use `coder.cstructname` to create a type `T1` that:
   - Has the properties of `T`.
   - Names the generated C structure type `myStruct`.
4. Pass the type to `codegen` by using the `-args` option.

For example:

```
S = struct('a',double(0),'b',single(0));
T = coder.typeof(S);
T1 = coder.cstructname(T,'myStruct');
codegen -config:lib myFunction -args T1
```

Alternatively, you can create the structure type directly from the example structure.

```
S = struct('a',double(0),'b',single(0));
T1 = coder.cstructname(S,'myStruct');
codegen -config:lib myFunction -args T1
```

Input Arguments

var — MATLAB structure or cell array variable

structure | cell array

MATLAB structure or cell array variable that is represented as a structure in the generated code.

structName — Name of C structure type

character vector | string scalar

Name of generated or externally defined C structure type, specified as a character vector or string scalar.

headerfile — Header file that contains the C structure type definition

character vector | string scalar

Header file that contains the C structure type definition, specified as a character vector or string scalar.

To specify the path to the file:

- Use the `codegen -I` option or the **Additional include directories** parameter on the MATLAB Coder app settings **Custom Code** tab.
- For a MATLAB Function block, on the **Simulation Target** and the **Code Generation > Custom Code** panes, under **Additional build information**, set the **Include directories** parameter.

Alternatively, use `coder.updateBuildInfo` with the `'addIncludePaths'` option.

Example: `'mystruct.h'`

`alignment` — Run-time memory alignment for structure
-1 (default) | power of 2 not greater than 128

Run-time memory alignment for generated or externally defined structure.

`intype` — Type object or variable for creation of new type object
`coder.StructType` | `coder.CellType` | structure | cell array

Structure type object, cell array type object, structure variable, or cell array variable from which to create a type object.

Limitations

- You cannot apply `coder.cstructname` directly to a global variable. To name the structure type to use with a global variable, use `coder.cstructname` to create a type object that names the structure type. Then, when you run `codegen`, specify that the global variable has that type. See “Name the C Structure Type to Use With a Global Structure Variable” (MATLAB Coder).
- For cell array inputs, the field names of externally defined structures must be `f1`, `f2`, and so on.

Tips

- For information about how the code generator determines the C/C++ types of structure fields, see “Mapping MATLAB Types to Types in Generated Code” (MATLAB Coder).
- Using `coder.cstructname` on a structure array sets the name of the structure type of the base element, not the name of the array. Therefore, you cannot apply `coder.cstructname` to a structure array element, and then apply it to the array with a different C structure type name. For example, the following code is not allowed. The second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previously specified name, `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');
```

- Applying `coder.cstructname` to an element of a structure array produces the same result as applying `coder.cstructname` to the entire structure array. If you apply `coder.cstructname` to an element of a structure array, you must refer to the element by using a single subscript. For example, you can use `var(1)`, but not `var(1,1)`. Applying `coder.cstructname` to `var(:)` produces the same result as applying `coder.cstructname` to `var` or `var(n)`.
- Heterogeneous cell arrays are represented as structures in the generated code. Here are considerations for using `coder.cstructname` with cell arrays:

  - In a function from which you generate code, using `coder.cstructname` with a cell array variable makes the cell array heterogeneous. Therefore, if a cell array is an entry-point function input and its type is permanently homogeneous, then you cannot use `coder.cstructname` with the cell array.
  - Using `coder.cstructname` with a homogeneous `coder.CellType` object `intype` makes the returned object heterogeneous. Therefore, you cannot use `coder.cstructname` with a permanently homogeneous `coder.CellType` object. For information about when a cell array is permanently homogeneous, see “Specify Cell Array Inputs at the Command Line” (MATLAB Coder).
  - When used with a `coder.CellType` object, `coder.cstructname` creates a `coder.CellType` object that is permanently heterogeneous.
- When you use a structure named by `coder.cstructname` in a project with row-major and column-major array layouts, the code generator renames the structure in certain cases, appending `row_` or `col_` to the beginning of the structure name. This renaming provides unique type definitions for the types that are used in both array layouts.
- These tips apply only to MATLAB Function blocks:

  - MATLAB Function block input and output structures are associated with bus signals. The generated name for the structure type comes from the bus signal name. Do not use `coder.cstructname` to name the structure type for input or output signals. See “Create Structures in MATLAB Function Blocks”.
  - The code generator produces structure type names according to identifier naming rules, even if you name the structure type with `coder.cstructname`. If you have Embedded Coder, you can customize the naming rules. See “Construction of Generated Identifiers” (Embedded Coder).

See Also

`coder.ceval`

Topics

"Structure Definition for Code Generation"
"Code Generation for Cell Arrays"
"Integrate C Code Using the MATLAB Function Block"

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.
- The code generator does not support the use of `coder.extrinsic` to call functions that are located in a private folder.
- The code generator does not support the use of `coder.extrinsic` to call local functions.

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 2-102. `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_myfn`, which calls `myfcn`.

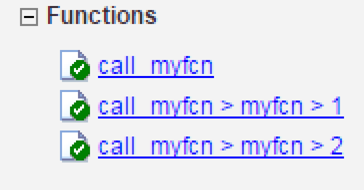
```
function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, 'mode1');
y = myfcn(n, 'mode2');
end

function y = myfcn(n,mode)
coder.inline('never');
if strcmp(mode,'mode1')
    y = n;
else
    y = -n;
```

```
end
end
```

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`.



⊟ Functions

- call_myfcn
- call_myfcn > myfcn > 1
- call_myfcn > myfcn > 2

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.

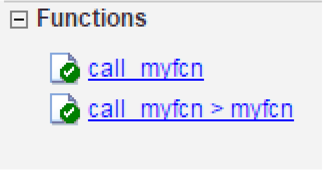
```
function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, coder.ignoreConst('mode1'));
y = myfcn(n, coder.ignoreConst('mode2'));
end

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.



⊟ Functions

call_myfcn

call_myfcn > myfcn

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 2-105 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 2-104
- "Use coder.inline in Control Flow Statements" on page 2-105

Prevent Function Inlining

In this example, function `foo` 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

- 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.

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.

Limitations

- You cannot use `coder.nullcopy` on sparse matrices, or on structures, cell arrays, or classes that contain sparse matrices.

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
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 2-117
- 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`

Topics

"Integrate C Code Using the MATLAB Function Block"

Introduced in R2011a

coder.ref

Indicate data to pass by reference

Syntax

```
coder.ref(arg)
```

Description

`coder.ref(arg)` indicates that `arg` is an expression or variable to pass by reference to an external C/C++ function. 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.

See also `coder.rref` and `coder.wref`.

Examples

Pass Scalar Variable by Reference

Consider the C function `addone` that returns the value of an input plus one:

```
double addone(double* p) {
  return *p + 1;
}
```

The C function defines the input variable `p` as a pointer to a double.

Pass the input by reference to `addone`:

```
...
y = 0;
u = 42;
```

```
y = 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`. This function reads from and writes to the input argument.

```
#include "MyStruct.h"

void incr_struct(struct MyStruct *my_struct)
{
  my_struct->f1 =  my_struct->f1 + 1;
  my_struct->f2 =  my_struct->f2 + 1;
}
```

The C header file, `MyStruct.h`, defines a structure type named `MyStruct`:

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
    double f1;
    double f2;
} MyStruct;

void incr_struct(struct MyStruct *my_struct);

#endif
```

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 the C header file `MyStruct.h`, use `coder.cstructname`.

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','incr_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
coder.ceval('incr_struct', coder.ref(s));
```

To generate standalone library code, enter:

```
codegen -config:lib foo -report
```

Pass Structure Field by Reference

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.ref(s.s1.a));
...
```

You can also pass an element of an array of structures:

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.ref(a.b(3,4).c(2).u));
...
```

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

- If `arg` is an array, then `coder.ref(arg)` provides the address of the first element of the array. The `coder.ref(arg)` function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

  ```
  coder.ceval('myFun',coder.ref(arg),int32(numel(arg));
  ```
- 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` | `numel`

Topics

"Integrate C Code Using the MATLAB Function Block"

Introduced in R2011a

coder.rref

Indicate read-only data to pass by reference

Syntax

`coder.rref(arg)`

Description

`coder.rref(arg)` indicates that `arg` is a read-only expression or variable to pass by reference to an external C/C++ function. Use `coder.rref` only inside a `coder.ceval` call.

The `coder.rref` function can enable the code generator to optimize the generated code. Because the external function is assumed to not write to `coder.rref(arg)`, the code generator can perform optimizations such as expression folding on assignments to `arg` that occur before and after the `coder.ceval` call. Expression folding is the combining of multiple operations into one statement to avoid the use of temporary variables and improve code performance.

Note The code generator assumes that the memory that you pass with `coder.rref(arg)` is read-only. To avoid unpredictable results, the C/C++ function must not write to this variable.

See also `coder.ref` and `coder.wref`.

Examples

Pass Scalar Variable as a Read-Only Reference

Consider the C function `addone` that returns the value of a constant input plus one:

```
double addone(const double* p) {
  return *p + 1;
}
```

The C function defines the input variable `p` as a pointer to a constant double.

Pass the input by reference to `addone`:

```
...
y = 0;
u = 42;
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`. This function reads from the input argument but does not write to it.

```
#include "MyStruct.h"

double use_struct(const struct MyStruct *my_struct)
{
  return my_struct->f1 + my_struct->f2;
}
```

The C header file, MyStruct.h, defines a structure type named MyStruct:

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
    double f1;
    double f2;
} MyStruct;

double use_struct(const struct MyStruct *my_struct);

#endif
```

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 the C header file MyStruct.h, use coder.cstructname.

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','use_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
y = coder.ceval('use_struct', coder.rref(s));
```

To generate standalone library code, enter:

```
codegen -config:lib foo -report
```

Pass Structure Field as a Read-Only Reference

```
...
s = struct('s1', struct('a', [0 1]));
y = coder.ceval('foo', coder.rref(s.s1.a));
...
```

You can also pass an element of an array of structures:

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.rref(a.b(3,4).c(2).u));
...
```

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

- If `arg` is an array, then `coder.rref(arg)` provides the address of the first element of the array. The `coder.rref(arg)` function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

  ```
  coder.ceval('myFun',coder.rref(arg),int32(numel(arg));
  ```

- 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 parametrize 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`

Topics

"Integrate C Code Using the MATLAB Function Block"

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.

Write the function `foo2` and save it in the file `foo2.m`.

```
function out = foo2(in)
  out = eval(in);
end
```

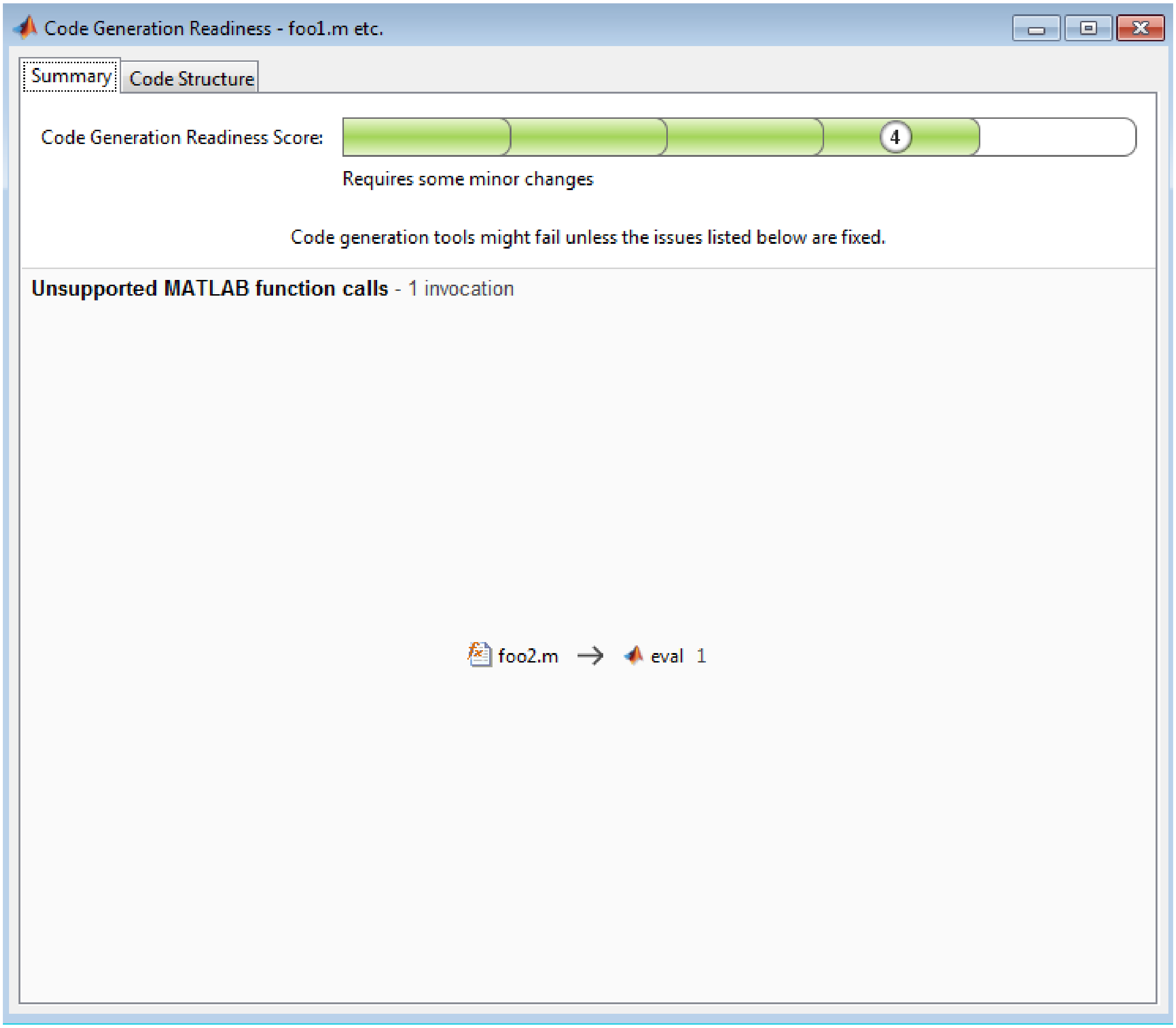
Write the function `foo1` that calls `foo2`. Save `foo1` in the file `foo1.m`.

```
function out = foo1(in)
  out = foo2(in);
  disp(out);
end
```

Analyze `foo1`.

```
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.



Code Generation Readiness Score: 4

Requires some minor changes

Code generation tools might fail unless the issues listed below are fixed.

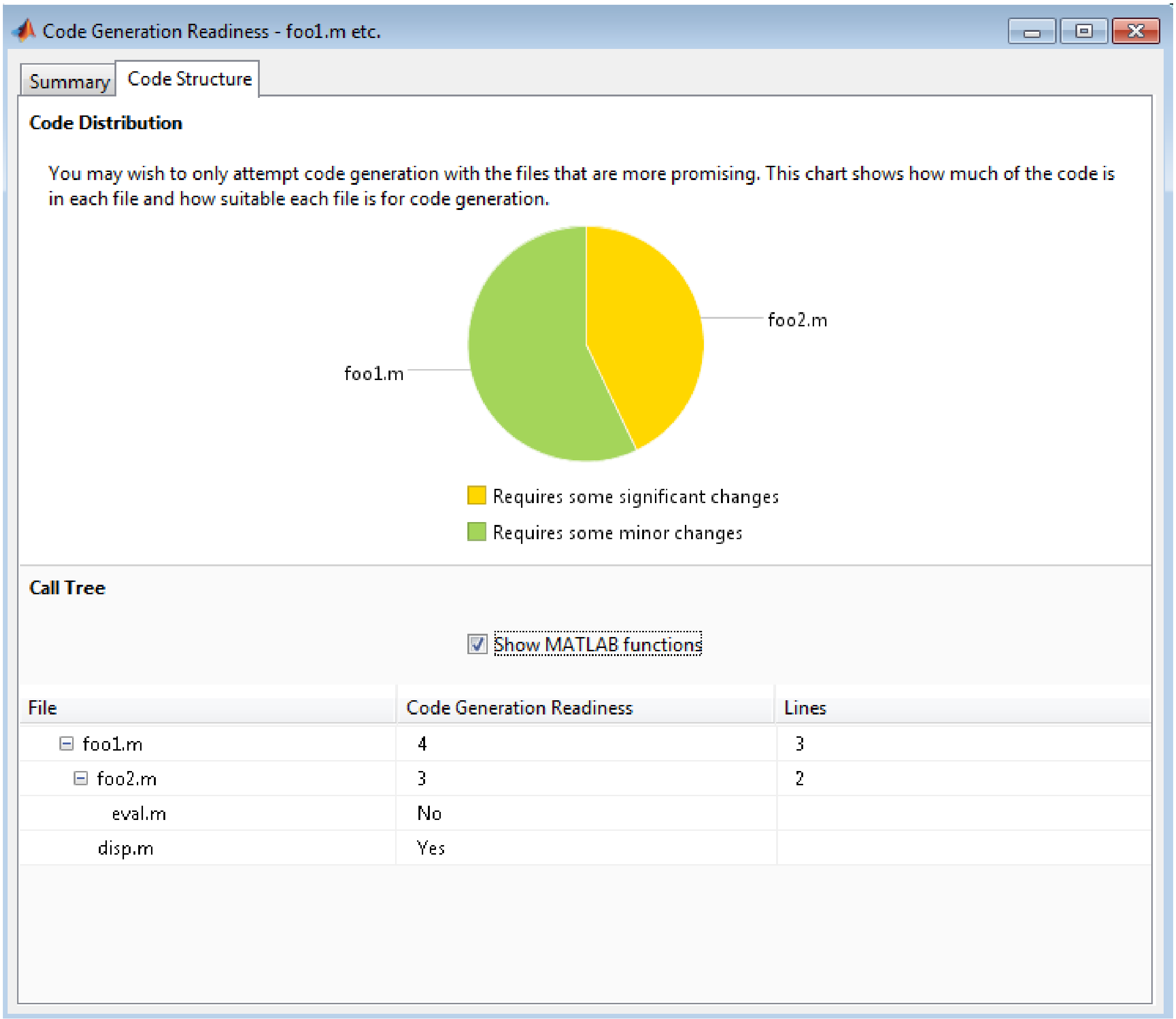
Unsupported MATLAB function calls - 1 invocation

foo2.m → eval 1

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.


Code Generation Readiness - foo1.m etc.
Summary
Code Structure
foo2.m
foo1.m
🟨 Requires some significant changes
🟩 Requires some minor changes

Code Distribution

You may wish to only attempt code generation with the files that are more promising. This chart shows how much of the code is in each file and how suitable each file is for code generation.

Call Tree

☑ Show MATLAB functions

File	Code Generation Readiness	Lines
⊟ foo1.m	4	3
⊟ foo2.m	3	2
eval.m	No	
disp.m	Yes	

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.

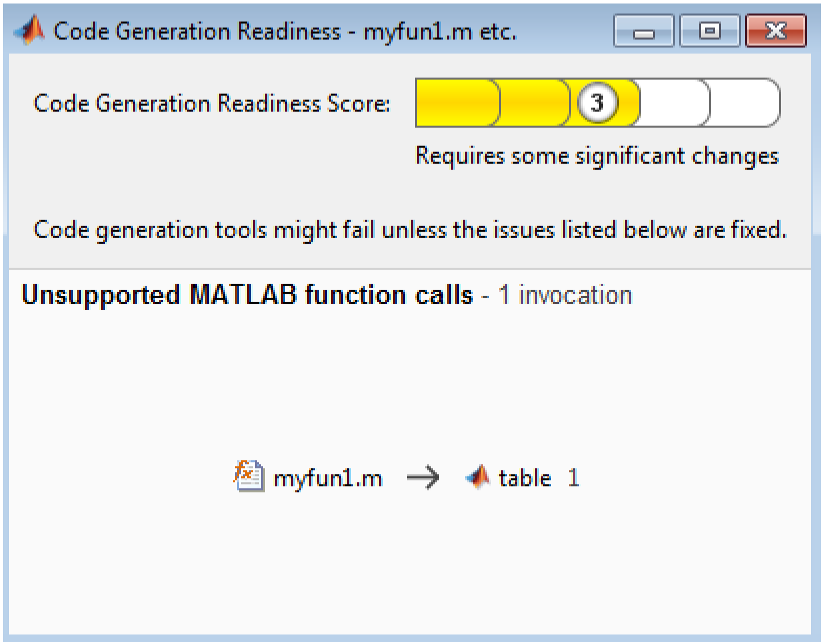
Write the function `myfun` that contains a MATLAB table.

```
function outTable = myfun1(A)
outTable = table(A);
end
```

Analyze `myfun`.

```
coder.screener('myfun1');
```

The code generation readiness report indicates that table data types are not supported for code generation.


Code Generation Readiness - myfun1.m etc.

Code Generation Readiness Score: 3

Requires some significant changes

Code generation tools might fail unless the issues listed below are fixed.

Unsupported MATLAB function calls - 1 invocation

myfun1.m → table 1

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 Parametrize a MATLAB Function

Parametrize a MATLAB function so that it works in MATLAB or in 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 a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `\codegen\lib\myabsval`. (The library file extension can change depending on your platform.) It 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(y)
%#codegen
% 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
  % add the required include statements to generated function code
  coder.updateBuildInfo('addIncludePaths','$(START_DIR)\codegen\lib\myabsval');
  coder.cinclude('myabsval_initialize.h');
  coder.cinclude('myabsval.h');
  coder.cinclude('myabsval_terminate.h');

  % 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
```

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

Rather than providing the library at the command line, you can use `coder.updateBuildInfo` to specify the library within the function. Use this option to preconfigure the build. Add this line to the `else` block:

```
coder.updateBuildInfo('addLinkObjects','myabsval.lib','$(START_DIR)\codegen\lib\myabsva
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex(-2.75)
```

```
ans =

    2.7500
```

Call the MATLAB function `callmyabsval`.

```
callmyabsval(-2.75)
```

```
ans =

    2.7500
```

The `callmyabsval` function exhibits the desired behavior for execution in MATLAB and in code generation.

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.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.cinclude` | `coder.updateBuildInfo`

Topics

"Integrate C Code Using the MATLAB Function Block"

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)
```

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.

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
```

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.

Example: `coder.updateBuildInfo('addSourceFiles', 'fact.c')`

`path` — Path name

character vector | string scalar

Relative 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. The function adds the path to the end of a path name vector.

Leading Argument	Path Specified by path
'addLinkObjects'	Path to linkable objects
'addNonBuildFiles'	Path to nonbuild-related files
'addSourceFiles', 'addSourcePaths'	Path to source files

The relative path starts from the *build folder*. If you have a function `foo` contained in the folder `C:\myCode`, and you generate MEX code by using:

```
codegen foo -report
```

then the build folder is `C:\myCode\codegen\mex\foo`. You can write the path from the build folder or write the path from the current working folder in which you generate code. Reference the current working folder by using the `START_DIR` macro. For example, suppose that your source file is contained in `C:\myCode\mySrcDir`, and you generate code from `C:\myCode`. Write the path as in these examples:

Example: `coder.updateBuildInfo('addSourceFiles','fact.c','..\..\..\mySrcDir')`

Example: `coder.updateBuildInfo('addSourceFiles','fact.c','$(START_DIR)\mySrcDir')`

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

`coder.ExternalDependency` | `coder.ceval` | `coder.cinclude` | `coder.ref` | `coder.rref` | `coder.target` | `coder.wref`

Topics

“Build Process Customization” (MATLAB Coder)
“Integrate C Code Using the MATLAB Function Block”

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('`*`var`*$_1$`', '`*`var`*$_2$`', ...)` declares one or more variables as variable-size data, allowing subsequent assignments to extend their size. Each `'`*`var`*$_n$`'` 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('`*`var`*$_1$`', '`*`var`*$_2$`', ..., `*`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 `'`*`var`*$_n$`'`. If you specify more than one `'`*`var`*$_n$`'`, each variable must have the same number of dimensions.

`coder.varsize('`*`var`*$_1$`', '`*`var`*$_2$`', ..., `*`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 `'`*`var`*$_n$`'`.

coder.varsize('*var*$_1$', '*var*$_2$', ..., [], *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 '*var*$_n$' 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 '*var*$_n$' is used (read). You can add the declaration before the first assignment to each '*var*$_n$'. 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
```

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.varsize` to make the elements of `c` variable-size.

```
function y =  mycell()
c = {1 2 3};
coder.varsize('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.varsize` with that cell array.
- If you use `coder.varsize` with 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` with global variables.
- You cannot use `coder.varsize` with MATLAB class properties.

- You cannot use `coder.varsize` with string scalars.
- For sparse matrices, `coder.varsize` drops upper bounds for variable-size dimensions.

Tips

- `coder.varsize` fixes the size of a singleton dimension unless the `dims` argument explicitly specifies that the singleton dimension has a variable size.

  For example, the following code specifies that `v` has size 1-by-:10. The first dimension (the singleton dimension) has a fixed size. The second dimension has a variable size.

  ```
  coder.varsize('v', [1 10])
  ```

  By contrast, the following code specifies that `v` has size :1-by-:10. Both dimensions have a variable size.

  ```
  coder.varsize('v',[1,10],[1,1])
  ```

  Note For a MATLAB Function block, singleton dimensions of input or output signals cannot have a variable size.

- 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('`*var_1*`', '`*var_2*`', ..., `*`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 1x:2 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

Indicate write-only data to pass by reference

Syntax

`coder.wref(arg)`

Description

`coder.wref(arg)` indicates that `arg` is a write-only expression or variable to pass by reference to an external C/C++ function. Use `coder.wref` only inside a `coder.ceval` call. This function enables the code generator to optimize the generated code by ignoring prior assignments to `arg` in your MATLAB code, because the external function is assumed to not read from the data. Write to all the elements of `arg` in your external code to fully initialize the memory.

Note The C/C++ function must fully initialize the memory referenced by `coder.wref(arg)`. Initialize the memory by assigning values to every element of `arg` in your C/C++ code. If the generated code tries to read from uninitialized memory, it can cause undefined run-time behavior.

See also `coder.ref` and `coder.rref`.

Examples

Pass Array by Reference as Write-Only

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 defines the input variable `array` as a pointer to a double.

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`. This function writes to the input argument but does not read from it.

```
#include "MyStruct.h"

void init_struct(struct MyStruct *my_struct)
{
   my_struct->f1 = 1;
   my_struct->f2 = 2;
}
```

The C header file, MyStruct.h, defines a structure type named MyStruct:

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
    double f1;
    double f2;
} MyStruct;

void init_struct(struct MyStruct *my_struct);

#endif
```

In your MATLAB function, pass a structure as a write-only reference to init_struct. Use coder.cstructname to indicate that the structure type for s has the name MyStruct that is defined in the C header file MyStruct.h.

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','init_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
coder.ceval('init_struct', coder.wref(s));
```

To generate standalone library code, enter:

```
codegen -config:lib foo -report
```

Pass Structure Field as a Write-Only Reference

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.wref(s.s1.a));
...
```

You can also pass an element of an array of structures:

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
```

```
a = struct('b',b);
coder.ceval('foo', coder.wref(a.b(3,4).c(2).u));
...
```

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

- If `arg` is an array, then `coder.wref(arg)` provides the address of the first element of the array. The `coder.wref(arg)` function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

  ```
  coder.ceval('myFun',coder.wref(arg),int32(numel(arg)));
  ```
- 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 parametrize 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`

Topics

"Integrate C Code Using the MATLAB Function Block"

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.

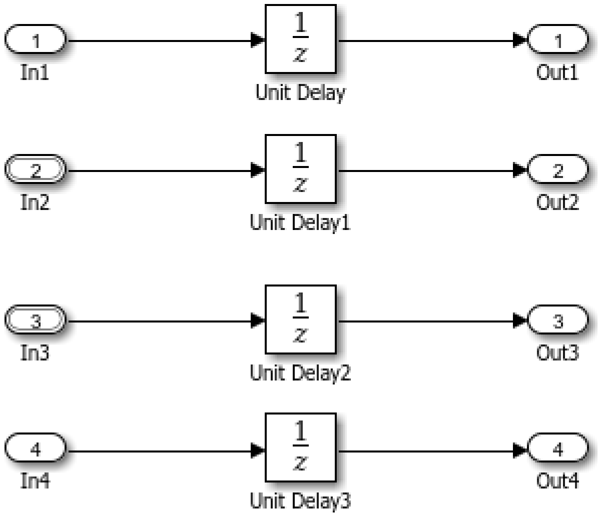
Examples

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 Inport Blocks”

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 2-185. 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 2-185

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 = 'MyLibary_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

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.updateBuildInfo`

Topics

"Develop Interface for External C/C++ Code" (MATLAB Coder)
"Build Process Customization" (MATLAB Coder)
"Integrate External/Custom Code" (MATLAB Coder)

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 2-188 in the current "build context" on page 2-188 . 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 2-188 in the current "build context" on page 2-188, 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 2-188.

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

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.updateBuildInfo`

Topics

“Develop Interface for External C/C++ Code” (MATLAB Coder)
“Build Process Customization” (MATLAB Coder)
“Integrate External/Custom Code” (MATLAB Coder)

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 2-191.

You must implement this method in a subclass of `coder.ExternalDependency`. For an example, see `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 2-191.

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

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.updateBuildInfo`

Topics

"Develop Interface for External C/C++ Code" (MATLAB Coder)
"Build Process Customization" (MATLAB Coder)
"Integrate External/Custom Code" (MATLAB Coder)

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');
Outport_handle = add_block('built-in/Outport','vdp/MyOutport');
```

Delete the blocks you added using a vector of handles.

```
delete_block([Constant_handle Gain_handle Outport_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.Outport(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 = 1×2

   190   150


ans = 1×2

   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

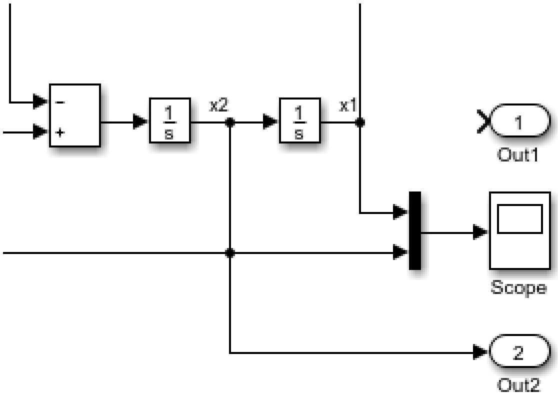
Use `delete_line` with branched lines to remove the segment for any connection.

Open the model `vdp`.

```
open_system('vdp');
```

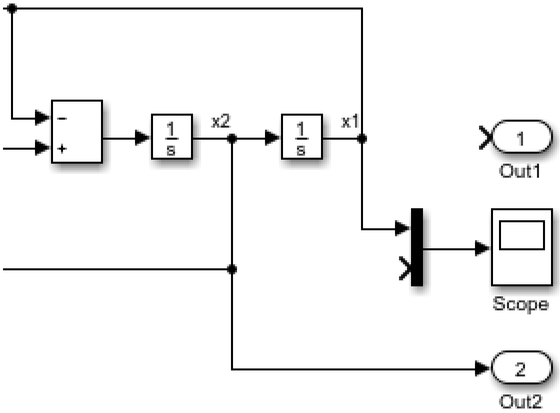
Delete the line from x1 to the Out1 block. This command deletes only the segment of the line that connects the branch to the specified block.

```
delete_line('vdp','x1/1','Out1/1')
```



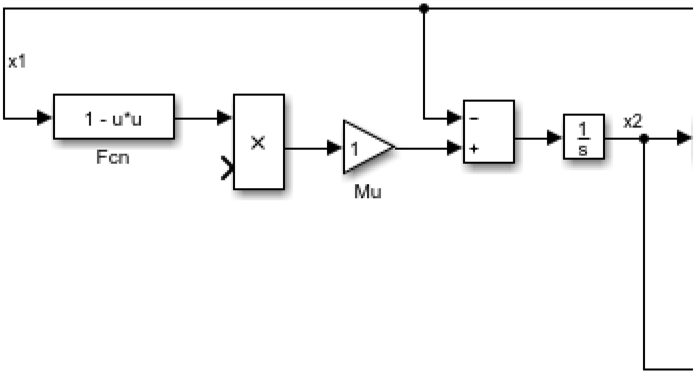
Delete the line segment from x2 to the Mux.

```
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 Simulinkproject, 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 2-224.

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

depview

Display graph of model referencing dependencies with or without library dependencies

`depview` opens the “Model Dependency Viewer”. While `view_mdlrefs` also opens the Model Dependency Viewer, `depview` provides programmatic options that allow you to open a specific configuration of the Model Dependency Viewer.

Syntax

```
depview(sys)
depview(sys,Name,Value)
```

Description

`depview(sys)` opens the Model Dependency Viewer, which displays a graph of dependencies for the model or library specified by `sys`.

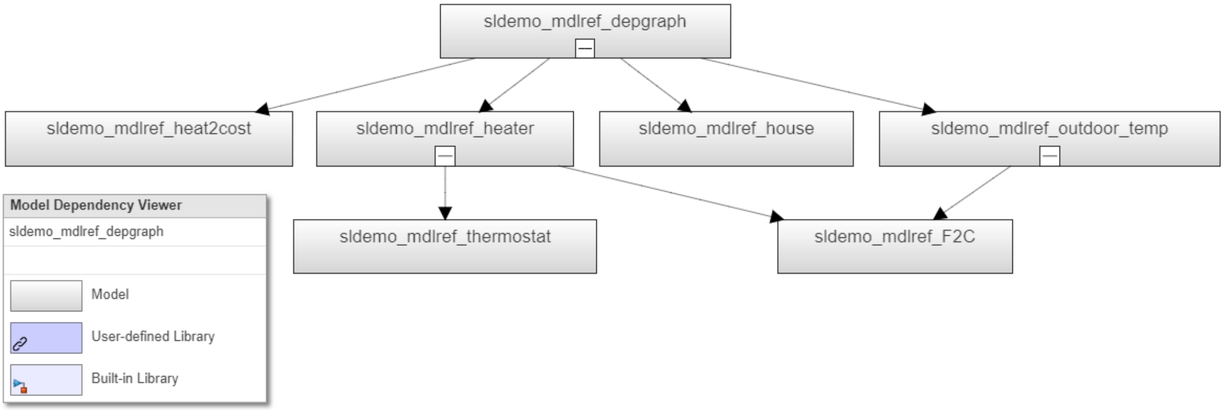
`depview(sys,Name,Value)` opens the Model Dependency Viewer as specified by one or more `Name,Value` pair arguments.

Examples

Open Model Dependency Viewer with Default Settings

Open the default Model Dependency Viewer for the model `sldemo_mdlref_depgraph`.

```
depview('sldemo_mdlref_depgraph');
```

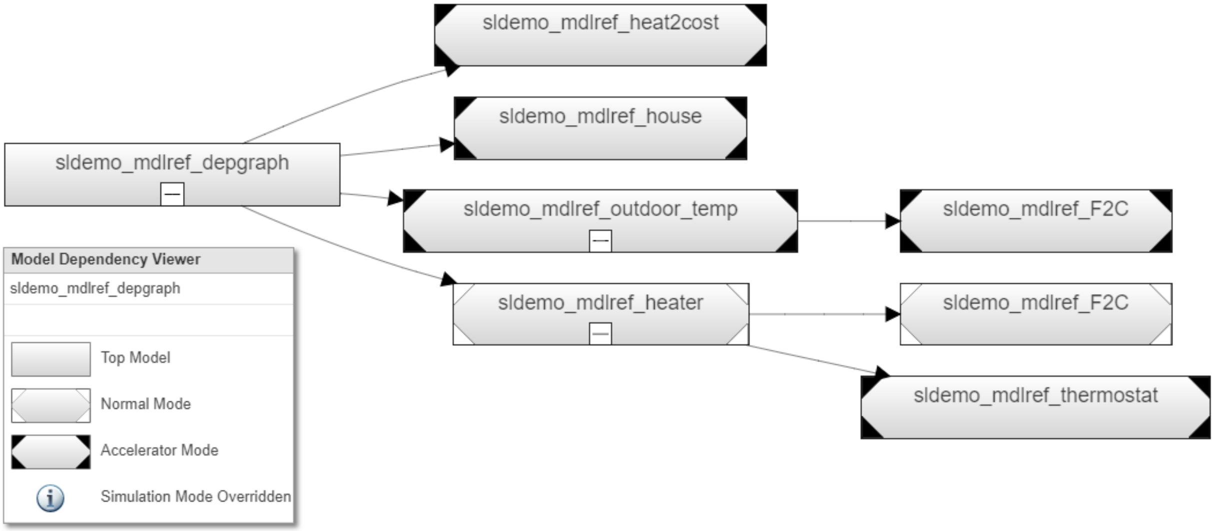


In **Model Files** view, node appearance corresponds to file type.

Open Model Dependency Viewer with Custom Settings

Open the Model Dependency Viewer for the model `sldemo_mdlref_depgraph` in **Model Instances** view with a **Horizontal** layout.

```
depview('sldemo_mdlref_depgraph','ModelReferenceInstance',true,'ShowHorizontal',true);
```



In **Model Instances** view, node appearance corresponds to simulation mode.

Input Arguments

sys — Name or path of model or library

`' '` (default) | character vector

Name or path of model or library, specified as a character vector.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

```
depview('sldemo_mdlref_depgraph','ModelReferenceInstance',true,'ShowFullPath',true,'ShowHorizontal',true)
```

`FileDependenciesExcludingLibraries` — Graph with only models

false (default) | true

To open a Model Dependency Viewer that shows only models, set this parameter to `'true'`. In this view, node appearance corresponds to file type.

Dependencies

You can only set one of `'FileDependenciesExcludingLibraries'`, `'FileDependenciesIncludingLibraries'`, and `'ModelReferenceInstance'` to `'true'`.

`FileDependenciesIncludingLibraries` — Graph with models and user-defined libraries

true (default) | false

To open a Model Dependency Viewer that shows models and user-defined libraries, use the default setting. In this view, node appearance corresponds to file type.

Dependencies

To set this parameter to `'false'`, enable `'FileDependenciesExcludingLibraries'` or `'ModelReferenceInstance'`. You can only set one of `'FileDependenciesExcludingLibraries'`, `'FileDependenciesIncludingLibraries'`, and `'ModelReferenceInstance'` to `'true'`.

`FactoryDependencies` — Graph with models, user-defined libraries, and built-in libraries

false (default) | true

To open a Model Dependency Viewer that shows models, user-defined libraries, and built-in libraries, set this parameter to `'true'`. In this view, node appearance corresponds to file type.

Dependencies

To enable this option, set `'FileDependenciesIncludingLibraries'` to `'true'`.

`ModelReferenceInstance` — Graph with only models that shows each model instance separately

false (default) | true

To open a Model Dependency Viewer that shows each instance of a model as a separate node in the graph, set this parameter to `'true'`. In this view, node appearance corresponds to simulation mode.

Dependencies

You can only set one of `'FileDependenciesExcludingLibraries'`, `'FileDependenciesIncludingLibraries'`, and `'ModelReferenceInstance'` to `'true'`.

`ShowFullPath` — Full path for referenced models and libraries

false (default) | true

To open a Model Dependency Viewer that shows the full path from the top model to each referenced model or library, set this parameter to `'true'`. By default, the Model Dependency Viewer shows only the file name in each node. This parameter is available for only the **Model Instances** view.

Dependencies

To enable this option, set `'ModelReferenceInstance'` to `'true'`.

`ShowHorizontal` — Horizontal dependency display

false (default) | true

To open a Model Dependency Viewer that shows referenced models and libraries to the right of their parents, set this parameter to `'true'`. By default, the Model Dependency Viewer shows referenced models and libraries below their parents.

See Also

Model | `find_mdlrefs` | `view_mdlrefs`

Topics

"Model References"
"Model Dependency Viewer"

Introduced in R2006b

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','')
```

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')
```

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,fileorfolder)
```

Description

`file = findFile(proj,fileorfolder)` 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.

`fileorfolder` — Path of file or folder
character vector | cell array of character vectors | string array

Path of the file or folder to find relative to the project root folder, specified as a character vector, string, or array. Files must include any subfolders under the project root and the file extension. The file or folder must be within the root folder.

Example: ‘models/myModelName.slx’

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 `Utilility`

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_mdlrefs

Find Model blocks and referenced models at all levels or at top level only

Syntax

```
[refMdls,mdlBlks] = find_mdlrefs(system)
[refMdls,mdlBlks] = find_mdlrefs(system,Name,Value)
[refMdls,mdlBlks] = find_mdlrefs(system,allLevels)
```

Description

`[refMdls,mdlBlks] = find_mdlrefs(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_mdlrefs(system,Name,Value)` finds referenced models and Model blocks with additional options specified by one or more `Name,Value` pair arguments.

`[refMdls,mdlBlks] = find_mdlrefs(system,allLevels)` specifies the levels of the system to search.

Tip The `find_mdlrefs` 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 models in `refMdls`.

Examples

Find Referenced Models in Model Hierarchy

Find referenced models and Model blocks for all models referenced by the specified model.

```
load_system('sldemo_mdlref_basic');
[myModels,myModelBlks] = find_mdlrefs('sldemo_mdlref_basic')
```

```
myModels = 2x1 cell array
    {'sldemo_mdlref_counter'}
    {'sldemo_mdlref_basic'  }


myModelBlks = 3x1 cell array
    {'sldemo_mdlref_basic/CounterA'}
    {'sldemo_mdlref_basic/CounterB'}
    {'sldemo_mdlref_basic/CounterC'}
```

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 quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `refModels = find_mdlrefs(top_model,'IncludeProtectedModels',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 model variants in search
`'ActivePlusCodeVariants'` (default) | `'ActiveVariants'` | `'AllVariants'`

Include model variants in search, specified as `'ActivePlusCodeVariants'`, `'ActiveVariants'`, or `'AllVariants'`.

- `'ActivePlusCodeVariants'` — Include all model variants in Variant Subsystem blocks for which you select the **Analyze all choices during update diagram and generate preprocessor conditionals** option.
- `'ActiveVariants'` — Include active model variants for Variant Subsystem blocks.
- `'AllVariants'` — Include all model variants in Variant Subsystem blocks.

`IncludeCommented` — Include commented blocks in search
`false` (default) | `true`

Include commented blocks in search, specified as `false` or `true`.

Data Types: `logical`

`KeepModelsLoaded` — Keep loaded models that function loads

`false` (default) | `true`

The `find_mdlrefs` 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

"Model Reference Basics"
"Inspect Model Hierarchies"
"Reference Protected Models from Third Parties"

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 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. find_system returns a vector of handles when this option is set to 'on', regardless of how you specify System.
'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, find_system 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 2-272.
'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')
```

```
L = 17×1

   32.0020
   31.0020
   30.0020
   29.0020
   28.0020
   27.0020
   26.0020
   25.0020
   24.0020
   23.0020
      ⋮
```

```
A = find_system('vdp','FindAll','on','type','annotation')
```

```
A = 2×1

   34.0020
   33.0020
```

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','regexp','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...'             }
    {'sldemo_clutch/Friction Mode Logic/Lockup...'             }
    {'sldemo_clutch/Friction Mode Logic/Lockup...'             }
    {'sldemo_clutch/Friction Mode Logic/Lockup...'             }
    {'sldemo_clutch/Friction Mode Logic/Lockup...'             }
    {'sldemo_clutch/Friction Mode Logic/Lockup...'             }
    {'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 = 6×1

   10.0001
   13.0001
   14.0001
   14.0001
   15.0001
```

```
    15.0001
```

See Also

`Simulink.allBlockDiagrams` | `Simulink.findBlocks` | `Simulink.findBlocksOfType` | `find_mdlrefs` | `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 sfix(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_interp1 | 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 sfix(8).

xscale

Scaling for input x, such as 2^-3.

ydt

Data type of output y, such as sfix(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...
errmax and *nptsmax*	Meet both criteria, if possible. The *errmax* requirement has higher priority than *nptsmax*. If the breakpoints cannot meet both criteria with the specified spacing, *nptsmax* does not apply.
errmax only	Meet the error criteria, and `fixpt_look1_func_approx` returns the fewest number of points.
nptsmax only	Meet the points criteria, and `fixpt_look1_func_approx` 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_interp1` | `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.

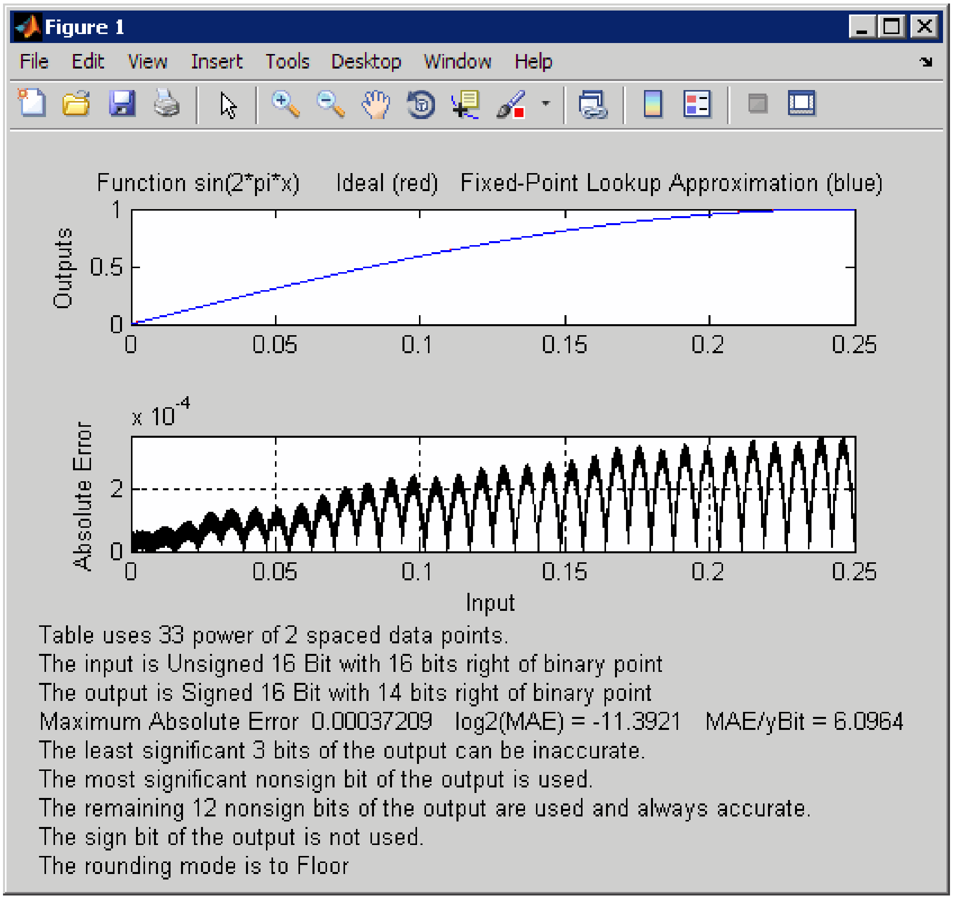


Figure 1
File Edit View Insert Tools Desktop Window Help
Function sin(2*pi*x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0 0.05 0.1 0.15 0.2 0.25
x 10^-4
Absolute Error
2
0
0 0.05 0.1 0.15 0.2 0.25
Input

```
Table uses 33 power of 2 spaced data points.
The input is Unsigned 16 Bit with 16 bits right of binary point
The output is Signed 16 Bit with 14 bits right of binary point
Maximum Absolute Error 0.00037209   log2(MAE) = -11.3921   MAE/yBit = 6.0964
The least significant 3 bits of the output can be inaccurate.
The most significant nonsign bit of the output is used.
The remaining 12 nonsign bits of the output are used and always accurate.
The sign bit of the output is not used.
The rounding mode is to Floor
```

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
```

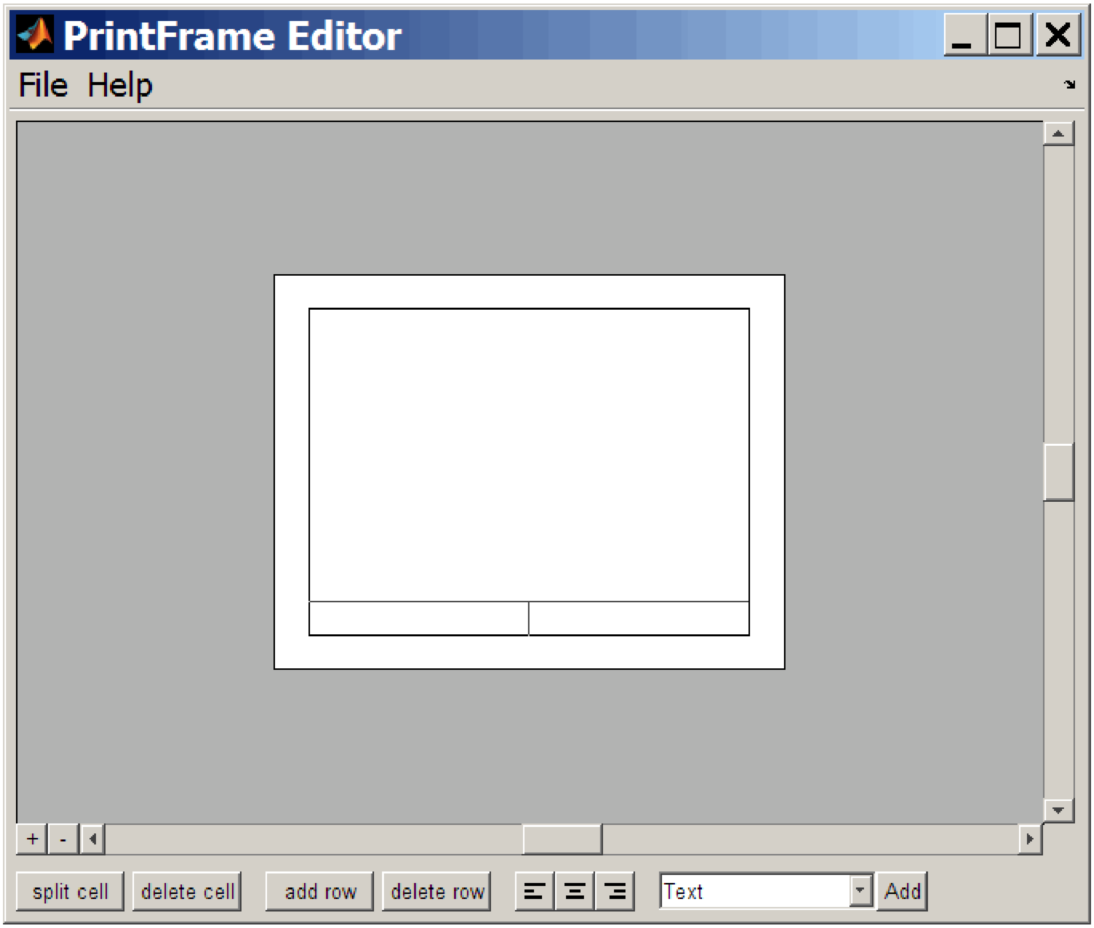
Description

`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

```
bl = gcb
bl = gcb(sys)
```

Description

`bl = 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

`bl = 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.

```
bl = gcb('sldemo_fuelsys/fuel_rate_control')
```

```
bl =

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 path name of current system

Syntax

```
gcs
```

Description

`gcs` returns the path name of the current system. The current system is one of these:

- During editing, the system or subsystem most recently clicked or that contains the block most recently clicked
- During simulation of a system that contains an S-Function block, the system or subsystem containing the S-Function block currently being evaluated
- During callbacks, the system containing any block whose callback routine is being executed
- During evaluation of `MaskInitialization`, the system containing the block whose mask is being evaluated
- The system most recently loaded into memory with `load_system`; only the first use of `load_system` makes the model the current system

The current system is the current model or a subsystem of the current model. Use `bdroot` to get the current model.

If you close the model that contains the current system, another open or loaded system becomes the current one. Use `gcs` to find out the new current system.

To explicitly set the current system, you can either:

- Use `set_param` with the `'CurrentSystem'` parameter on the root Simulink model, for example:

  ```
  set_param(0,'CurrentSystem','mymodel')
  ```

- Open the model by using `open_system` or the model name at the MATLAB command prompt.

Examples

Get Current System That Contains a Block

Return the path of the system that contains the most recently selected block.

Open the model `sldemo_fuelsys`. Open the subsystem To Controller.

```
sldemo_fuelsys
open_system('sldemo_fuelsys/To Controller')
```

Click the Rate Transition block. Get the current system.

```
gcs

ans =

    'sldemo_fuelsys/To Controller'
```

Get Current System After Loading a Model

Open the model `f14` and get the current system.

```
f14
gcs

ans =

    'f14'
```

Load the model `vdp` using `load_system`. Then get the current system.

```
load_system('vdp');
gcs

ans =

    'vdp'
```

To remove `vdp` from memory, close it. In this example, the current system becomes the open model, `f14`.

```
close_system('vdp');
gcs
```

```
ans =

    'f14'
```

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'
```

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, the function returns an empty object.

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 = getSlRootInportMap('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 `getSlRootInportMap` 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 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]
```

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] = getSlRootInportMap('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 getSlRootInport 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 = getSlRootInportMap('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 = getSlRootInportMap('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

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.

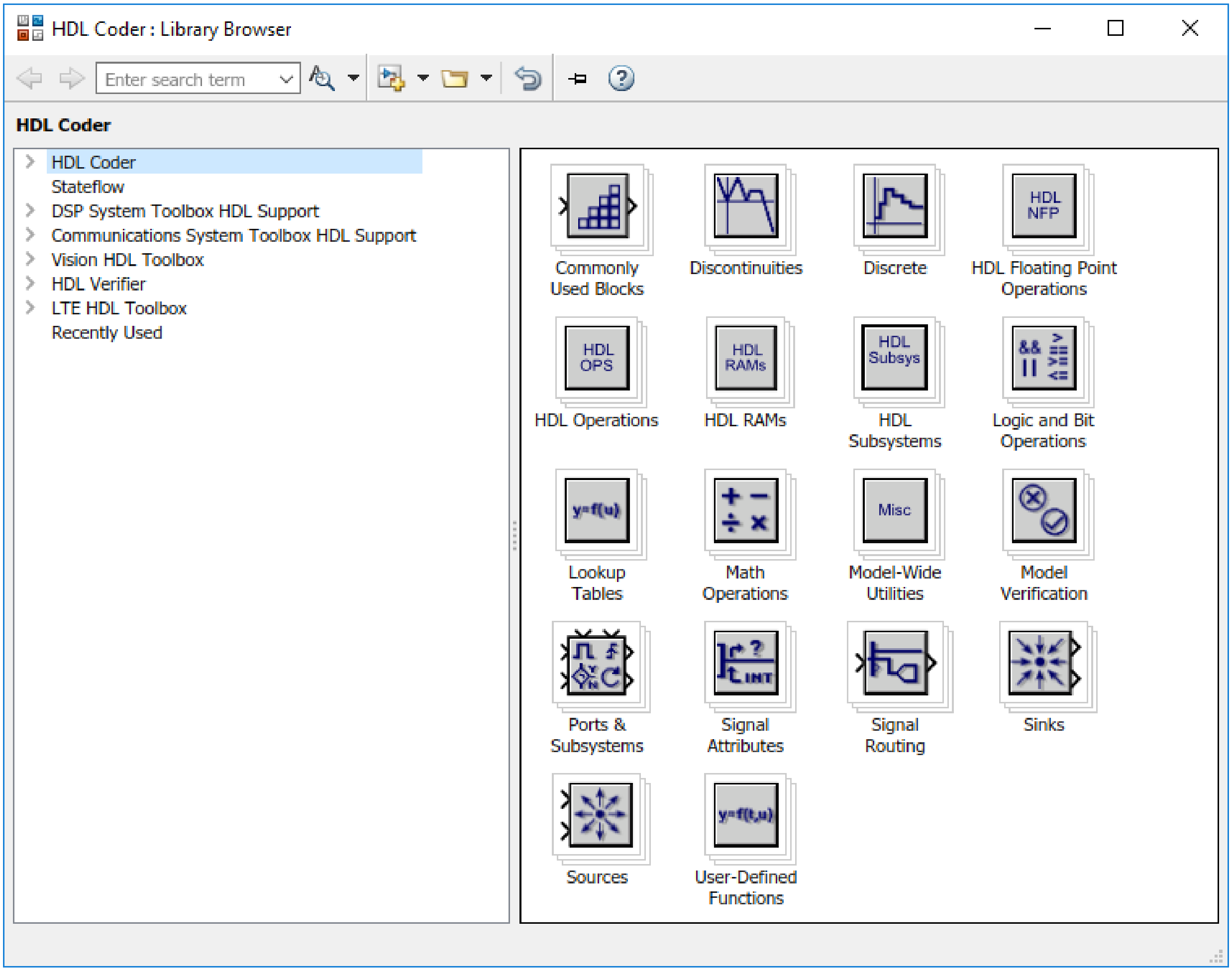
Examples

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".
```

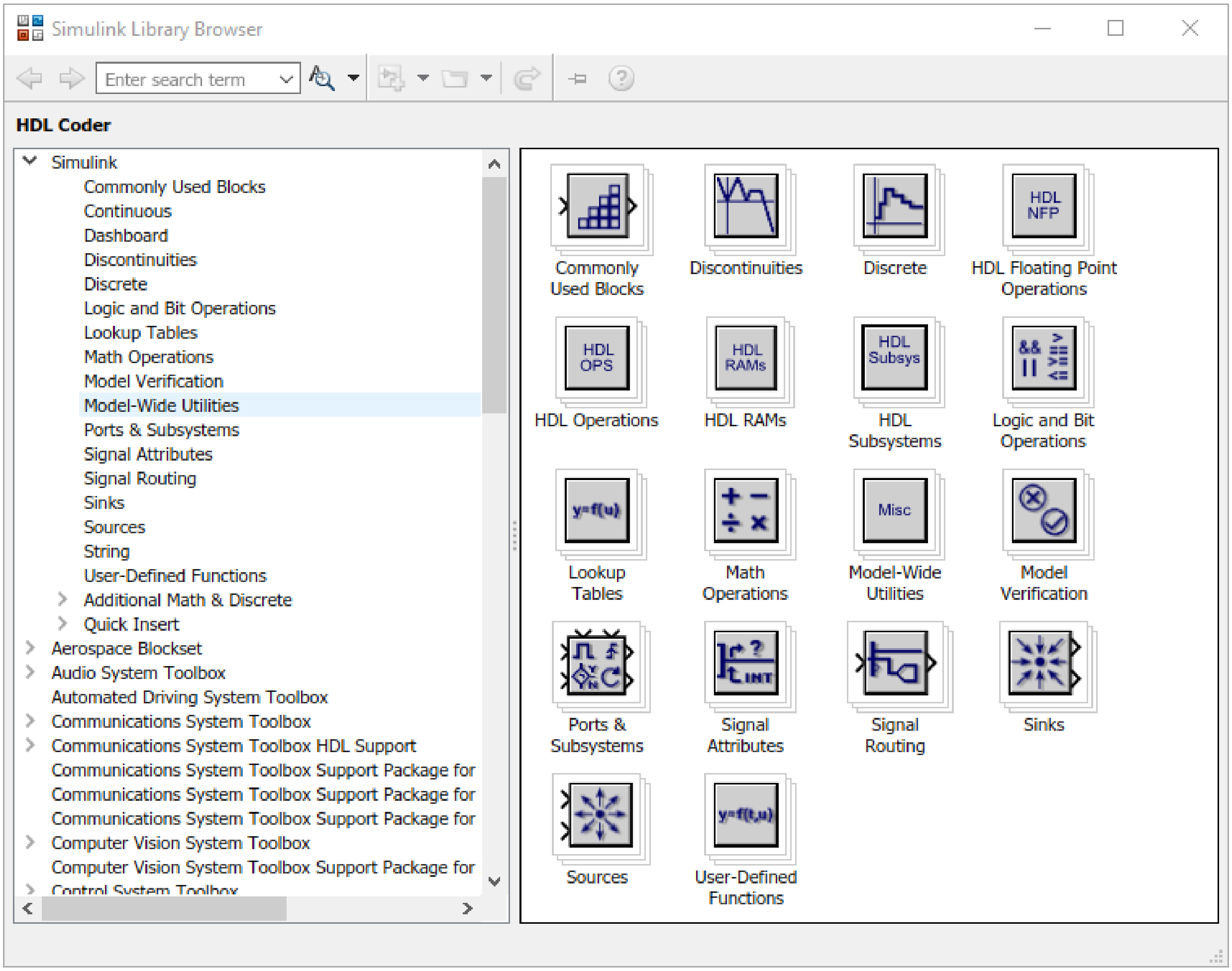


Display All Blocks in the Library Browser

To display all blocks in the Library Browser, regardless of HDL code generation compatibility:

```
hdllib('off')
```

```
### Restoring Library Browser to default view; removing the HDL Coder compatibility fi
```



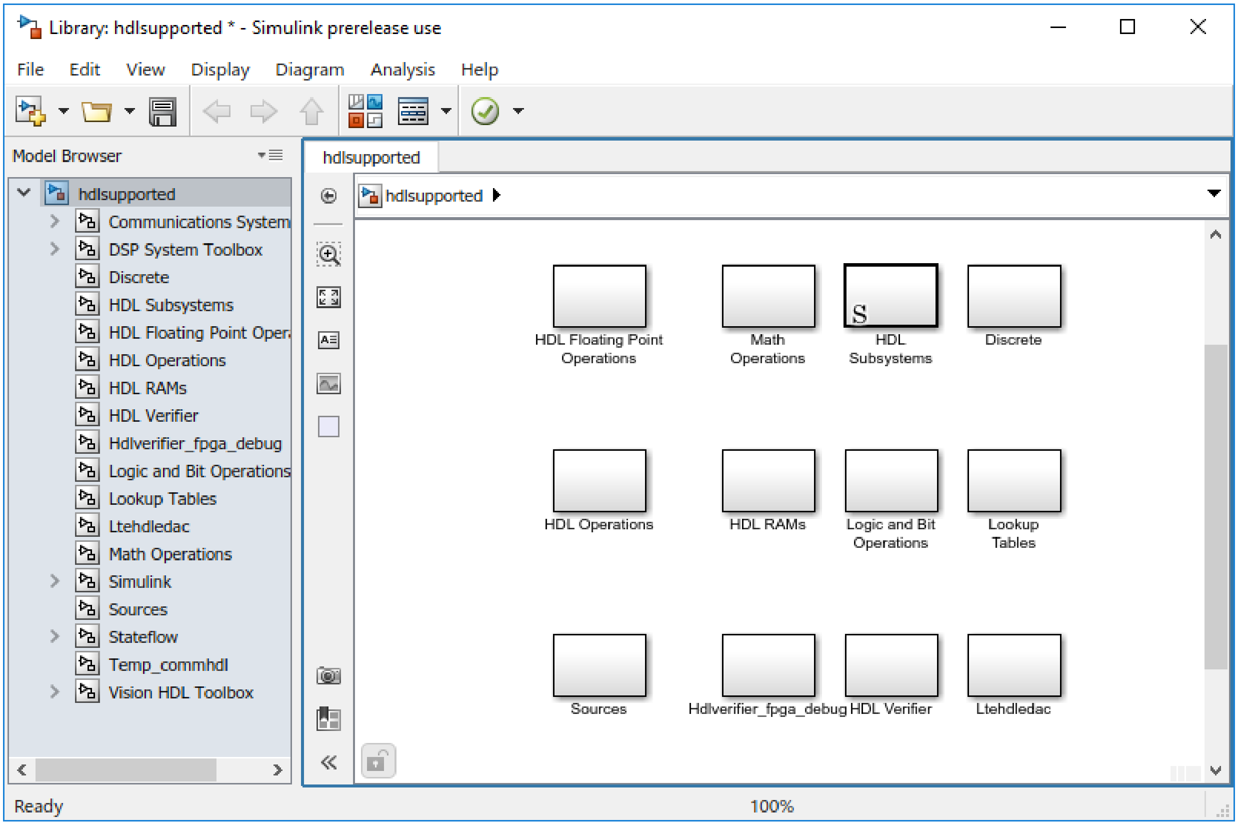
Create a Supported Blocks Library and HTML Reports

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 hdlsupported.html
```

The `hdlsupported` library opens. To view the reports, click the `hdlblklist.html` and `hdlsupported.html` links.

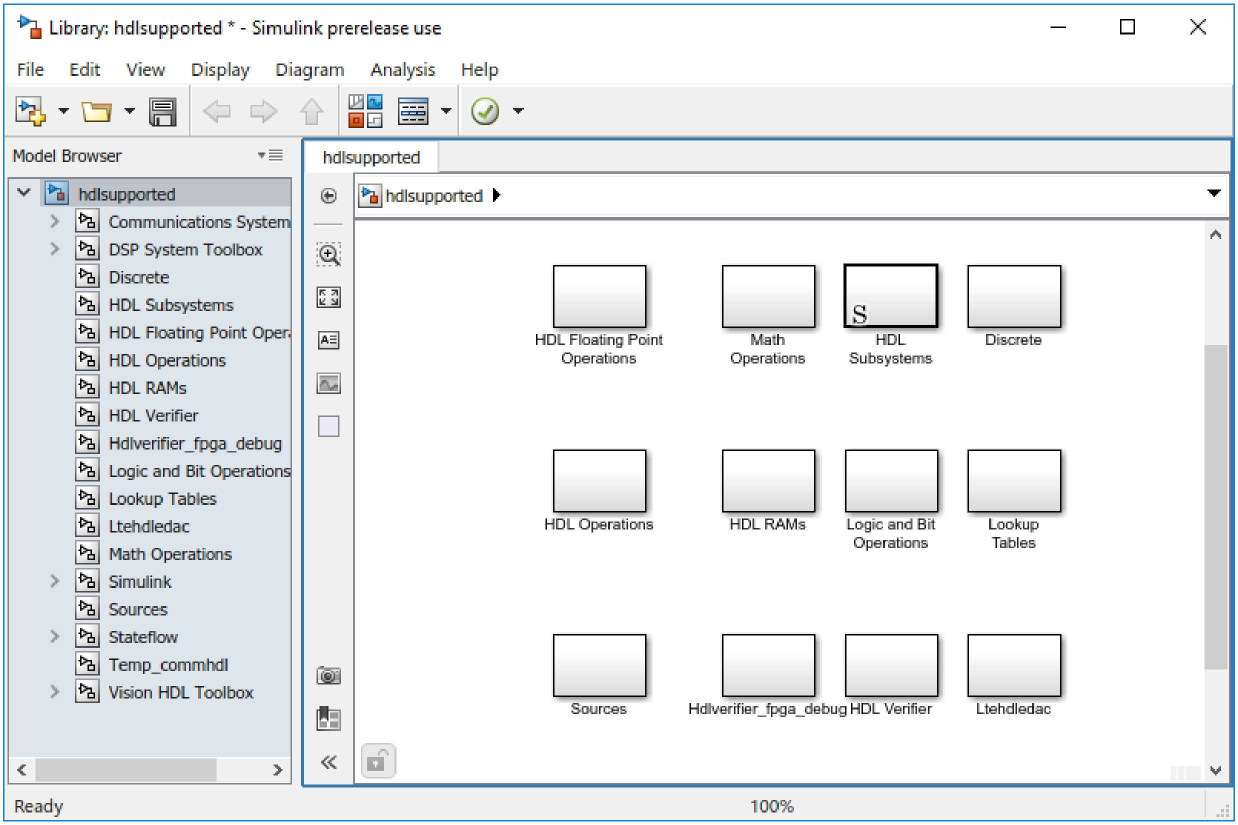


Create a Supported Blocks Library

To create a library that contains blocks that are compatible with HDL code generation:

```
hdllib('librarymodel')
```

The `hdlsupported` block library opens.



See Also

“Supported Blocks” (HDL Coder)

Topics

“Show Blocks Supported for HDL Code Generation” (HDL Coder)
“View HDL-Specific Block Documentation” (HDL Coder)
“Create HDL-Compatible Simulink Model” (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 | Simulink identifier | traceability tag

Block, port, line, or annotation to highlight, specified as:

- The full block path name
- A numeric handle for lines, ports, or annotations
- Simulink identifier
- 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'`, `'vdp:3'`, `'<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 2-361.

- `'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
```

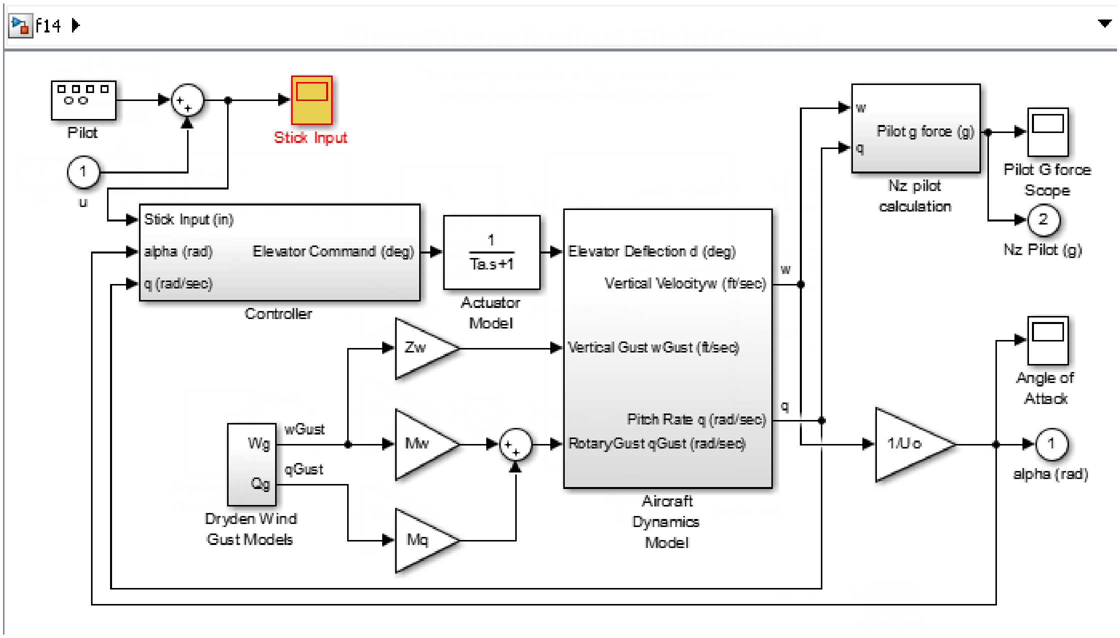
In the model configuration parameters, in the **Solver** pane, set **Type** to `Fixed-step`.

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.

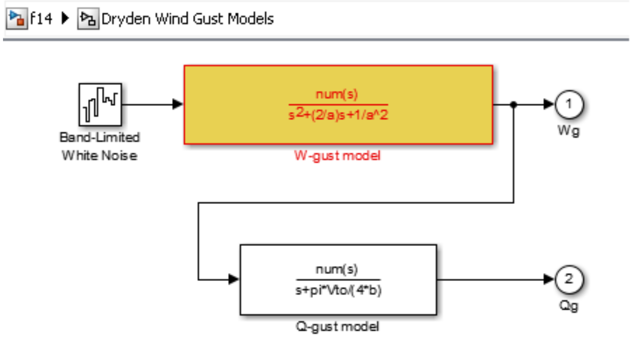
Highlight a block using a traceability tag.

```
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 (MATLAB) 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).
- convertNDArrayToRowMajor— 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 convertNDArrayToRowMajor 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`	Path of the link to the library block
`Library`	Name of the library containing the referenced block
`ReferenceBlock`	Path of the library block
`LinkStatus`	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

sys

Name of the Simulink system from which the linear model is extracted.

`x` and `u`

State (`x`) and the input (`u`) 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 `x`. To extract the `x` 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'`.

`Ts`

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 `linmodv5`.

`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 (`t`) 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_struc = 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

sys

Name of the Simulink system from which the linear model is extracted.

x, *u*

State (*x*) and the input (*u*) 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 `x`. To extract the `x` 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 (`t`) 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_struc = 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 ith 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

sys

Name of the Simulink system from which the linear model is extracted.

x, *u*

State (*x*) and the input (*u*) 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 `x`. To extract the `x` 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 (t) 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_struc = 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

Load Simulink model into memory

Syntax

```
handle = load_system(sys)
```

Description

`handle = load_system(sys)` loads the model `sys` into memory without opening the model in the Simulink Editor. After you load a model into memory, you can work with it using Simulink API commands. Save changes to the model using `save_system`.

Examples

Load Model into Memory

Load the model `vdp` into memory and return the model handle.

```
h = load_system("vdp")
```

```
h =

  172.0004
```

Input Arguments

sys — Model to load into memory
character vector | string scalar

Model to load into memory, specified as a character vector or string scalar.

Example: `"vdp"`

Output Arguments

`handle` — Handle of loaded model

handle

Handle of loaded model.

See Also

`close_system` | `open_system` | `save_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: `logsout`

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

`sys`	Vector of model size data: • `sys(1)` = number of continuous states • `sys(2)` = number of discrete states • `sys(3)` = number of outputs • `sys(4)` = number of inputs • `sys(5)` = reserved • `sys(6)` = direct-feedthrough flag (1 = yes, 0 = no) • `sys(7)` = number of sample times (= number of rows in `ts`)
`x0`	Vector containing the initial conditions of the system's states
`str`	Vector of names of the blocks associated with the model's states. The state names and initial conditions appear in the same order in `str` and `x0`, respectively.
`ts`	An *m*-by-2 matrix containing the sample time (period, offset) information
outputs	Outputs of the model at time step `t`.
derivs	Derivatives of the continuous states of the model at time `t`.

dstates	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 (`[]`) or in structure format. Otherwise, Simulink returns an array. • 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`	Specification of the simulation phase to be executed: • `'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. **Note** The `output`, `update`, and `derivs` flags are valid only for single-tasking models. For more information on single-tasking and multi-tasking, see “Tasking Modes” (Simulink Coder).

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',subsys)
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',subsys)` creates a model based on the subsystem `subsys` 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

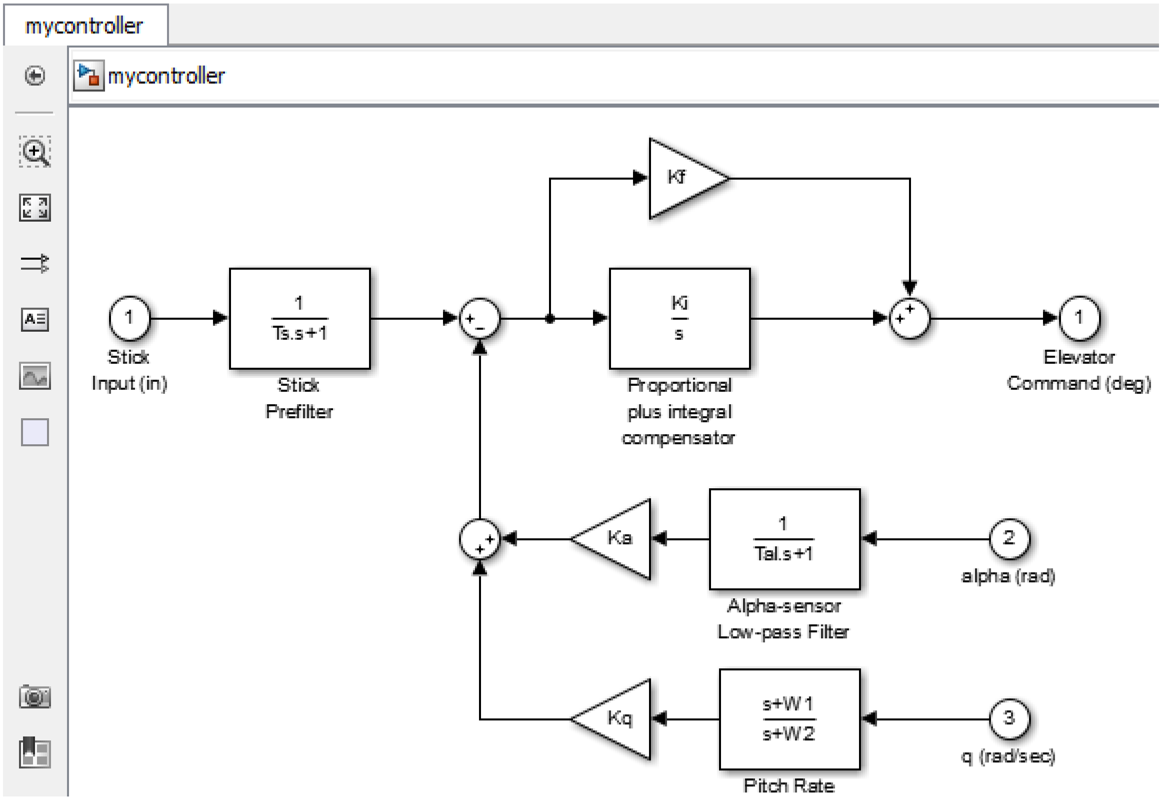
Create a model named `mynewmodel` based on `myoldmodel`, which is in the current folder.

```
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 'ErrorIfShadow
```

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:

OrigValue	Value to be converted to a fixed-point representation. Must be specified using a double data type.
FixPtDataType	The fixed-point data type used to convert OrigValue.
FixPtScaling	Scaling of the output in either Slope or [Slope Bias] format. If FixPtDataType does not specify a generalized fixed-point data type using the sfix or ufix command, FixPtScaling is ignored.
RndMeth	Rounding technique used if the fixed-point data type lacks the precision to represent OrigValue. If FixPtDataType specifies a floating-point data type using the float command, RndMeth is ignored. Valid values are Zero, Nearest, Ceiling, or Floor (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 =

   9.50000000000000
```

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 =

   9.75000000000000
```

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 =

   9.87500000000000
```

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_mdlref_counter`, which is referenced by the CounterA model block in `sldemo_mdlref_basic`.

```
open_system('sldemo_mdlref_basic')
open_system('sldemo_mdlref_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 multiple times in parallel or serial

Syntax

```
simOut = parsim(in)
simOut = parsim(in,'ShowSimulationManager','on')
simOut = parsim(in,Name,Value,...NameN,ValueN)
```

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.

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 or if Parallel Computing Toolbox is not used.

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')
```

```
[08-Jan-2018 14:10:43] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.
[08-Jan-2018 14:11:12] Loading Simulink on parallel workers...
[08-Jan-2018 14:11:40] Configuring simulation cache folder on parallel workers...
[08-Jan-2018 14:11:40] Loading model on parallel workers...
[08-Jan-2018 14:11:48] Running simulations...
[08-Jan-2018 14:12:04] Completed 1 of 6 simulation runs
[08-Jan-2018 14:12:04] Completed 2 of 6 simulation runs
[08-Jan-2018 14:12:04] Completed 3 of 6 simulation runs
[08-Jan-2018 14:12:08] Completed 4 of 6 simulation runs
[08-Jan-2018 14:12:09] Completed 5 of 6 simulation runs
[08-Jan-2018 14:12:09] Completed 6 of 6 simulation runs
[08-Jan-2018 14:12:09] Cleaning up parallel workers...

out = 

1x6 Simulink.SimulationOutput array
```

Using `parsim` with Rapid Accelerator

Simulate the model, `vdp`, in rapid accelerator mode.

Load the model.

```
model = 'vdp';
load_system(model)
```

This step builds the Rapid Accelerator target

```
Simulink.BlockDiagram.buildRapidAcceleratorTarget(model);
```

Create a `SimulationInput` object and use `setModelParameter` method to set `RapidAcceleratorUpToDateCheck` to `'off'`.

```
in = in.setModelParameter('SimulationMode', 'rapid-accelerator');
in = in.setModelParameter('RapidAcceleratorUpToDateCheck', 'off');
```

Simulate the model.

```
out = parsim(in)
```

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. To pass to the `parsim` command, use the list of following input arguments

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'`

AttachedFiles — Files to attach to parallel pool

cell array

Specified as a cell array of additional files to attach to the parallel pool.

`ShowProgress` — Show the progress of the simulations

'on'(default) | 'off'

Set to `'on'`, to 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`. For more information, see `Simulink.Simulation.Future`.

`RunInBackground` — Run simulations in background

'off' (default) | 'on'

Set to `'on'` to run simulations asynchronously, keeping the MATLAB command prompt available for use.

`SetupFcn` — Function handle to run once per worker

function handle

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

Specify a `function handle` to `'CleanupFcn'` to run once per worker after the simulations are completed.

`ManageDependencies` — Manage model dependecies

'on' (default) | 'off'

When `ManageDependencies` is set to `on`, model dependencies are automatically sent to the parallel workers if required. If `ManageDependencies` is set to `off`, explicitly attache model dependencies to the parallel pool.

`UseFastRestart` — Use fast restart

'off' (default) | 'on'

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 “Get Started with Fast Restart” for more information.

`TransferBaseWorkspaceVariables` — Transfer variables to the parallel workers

'off' (default) | 'on'

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.

`StopOnError` — Starts the Simulation Manager app

'off' (default) | 'on'

Setting `'StopOnError'` to `'on'` stops the execution of simulations if an error is encountered.

Output Arguments

`simOut` — Simulation object containing logged simulation results

object

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.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

If you have Parallel Computing Toolbox installed, then when you use `parsim`, MATLAB automatically opens a parallel pool of workers on your local machine. MATLAB runs the simulations across the available workers. Control parallel behavior with the parallel preferences, including scaling up to a cluster.

For details, see “Run Multiple Simulations”.

See Also

`ExternalInput` | `Simulation Manager` | `Simulink.Simulation.Future` | `Simulink.SimulationInput` | `applyToModel` | `cancel` | `fetchNext` | `fetchOutputs` | `setBlockParameter` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate` | `wait`

Topics

“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”
“Run Parallel Simulations”

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.

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
`isLoaded` | `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 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/To Workspace'`. 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

`replBlks` — 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 model

Syntax

```
filename = save_system
filename = save_system(sys)
filename = save_system(sys,newsys)
filename = save_system(sys,newsys,Name,Value)
```

Description

`filename = save_system` saves the current top-level model. If the model was not previously saved, `save_system` creates a file in the current folder.

To save a subsystem, instead use `Simulink.SubSystem.copyContentsToBlockDiagram` to copy the subsystem contents to a new model. You can then save that model using `save_system`. See `Simulink.SubSystem.copyContentsToBlockDiagram`.

If you set the model `UpdateHistory` property to `UpdateHistoryWhenSave`, no dialog box prompt appears when you use `save_system` to save the model. If you want to update the comment, use the `'ModifiedComment'` parameter with `set_param` before saving, for example:

```
set_param('mymodel','ModifiedComment','Here is my comment.')
```

`filename = save_system(sys)` saves the model `sys`. The model must be open or loaded.

`filename = save_system(sys,newsys)` saves the model to a new file `newsys`. If you do not specify an extension, then `save_system` uses the file format specified in your Simulink preferences.

`filename = save_system(sys,newsys,Name,Value)` saves the system with additional options specified by one or more `Name,Value` pair arguments. To use `Name,Value` pairs without saving to a new file, use `[]` for `newsys`.

Input Arguments

sys — Name of model to save

character vector | cell array of character vectors | string array | handle | array of handles

Name of model to save, specified as a character, cell array of character vectors, string array, handle, or array of handles. Do not use a file extension.

newsys — File to save to

character vector | cell array of character vectors | string array | [ ]

File to save to, specified as a character vector, cell array of character vectors, string array, or, to use Name,Value pairs without changing the file name, []. You can specify a model name in the current folder or the full path name, with or without an extension.

With no an extension, save_system saves to the file format specified in your Simulink preferences. Possible model extensions are .slx and .mdl. With the 'ExportToXML' option, use the extension .xml.

For information on rules for naming models, see “Model Names”.

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 quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example:

```
save_system('mymodel','newmodel','SaveModelWorkspace',true,'BreakUse
rLinks',true,'OverwriteIfChangedOnDisk',true)
```

AllowPrompt — Allow dialog box prompts

false (default) | true | 'on' | 'off'

Option to allow dialog box prompts, specified as true, false, 'on', or 'off'. By default, warnings and error messages appear at the command line.

BreakAllLinks — Replace links to library blocks

false (default) | true | 'on' | 'off'

Option to, in the saved file, replace links to library blocks with copies of the library blocks, specified as `true`, `false`, `'on'`, or `'off'`. This option affects user-defined blocks and Simulink library blocks.

Caution The `'BreakAllLinks'` option can result in compatibility issues when upgrading to newer versions of Simulink. For example:

- Any masks on top of library links to Simulink S-functions do not upgrade to the new version of the S-function.
- Any library links to masked subsystems in a Simulink library do not upgrade to the new subsystem behavior.
- Any broken links prevent the library forwarding mechanism from upgrading the link.

If you saved a model with broken links to built-in libraries, use the Upgrade Advisor to scan the model for out-of-date blocks. Then upgrade the Simulink blocks to their current versions.

`BreakUserLinks` — Replace links to user-defined blocks

`false` (default) | `true` | `'on'` | `'off'`

Option to, in the saved file, replace links to user-defined library blocks with copies of the library blocks, specified as `true`, `false`, `'on'`, or `'off'`.

`BreakToolboxLinks` — Replace links to built-in library block

`false` (default) | `true` | `'on'` | `'off'`

Option to, in the saved file, replace links to built-in library blocks with copies of the library blocks, specified as `true`, `false`, `'on'`, or `'off'`. This option affects Simulink library blocks and blocks from libraries supplied with MathWorks toolboxes or blocksets.

`ErrorIfShadowed` — Return an error if name exists

`false` (default) | `true` | `'on'` | `'off'`

Option to return an error if the new name exists on the MATLAB path or workspace, specified as `true`, `false`, `'on'`, or `'off'`.

`ExportToXML` — Export model to XML format

`false` (default) | `true` | `'on'` | `'off'`

Option to export the model to a file in a simple XML format, specified as `true`, `false`, `'on'`, or `'off'`. Specify the full name of the file, including the `.xml` extension. The block diagram in memory does not change and no callbacks execute. Use this option without any other Name,Value pair arguments. This option warns and will be removed in a future release.

Example: save_system('mymodel','exportfile.xml','ExportToXML',true)

ExportToVersion — MATLAB release name to export to

character vector | string scalar

MATLAB release name to export to, specified in either of these forms (not case sensitive). You can export to seven years of previous releases.

- Release name, for example, 'R2012A', 'R2016B'
- Release name, followed by an underscore and then the extension, for example, 'R2016A_SLX', 'R2014A_MDL'. For releases before R2012a, you can specify only to .mdl files. If you do not specify an extension, you export to the file format specified in your Simulink preferences.

save_system exports the system such that the specified Simulink version can load it. If the system contains functionality not supported by the specified Simulink version, the command removes the functionality in the exported file. It also replaces unsupported blocks with empty masked subsystem blocks colored yellow. As a result, the exported system might generate different results.

Alternatively, use Simulink.exportToVersion or, interactively, the Export to Previous Version dialog box.

OverwriteIfChangedOnDisk — Overwrite file

false (default) | true | 'on' | 'off'

Option to overwrite the file on disk even if it has been modified since the system was loaded, specified as true, false, 'on', or 'off'. By default, if the file changed on disk since the model was loaded, save_system displays an error to prevent the changes on disk from being overwritten.

You can control whether save_system displays an error if the file has changed on disk using a Simulink preference. In the **Model File** pane of the Simulink Preferences dialog box, under **Change Notification**, select **Saving the model**. This preference is on by default.

SaveDirtyReferencedModels — Save referenced models with unsaved changes

false (default) | true | 'on' | 'off'

Option to save referenced models that contain unsaved changes while also saving the model, specified as `true`, `false`, `'on'`, or `'off'`. By default, attempting to save a model that contains unsaved referenced models return an error.

`SaveModelWorkspace` — Save model workspace
`false` (default) | `true` | `'on'` | `'off'`

Option to save the contents of the model workspace, specified as `true`, `false`, `'on'`, or `'off'`. 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”.

Output Arguments

`filename` — Name of saved file
character vector | cell array of character vectors

Full name of saved file, returned as a character vector or a cell array of character vectors.

Examples

Save Named Model

Create a model.

```
new_system('newmodel')
```

Save the model.

```
save_system('newmodel')
```

Save Model with Another Name

Open the model `vdp`. Save it to a model named `myvdp` in the current folder. Without a file extension, the function saves the model using the format specified in your Simulink preferences.

```
open_system('vdp')
save_system('vdp','myvdp')
```

After you save the model by another name, the model is no longer open under its original name. Open vdp again and save it as an .mdl file in the current folder.

```
open_system('vdp')
save_system('vdp','mynewvdp.mdl')
```

Return Error If Name Exists

Save a model with a new name and return an error if something with this name exists on the MATLAB path. In this case, save_system displays an error because max is the name of a MATLAB function. The model is not saved.

```
open_system('vdp')
save_system('vdp','max','ErrorIfShadowed',true)
```

```
Error using save_system (line 38)
The model 'vdp' cannot be saved with the new name 'max', because this name is
shadowing another name on the MATLAB path or in the workspace.  Choose another
name, or do not use the option 'ErrorIfShadowed'
```

Save Model with Options

Suppose that you have a model named mymodel. Open the model and save it to a model named newmodel. Also save the model workspace, break links to user-defined library blocks, and overwrite if the file has changed on disk,

```
open_system('mymodel')
save_system('mymodel','mynewmodel','SaveModelWorkspace',
true,'BreakUserLinks',true,'OverwriteIfChangedOnDisk',true)
```

Save Model to Same Name and Use Options

Save the model mymodel, breaking links to user-defined library blocks in the model.

```
save_system('mymodel',[],'BreakUserLinks',true)
```

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])
```

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. Parameter names and values 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”
“Run 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

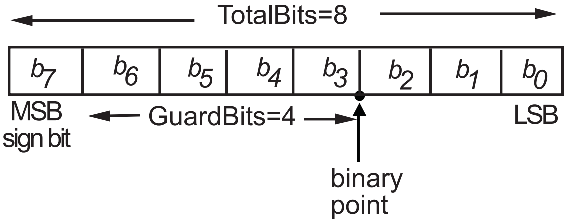
```
a = sfrac(WordLength)
a = sfrac(WordLength, GuardBits)
```

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 $(1 - 2^{(1 - 8)}).2^4 = 15.875$.

```
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 2-453
- "Adding New Groups" on page 2-455
- "Working with Signals" on page 2-456
- "Using Get/Set Methods for Specific Signals and Groups" on page 2-457
- "Querying, Labelling, and Setting the Active Group" on page 2-458
- "Enabling Current Group Display" on page 2-458
- "Printing Signal Groups" on page 2-458
- "Interpolating Missing Data Values" on page 2-459

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.
>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.

***onoff* 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
groupIndex	Scalar specifying index of signal group to print	2
timeRange	Two-element vector specifying the time range to print (must not exceed the block's time range)	[3 6]
visibleSignals	Vector specifying index of signals to print	[1 2]
yLimits	Cell array specifying limits for each signal's *y*-axis	{[-1 1], [0 1]}
extent	Two-element vector of the form: [width, height] specifying the dimensions (in pixels) of the area in which to print the signals	[500 300]
showTitle	Logical value specifying whether to print a title; true (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).

fighs = 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 2-475.

`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:

```
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget('vdp');
```

```
### Building the rapid accelerator target for model: vdp
### Successfully built the rapid accelerator target for model: vdp
```

```
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','logsout')
```

```
simOut = 
  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

simOut =
  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);
```

```
### Building the rapid accelerator target for model: vdp
### Successfully built the rapid accelerator target for model: vdp
```

```
set_param(mdl, 'SimulationMode', simMode)
```

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.setBlockParameter('CSTR/Feed Temperature', 'Value', '300'); simIn.setModelParameter('StartTime', '1'); simIn.setVariable('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 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
T	The time vector returned.
X	The state returned in matrix or structure format. The state matrix contains continuous states followed by discrete states.

Argument	Description
Y	The output returned in matrix or structure format. For block diagram models, this variable contains all root-level blocks.
Y1,...,*Yn*	The outports, which can only be specified for diagram models. Here, *n* must be the number of root-level blocks. Each outport will be returned in the *Y*1,...,*Yn* variables.
'*model*'	The name of the model to simulate.
Timespan	The timespan can be `TFinal`, `[TStart TFinal]`, or `[TStart OutputTimes TFinal]`. Output times are time points returned in *T*, but in general, *T* includes additional time points.
Options	Optional simulation parameters created in a structure by the `simset` command using name-value pairs.
UT	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**. **Ctrl+C** breaks the simulation and the simulation results are not saved in the MATLAB workspace.
- 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 simulations that error out.
- Abort simulations.
- Select a simulation run and open the model in Simulink, with all of the simulation's settings applied to the model.
- View simulation results in the **Simulation Data Inspector**.

Open the Simulation Manager App

- `sim` command – You can use `sim` with `SimulationInput` object to open the UI. Set the argument `ShowSimulationManager` to `on`.

  ```
  out = sim(in, 'ShowSimulationManager', 'on')
  ```
- `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')
  ```
- To reopen the Simulation Manager, use the command `openSimulationManager('modelName')`. This command lets you reopen the last running session.

  Note You can use this command to reopen the Simulation Manager, if you close the window unintentionally.

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
```

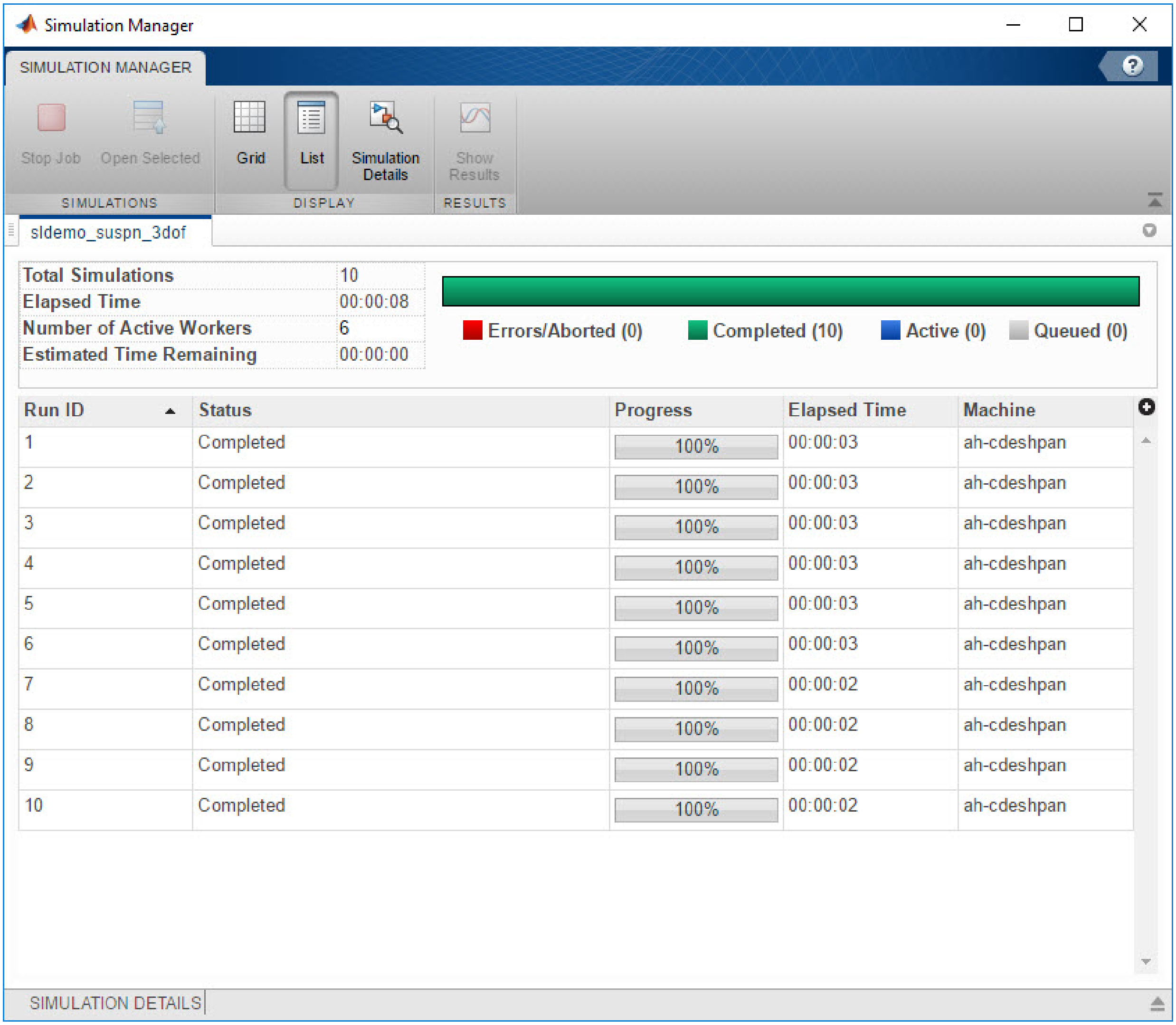
Run multiple simulations and open the Simulation Manager to monitor them.

```
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:



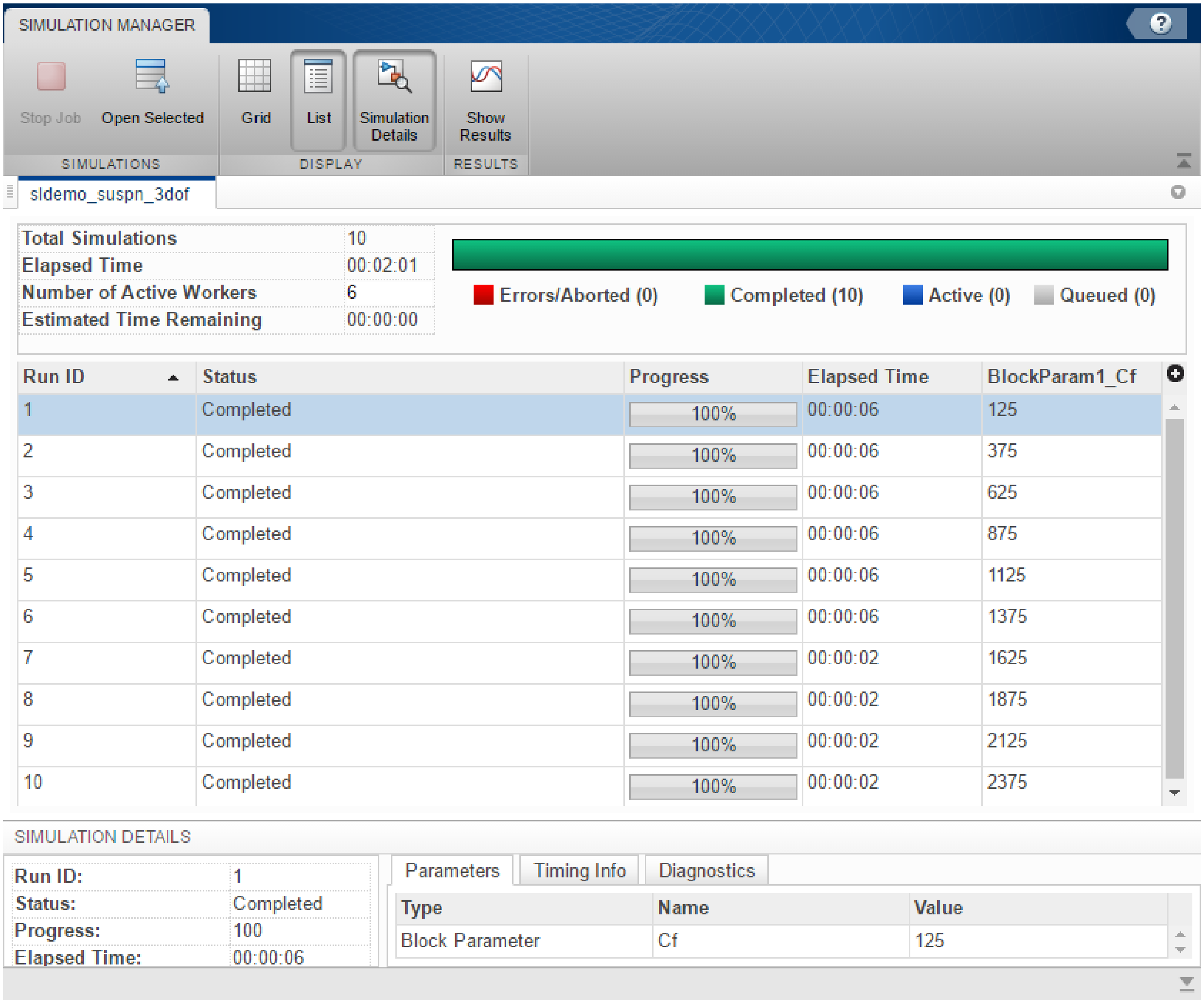
Total Simulations	10
Elapsed Time	00:00:08
Number of Active Workers	6
Estimated Time Remaining	00:00:00

🟥 Errors/Aborted (0) 🟩 Completed (10) 🟦 Active (0) ⬜ Queued (0)

Run ID ▲	Status	Progress	Elapsed Time	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

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. Use the button, ⊕, to choose which columns to display. You can also sort the columns based on your preference.



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)

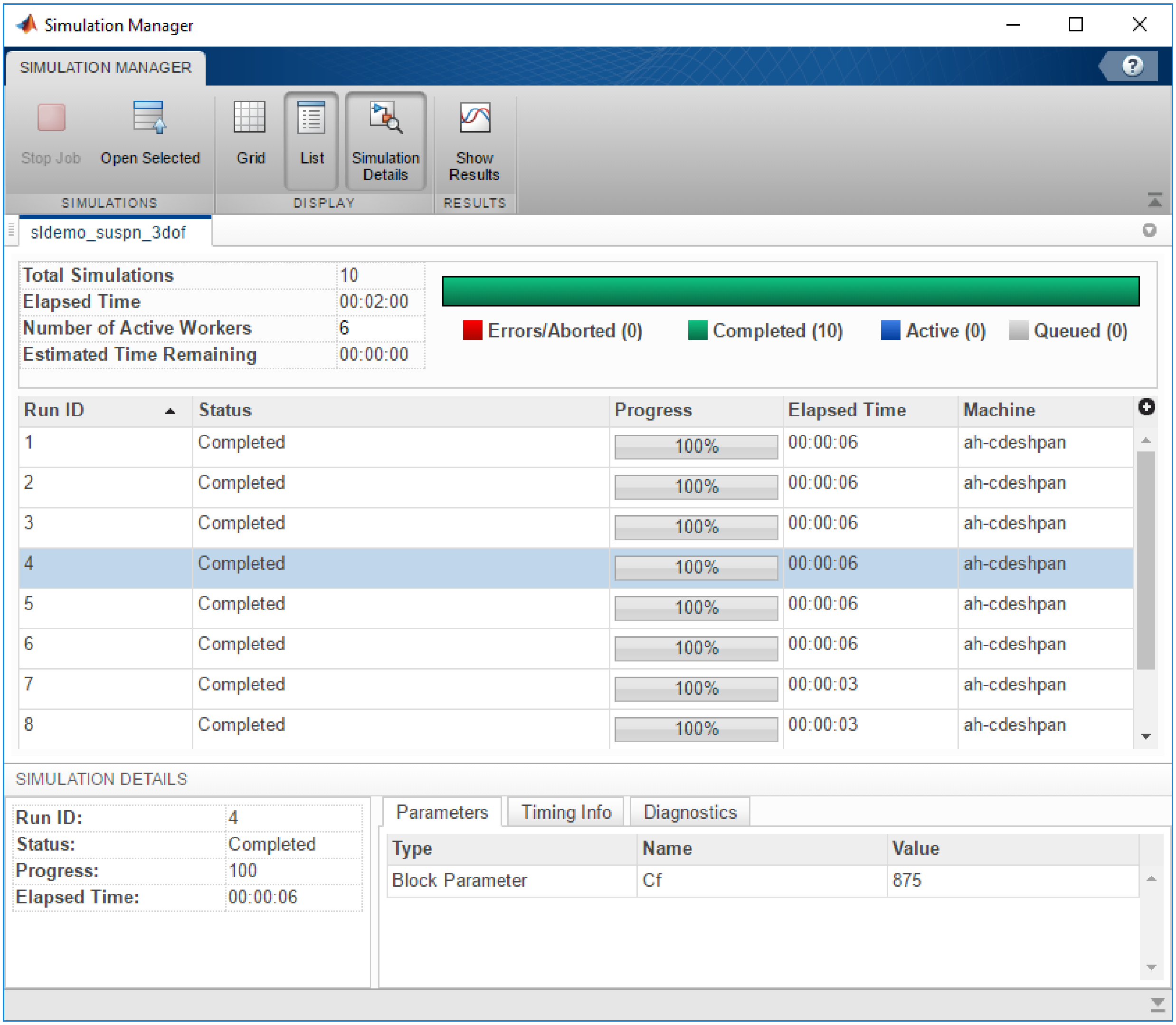
Run ID	Status	Progress	Elapsed Time	BlockParam1_Cf
1	Completed	100%	00:00:06	125
2	Completed	100%	00:00:06	375
3	Completed	100%	00:00:06	625
4	Completed	100%	00:00:06	875
5	Completed	100%	00:00:06	1125
6	Completed	100%	00:00:06	1375
7	Completed	100%	00:00:02	1625
8	Completed	100%	00:00:02	1875
9	Completed	100%	00:00:02	2125
10	Completed	100%	00:00:02	2375

SIMULATION DETAILS

Run ID:	1
Status:	Completed
Progress:	100
Elapsed Time:	00:00:06

Type	Name	Value
Block Parameter	Cf	125

You can view more information about a particular run by clicking on it. When you click on a run, simulation details appear at the bottom of the window.



Total Simulations	10
Elapsed Time	00:02:00
Number of Active Workers	6
Estimated Time Remaining	00:00:00

Run ID	Status	Progress	Elapsed Time	Machine
1	Completed	100%	00:00:06	ah-cdeshpan
2	Completed	100%	00:00:06	ah-cdeshpan
3	Completed	100%	00:00:06	ah-cdeshpan
4	Completed	100%	00:00:06	ah-cdeshpan
5	Completed	100%	00:00:06	ah-cdeshpan
6	Completed	100%	00:00:06	ah-cdeshpan
7	Completed	100%	00:00:03	ah-cdeshpan
8	Completed	100%	00:00:03	ah-cdeshpan

Run ID:	4
Status:	Completed
Progress:	100
Elapsed Time:	00:00:06

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 `Simulink.SimulationOutput` object and is displayed in Simulation Data Inspector. You can also return your results in the `Simulink.SimulationOutput` object.

To stop the job at the beginning of your simulations, you can use the **Stop Job** button, 🟥.

See Also

Functions
`Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `sim` | `validate`

Topics

"Run Parallel Simulations Using parsim"
"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 `slLibraryBrowser` 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

`slLibraryBrowser` | `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: [1×1 slproject.Category]
            Files: [1×31 slproject.ProjectFile]
        Shortcuts: [1×7 slproject.Shortcut]
      ProjectPath: [1×7 slproject.PathFolder]
ProjectReferences: [1×0 slproject.ProjectReference]
     StartupFiles: [1×0 slproject.StartupFile]
    ShutdownFiles: [1×0 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
isLoaded
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: [1×1 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”.

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.
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

Project Property	Description
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"
"Clone Git Repository or Check Out SVN Repository"
"What Are Simulink Projects?"

Introduced in R2012a

Simulink.allBlockDiagrams

Find loaded Simulink models and libraries

Syntax

```
bd = Simulink.allBlockDiagrams()
bd = Simulink.allBlockDiagrams(type)
```

Description

`bd = Simulink.allBlockDiagrams()` returns all loaded block diagrams, including models and libraries.

`bd = Simulink.allBlockDiagrams(type)` returns either models or libraries.

Examples

Find Loaded Models

Find all loaded models in the current Simulink session, excluding libraries. The example shows a result from a typical session.

```
Simulink.allBlockDiagrams('model')
```

```
ans =

  237.0001
   56.0001
    2.0001
```

Get Names of Loaded Block Diagrams

Find all loaded models in the current Simulink session and return results as names. Use `Simulink.allBlockDiagrams` with `get_param` to get the names. The example shows a result from a typical session and includes loaded libraries and models.

```
get_param(Simulink.allBlockDiagrams(),'Name')
```

```
ans =

  5×1 cell array

    {'simulink_extras'}
    {'simulink'       }
    {'sldemo_fuelsys' }
    {'f14'            }
    {'vdp'            }
```

Get Loaded Block Diagrams Based on Parameter

Find all loaded models in the current Simulink session whose `'Dirty'` parameter is `'on'`.

```
bds = Simulink.allBlockDiagrams();
dirtyBds = bds(strcmp(get_param(bds,'Dirty'),'on'));
```

Input Arguments

type — Type of block diagram whose blocks to return
`'model'` | `'library'`

Type of block diagram whose blocks to return, specified as `'model'` or `'library'`.

Output Arguments

bd — Loaded block diagrams
array of handles

Loaded block diagrams, returned as an array of handles.

See Also

`Simulink.FindOptions` | `Simulink.findBlocks` | `Simulink.findBlocksOfType`

Introduced in R2018a

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: `'slexMulticoreExample/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

signalBuilderToSignalEditor

Import signal data and properties from Signal Builder block to Signal Editor block

Syntax

```
signal_editor = signalBuilderToSignalEditor(signal_builder,Name,
Value)
[signal_editor,sorted_group_index,sorted_group_names] =
signalBuilderToSignalEditor(signal_builder,Name,Value)
```

Description

`signal_editor = signalBuilderToSignalEditor(signal_builder,Name,Value)` imports signal data and properties from Signal Builder block to the Signal Editor block. This function adds a Signal Editor block to the current model using the signal data and properties from the Signal Builder block.

During the port, the `signalBuilderToSignalEditor` function:

- Orders signal groups alphabetically.
- Removes spaces from group names.
- Creates unique group names from existing names following MATLAB conventions.

The `signalBuilderToSignalEditor` function does not support models that contain test case parameters. You can successfully port data from the Signal Builder block, but you cannot initialize parameters with the Signal Editor block in test harnesses generated by Simulink Design Verifier.

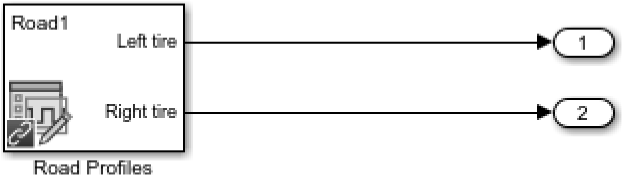
`[signal_editor,sorted_group_index,sorted_group_names] = signalBuilderToSignalEditor(signal_builder,Name,Value)` outputs vectors containing the signal groups and group names.

Examples

Replace Signal Builder Block with Signal Editor Block

This example shows how to replace an existing Signal Builder block with a Signal Editor block. To store signals from Signal Builder, the example uses `RoadProfiles.mat`.

```
model = 'ex_replace_signalbuilder';
open_system(model);
sbBlockH = [model '/Road Profiles'];
seBlockH = signalBuilderToSignalEditor(sbBlockH,...
'Replace',true,'FileName','RoadProfiles.mat');
```



Copyright 2018 The MathWorks, Inc.

Input Arguments

`signal_builder` — Signal Builder block to replace
current model (default) | scalar

Signal Builder block to replace, specified as a scalar.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'FileName','RoadProfiles.mat'

`FileName` — MAT-file that stores signals
`'dataset.mat'` (default) | scalar

MAT-file that stores signals and properties, specified as a scalar. Do not use a file name from one locale in a different locale. When using the block on multiple platforms, consider specifying just the MAT-file name and having the MAT-file be on the MATLAB path.

Data Types: `char` | `string`

`Replace` — Replace Signal Builder block with Signal Editor block
`false` (default) | `true`

Replace Signal Builder block with Signal Editor block, specified as `true` or `false`.

Data Types: `logical`

Output Arguments

`signal_editor` — Signal Editor block handle
scalar

Signal Editor block handle, specified as a scalar.

`sorted_group_index` — List of Signal Builder group indices
vector

List of Signal Builder group indices, specified as a vector and ordered as they will appear in the Signal Editor.

`sorted_group_names` — List of Signal Builder group names
cell array

Signal Editor group names, specified as a cell array of vectors, in alphabetical order.

The names are unique valid MATLAB variable names generated from the Signal Builder group names.

.

See Also

Signal Editor | Signal Builder | `signalEditor` | `signalbuilder`

Introduced in R2018a

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 Explicitly Partitioned Models 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 `1xn` 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 or string 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 or string that represents the annotation of a specific sample time (e.g., 'D1')
- `OwnerBlock` — For asynchronous and variable sample times, a character vector or string containing the full path to the block that controls the sample time. For all other types of sample times, an empty character vector or string.
- `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
- Permute Dimensions

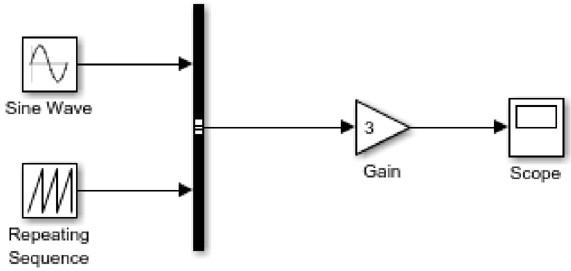
Examples

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'))
```

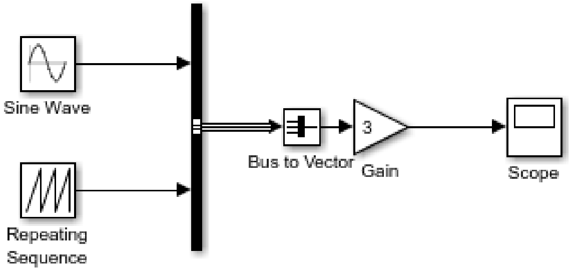


Insert a Bus to Vector block to convert the bus input signal for the Gain block to a vector signal because the Gain block can accept only non-bus signals.

```
[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
- Permute Dimensions

Specify as the fourth argument. You must also specify the `model`, `includeLibs`, and `reportOnly` 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:

<table>
<tr><th>Element</th><th colspan="2">Description</th></tr>
<tr><td>modelChecksum</td><td colspan="2">1x4 vector that encodes the structure of the model.</td></tr>
<tr><td rowspan="2">parameters</td><td colspan="2">A structure of the tunable parameters in the model. This structure contains the following fields.</td></tr>
<tr><td>Field</td><td>Description</td></tr>
</table>

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 `parameters` 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, `[startIdx, endIdx]`.
		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

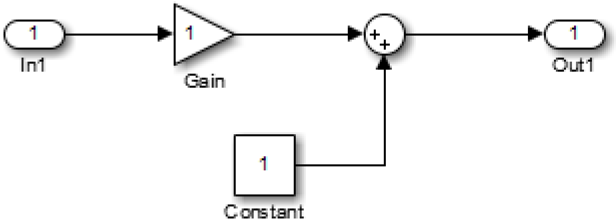
Default: `[]`

Examples

This function converts the contents of a model or subsystem into a subsystem.

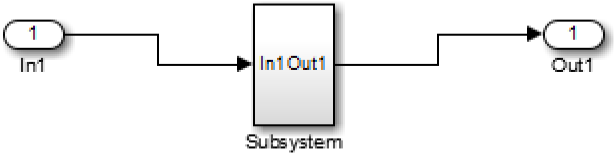
```
function convert2subsys(sys)
  blocks = find_system(sys, 'SearchDepth', 1);
  bh = [];
  for i = 2:length(blocks)
    bh = [bh get_param(blocks{i}, 'handle')];
  end
  Simulink.BlockDiagram.createSubsystem(bh);
end
```

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:

```
load_system('f14')
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 subsystem `block` into the system that contained the subsystem. Open or load the model first.

You can expand virtual subsystems that are not masked, linked, or commented. For details, see “Subsystems That You Can Expand”.

Input Arguments

`block` — Subsystem to expand
block path name | block handle

Subsystem to expand, specified as the block path name or block handle.

Example: `'sldemo_enginewc/Combustion'`

Examples

Expand Subsystem

Expand the Combustion subsystem. The blocks and signals that were in the Combustion subsystem become part of the system and replace the Subsystem block

Open the model `sldemo_enginewc`.

```
open_system('sldemo_enginewc')
```

The subsystem contents appear in the top level of the model.

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 or string 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 or string that represents the annotation of a specific sample time (e.g., `'D1'`).
- `OwnerBlock` — For asynchronous and variable sample times, a character vector or string containing the full path to the block that controls the sample time. For all other types of sample times, an empty character vector or string.
- `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_mdlref_depgraph')
```

Propagate a Configuration Reference to Listed Referenced Models

```
[isPropagated,convertedModels] = ...
Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','include',...
{'sldemo_mdlref_heater','sldemo_mdlref_house'})
```

Propagate a Configuration Reference to Referenced Models with Exclusions

```
[isPropagated,convertedModels] = ...
Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','exclude',...
{'sldemo_mdlref_heater','sldemo_mdlref_house'})
```

Open the Configuration Reference Propagation to Referenced Models Dialog Box for a Model

```
Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','gui')
```

Input Arguments

`model` — Top model

character vector | string scalar

Top model with configuration reference to propagate, specified as a character vector or string scalar.

Example: ‘mdl’

`refModels` — Referenced models

cell array of character vectors | string array

List of referenced models to be included or excluded in propagation, specified as a cell array of character vectors or string array.

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');
```

Input Arguments

`model` — Top model
character vector | string scalar

Name of top model, specified as a character vector or string scalar.

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.

Note If you specify a `.mat` extension when you save the active configuration set, all of the parameters are preserved. If you specify a `.m` extension, the `.m` file does not include hidden or disabled parameters.

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','Sample'}; ...
            {'b',1,'double', [0.2 0],'real','Sample',...
            'Fixed',-3,3,'m','b is distance from the origin'}},...
            }, ...
        };
```

Generate a bus object in the base workspace from the cell array.

```
Simulink.Bus.cellToObject(busCell)
```

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 must contain this bus object information:

1. Bus name
2. Header file
3. Description
4. Data scope
5. Alignment
6. Elements

The elements field is an array that must contain this data for each element:

1. Element name
2. Dimensions
3. Data type
4. Sample time
5. Complexity
6. Sampling mode

The elements field array can also contain this data:

1. Dimensions mode
2. Minimum
3. Maximum
4. Units
5. Description

Tips

The inverse function is `Simulink.Bus.objectToCell`.

Compatibility Considerations

`Simulink.BusElement` objects will no longer support the `SamplingMode` property

Not recommended starting in R2016b

In R2016b, the `SamplingMode` property of `Simulink.BusElement` objects was removed. Scripts that use the `SamplingMode` property of `Simulink.BusElement` objects continue to work. `Simulink.Bus.cellToObject` continues to require the `SamplingMode` field and `Simulink.Bus.objectToCell` continues to include the sampling mode in the output cell arrays.

In a future release, support for `SamplingMode` will be removed.

To specify whether a signal is sample-based or frame-based, define the sampling mode of input signals at the block level instead of at the signal level.

See Also

Classes

`Simulink.Bus` | `Simulink.BusElement`

Functions

`Simulink.Bus.createObject` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save`

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.

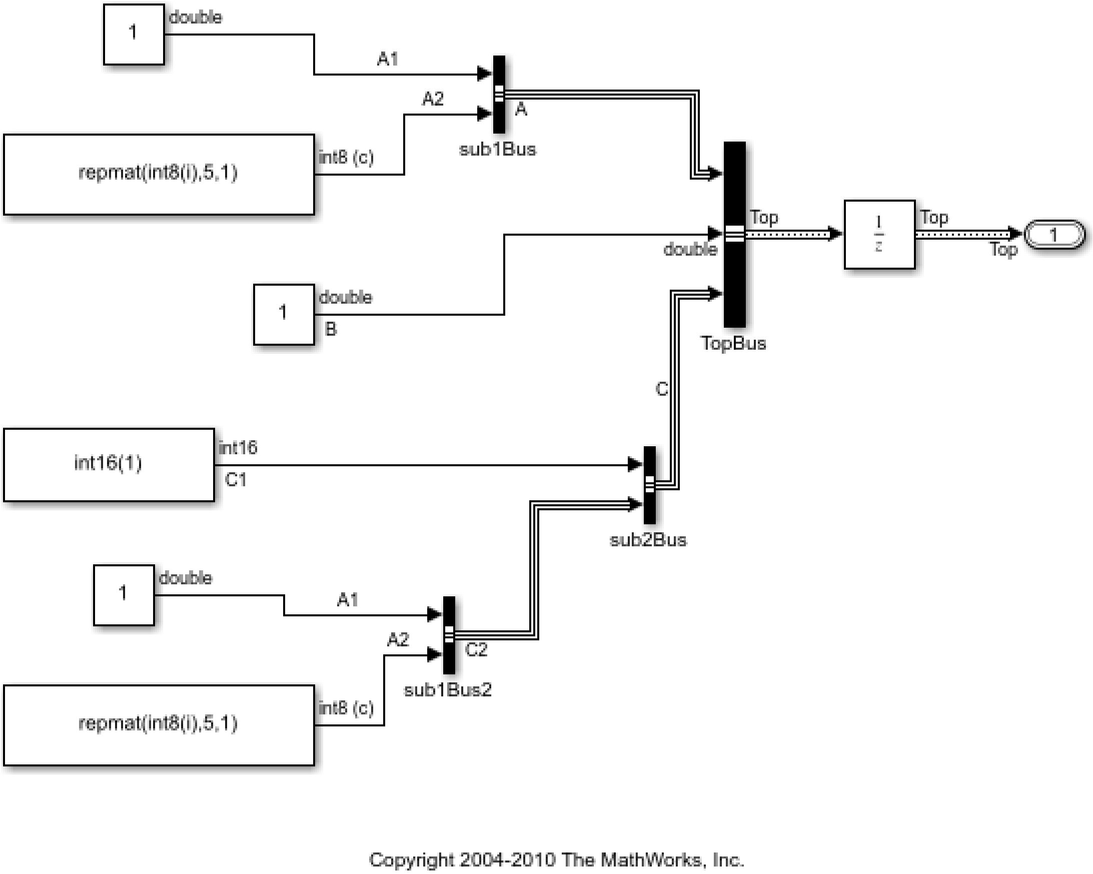
`structsForBuses = Simulink.Bus.createMATLABStruct(busObjectNames)` creates a cell array of structures based on the specified bus objects.

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')
```



Create a MATLAB® structure using the bus object `Top`, which the `busic_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.

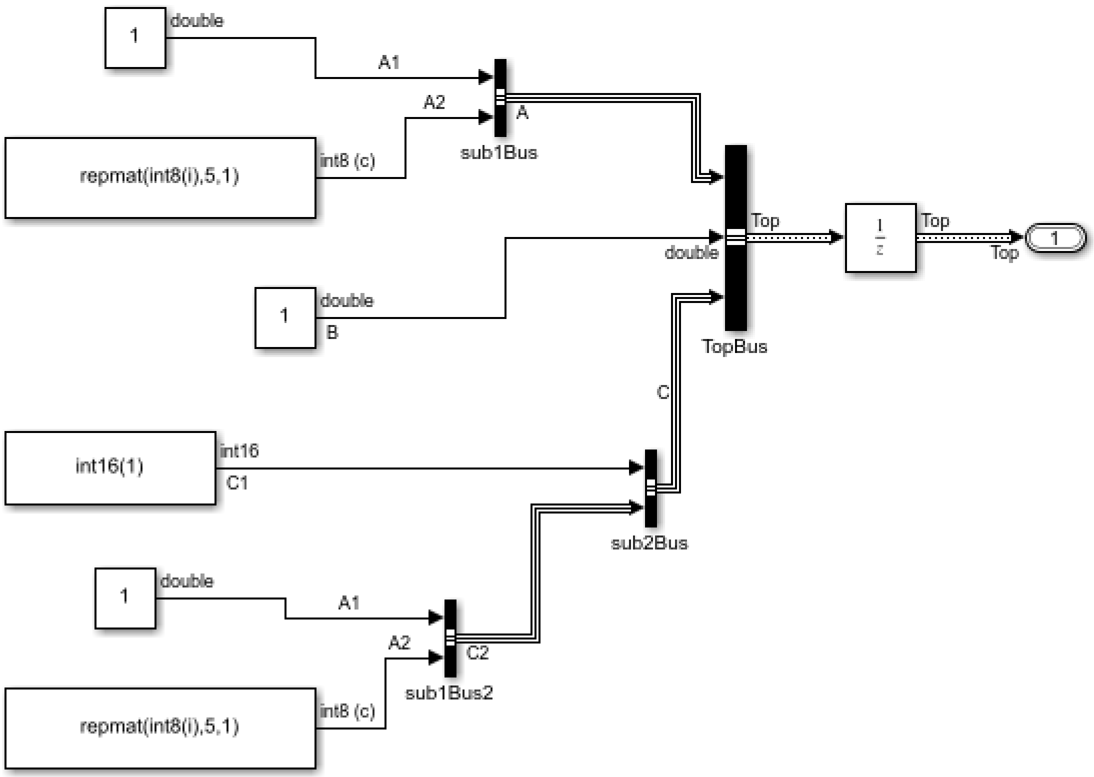
You can use `mStruct` as the initial condition structure for the Unit Delay block.

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.

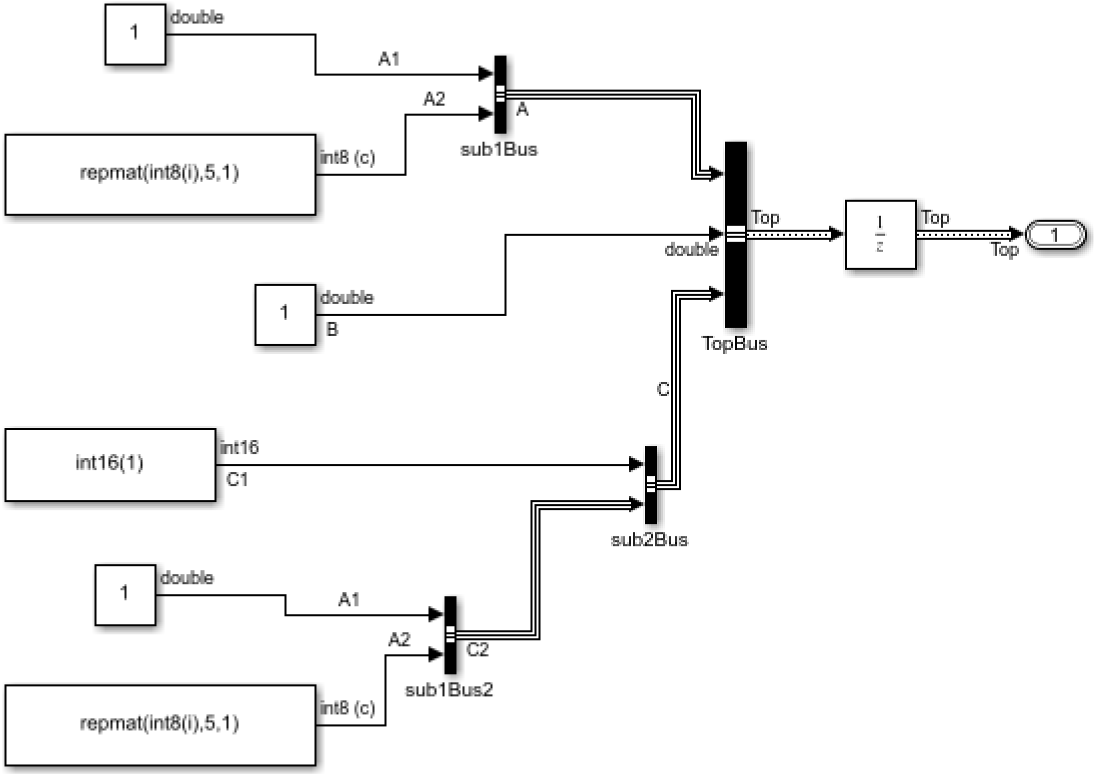
```
outPort = ph.Outport;
mStruct = Simulink.Bus.createMATLABStruct(outPort,PartialstructForK);
```

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','busic_example');
open_system(model);
sim('busic_example')
```



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 `busic_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
```

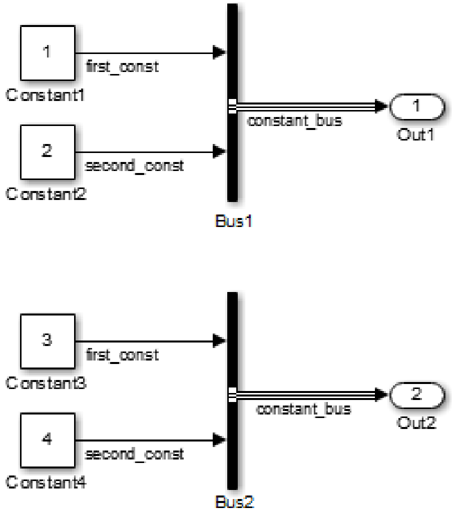
Close the system.

```
close_system('busic_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_outports_create_struct/Bus Creator','PortHandles')
ph_2 = get_param...
     ('ex_two_outports_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.

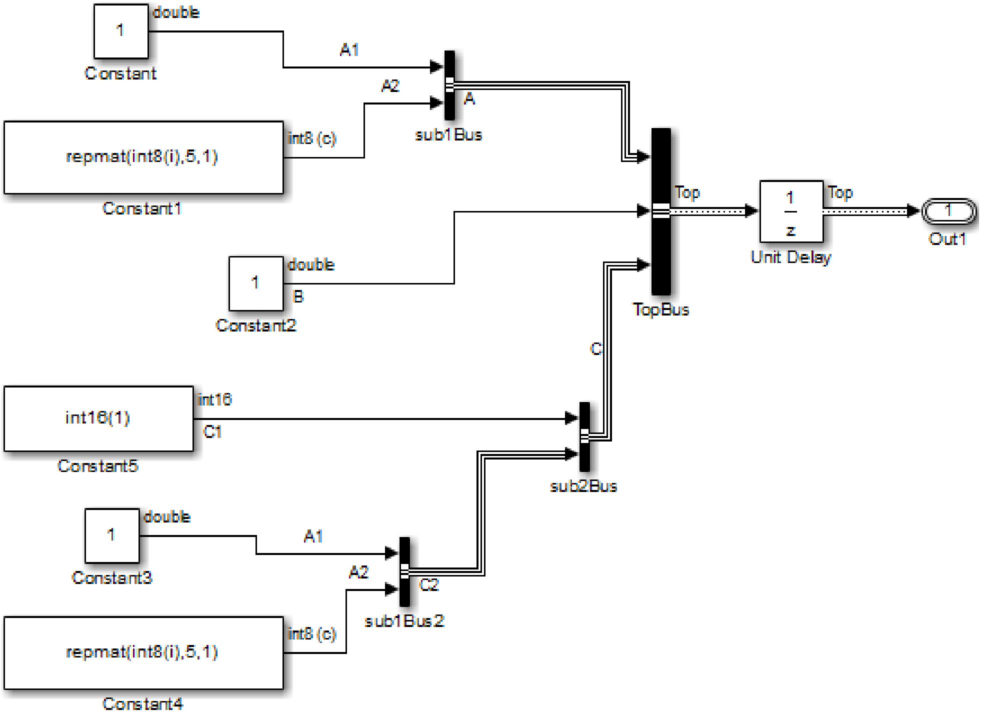
```
close_system('ex_two_outports_create_struct')
```

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/busic_example.mdl']);
sim('busic_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('busic_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 structure that can contain MATLAB `timeseries`, MATLAB `timetable`, and `matlab.io.datastore.SimulationDatastore` objects or a numeric 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

Create a Bus Object with Bus Creator Blocks

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;
```

To create a bus object, use the block handle variables in a vector.

```
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','Sample'}; ...
            {'b',1,'double', [0.2 0],'real','Sample',...
            'Fixed',-3,3,'m','b is distance from the origin'}},...
            }, ...
        };
```

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 objects

structure of MATLAB `timeseries`, MATLAB `timetable`, and `matlab.io.datastore.SimulationDatastore` objects | numeric structure

Structure used to create bus objects, specified as a structure that can contain MATLAB `timeseries`, MATLAB `timetable`, and `matlab.io.datastore.SimulationDatastore` 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 object 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

```
cells = Simulink.Bus.objectToCell(busNames)
cells = Simulink.Bus.objectToCell(busNames,scope)
```

Description

`cells = Simulink.Bus.objectToCell(busNames)` creates a cell array of bus information from a set of bus objects in the MATLAB base workspace. The cell array contains subordinate cell arrays that define each bus object. The order of elements in the output cell array corresponds to the order of names in the input cell array.

`cells = Simulink.Bus.objectToCell(busNames,scope)` specifies whether the bus objects reside in the MATLAB base workspace or a data dictionary.

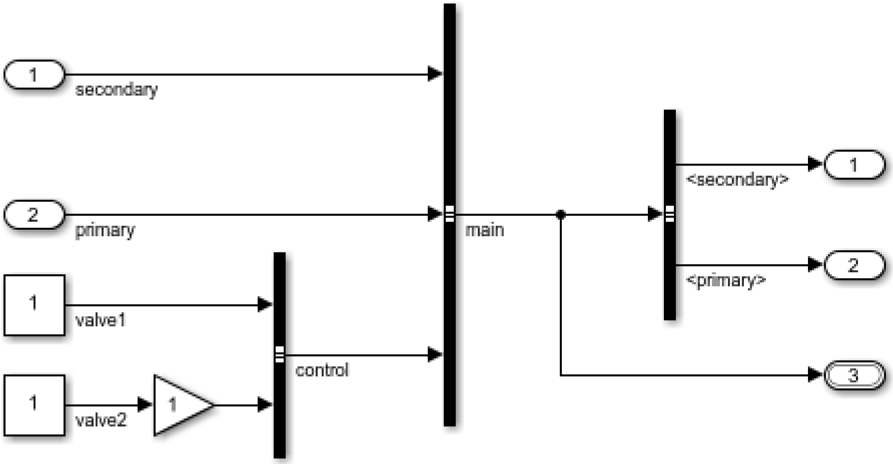
Examples

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 a cell array of information about the `CONTROL` bus object.

```
cells = Simulink.Bus.objectToCell({'CONTROL'});
cells{1}
```

```
ans =

  1x6 cell array

    {'CONTROL'}    {0x0 char}    {0x0 char}    {'Auto'}    {'-1'}    {2x1 cell}
```

Input Arguments

busNames — Names of bus objects

cell array of bus object names

Bus objects for which to create cell arrays of bus object information, specified as a cell array. Specify the bus object names as character vectors. If `busNames` is empty, the function converts all bus objects in the base workspace or data dictionary.

Example: `cells = Simulink.Bus.objectToCell({'busObject'})`

`scope` — Source of bus objects

character vector

Data dictionary or the MATLAB base workspace, specified as a character vector. If `scope` is empty, the function uses the base workspace.

Example: `cells = Simulink.Bus.objectToCell({},dataDictionaryObject)`

Output Arguments

`cells` — Bus object information

cell array of cell arrays

Bus object information, specified as a cell array of cell arrays. Each subordinate cell array contains this bus object information:

1. Bus name
2. Header file
3. Description
4. Data scope
5. Alignment
6. Elements

The elements field is an array that contains this data for each bus element:

1. Bus element name
2. Dimensions
3. Data type
4. Sample time
5. Complexity
6. Sampling mode

The elements field also contains this data if available:

1. Dimensions mode
2. Minimum

3. Maximum
4. Units
5. Description

Tips

The inverse function is `Simulink.Bus.cellToObject`.

Compatibility Considerations

`Simulink.BusElement` objects will no longer support the `SamplingMode` property

Not recommended starting in R2016b

In R2016b, the `SamplingMode` property of `Simulink.BusElement` objects was removed. Scripts that use the `SamplingMode` property of `Simulink.BusElement` objects continue to work. `Simulink.Bus.cellToObject` continues to require the `SamplingMode` field and `Simulink.Bus.objectToCell` continues to include the sampling mode in the output cell arrays.

In a future release, support for `SamplingMode` will be removed.

To specify whether a signal is sample-based or frame-based, define the sampling mode of input signals at the block level instead of at the signal level.

See Also

Classes
`Simulink.Bus` | `Simulink.BusElement`

Functions
`Simulink.Bus.cellToObject` | `Simulink.Bus.createObject` | `Simulink.Bus.save`

Topics

“When to Use Bus Objects”
“Create Bus Objects Programmatically”
“Create Bus Objects with the Bus Editor”
“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'});
```

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.clearIntEnumType

Delete enumeration classes defined by `Simulink.defineIntEnumType`

Syntax

```
Simulink.clearIntEnumType(typeName)
Simulink.clearIntEnumType()
```

Description

`Simulink.clearIntEnumType(typeName)` deletes a specific enumeration class that is defined by `Simulink.defineIntEnumType`. The function generates a warning if the class name is invalid or if a class cannot be deleted because instances of the class exist.

`Simulink.clearIntEnumType()` deletes all enumeration classes that are defined by `Simulink.defineIntEnumType`. The function generates a warning if a class cannot be deleted because instances of the class exist.

Examples

Delete a Specific Dynamic Enumerated Data Type

Define an enumeration type and confirm that it has been created.

```
Simulink.defineIntEnumType('myEnumType', {'e1', 'e2'}, [1 2]);
myResult = Simulink.findIntEnumType('myEnumType')
```

Delete the enumeration type that you created and confirm that it is no longer there.

```
Simulink.clearIntEnumType('myEnumType');
myResult = Simulink.findIntEnumType('myEnumType')
```

Delete All Dynamic Enumerated Data Types

Define two enumeration types and confirm that they have been created.

```
Simulink.defineIntEnumType('myEnumType1', {'e1', 'e2'}, [1 2]);
Simulink.defineIntEnumType('myEnumType2', {'e3', 'e4'}, [3 4]);
myResult = Simulink.findIntEnumType()
```

Delete all enumeration types and confirm that no enumeration types exist.

```
Simulink.clearIntEnumType();
myResult = Simulink.findIntEnumType()
```

Input Arguments

`typeName` — Name of enumeration class

character vector or string

Name of a specific enumeration class that is defined by `Simulink.defineIntEnumType`, specified as a character vector or string.

Example: `'myEnumType'`

Data Types: `char` | `string`

See Also

`enumeration` | `Simulink.defineIntEnumType` | `Simulink.findIntEnumType`

Introduced in R2018b

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','F
```

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 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)
```

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

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.closeAll` | `Simulink.data.dictionary.getOpenDictionaryPaths` | `Simulink.data.dictionary.setupWorkerCache` | `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.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
```

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:

          DataSources: {0x1 cell}
```

```
    HasUnsavedChanges: 0
          NumberOfEntries: 0
```

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.sldd'`

Example: `'C:\Users\jsmith\myDictionary.sldd'`

Example: `'..\myOtherDictionary.sldd'`

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
```

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:

          DataSources: {'myRefDictionary_ex_API.sldd'}
```

```
    HasUnsavedChanges: 0
          NumberOfEntries: 4
```

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.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

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)
```

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.

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.

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`

Specified value	Information returned	Example return value
`'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 *`model`*`_types.h` or imports the definition from the header file identified by `HeaderFile`. A return value of `'Exported'` or `'Imported'` indicates generated code exports or imports the definition to or from the header file identified by `HeaderFile`.	`'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)	If no value is specified for Headerfile, export the type definition to *model*_types.h, where *model* is the model name. If a value is specified for Headerfile, import the data type definition from the specified header file.
Exported	Export the data type definition to a header file. If no value is specified for Headerfile, the header file name defaults to *type*.h, where *type* is the data type name.
Imported	Import the data type definition from a header file. If no value is specified for Headerfile, the header file name defaults to *type*.h, where *type* is the data type name.

'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');
```

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
```

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 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);
```

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 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.fileGenControl

Specify root folders for files generated by diagram updates and model builds

Syntax

```
cfg = Simulink.fileGenControl('getConfig')
Simulink.fileGenControl(Action,Name,Value)
```

Description

`cfg = Simulink.fileGenControl('getConfig')` returns a handle to an instance of the `Simulink.FileGenConfig` object, which contains the current values of these file generation control parameters:

- `CacheFolder` – Specifies the root folder for model build artifacts that are used for simulation, including Simulink® cache files.
- `CodeGenFolder` – Specifies the root folder for code generation files.
- `CodeGenFolderStructure` – Controls the folder structure within the code generation folder.

To get or set the parameter values, use the `Simulink.FileGenConfig` object.

These Simulink preferences determine the initial parameter values for the MATLAB session:

- Simulation cache folder – `CacheFolder`
- Code generation folder – `CodeGenFolder`
- Code generation folder structure – `CodeGenFolderStructure`

`Simulink.fileGenControl(Action,Name,Value)` performs an action that uses the file generation control parameters of the current MATLAB session. Specify additional options with one or more `name,value` pair arguments.

Examples

Get File Generation Control Parameter Values

To obtain the file generation control parameter values for the current MATLAB session, use getConfig.

```
cfg = Simulink.fileGenControl('getConfig');

myCacheFolder = cfg.CacheFolder;
myCodeGenFolder = cfg.CodeGenFolder;
myCodeGenFolderStructure = cfg.CodeGenFolderStructure;
```

Set File Generation Control Parameters by Using Simulink.FileGenConfig Object

To set the file generation control parameter values for the current MATLAB session, use the setConfig action. First, set values in an instance of the Simulink.FileGenConfig object. Then, pass the object instance. This example assumes that your system has aNonDefaultCacheFolder and aNonDefaultCodeGenFolder folders.

```
% Get the current configuration
cfg = Simulink.fileGenControl('getConfig');

% Change the parameters to non-default locations
% for the cache and code generation folders
cfg.CacheFolder = fullfile('C:','aNonDefaultCacheFolder');
cfg.CodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');
cfg.CodeGenFolderStructure = 'TargetEnvironmentSubfolder';

Simulink.fileGenControl('setConfig', 'config', cfg);
```

Set File Generation Control Parameters Directly

You can set file generation control parameter values for the current MATLAB session without creating an instance of the Simulink.FileGenConfig object. This example assumes that your system has aNonDefaultCacheFolder and aNonDefaultCodeGenFolder folders.

```
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ...
   'CodeGenFolder', myCodeGenFolder, ...
   'CodeGenFolderStructure', ...
    Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);
```

If you do not want to generate code for different target environments in separate folders, for `'CodeGenFolderStructure'`, specify the value `Simulink.filegen.CodeGenFolderStructure.ModelSpecific`.

Reset File Generation Control Parameters

You can reset the file generation control parameters to values from Simulink preferences.

```
Simulink.fileGenControl('reset');
```

Create Simulation Cache and Code Generation Folders

To create file generation folders, use the `set` action with the `'createDir'` option. You can keep previous file generation folders on the MATLAB path through the `'keepPreviousPath'` option.

```
%
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', ...
   'CacheFolder',myCacheFolder, ...
   'CodeGenFolder',myCodeGenFolder, ...
   'keepPreviousPath',true, ...
   'createDir',true);
```

Input Arguments

Action — Specify action

`'reset'` | `'set'` | `'setConfig'`

Specify an action that uses the file generation control parameters of the current MATLAB session:

- `'reset'` – Reset file generation control parameters to values from Simulink preferences.
- `'set'` – Set file generation control parameters for the current MATLAB session by directly passing values.
- `'setConfig'` – Set file generation control parameters for the current MATLAB session by using an instance of a `Simulink.FileGenConfig` object.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Simulink.fileGenControl(`*`Action`*`, `*`Name`*`, `*`Value`*`);`

`config` — Specify instance of `Simulink.FileGenConfig`

object handle

Specify the `Simulink.FileGenConfig` object instance containing file generation control parameters that you want to set.

Option for `setConfig`.

Example: `Simulink.fileGenControl('setConfig', 'config', `*`cfg`*`);`

`CacheFolder` — Specify simulation cache folder

character vector

Specify a simulation cache folder path value for the `CacheFolder` parameter.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', `*`myCacheFolder`*`);`

`CodeGenFolder` — Specify code generation folder

character vector

Specify a code generation folder path value for the `CodeGenFolder` parameter. You can specify an absolute path or a path relative to build folders. For example:

- `'C:\Work\mymodelsimcache'` and `'/mywork/mymodelgencode'` specify absolute paths.
- `'mymodelsimcache'` is a path relative to the current working folder (`pwd`). The software converts a relative path to a fully qualified path at the time the `CacheFolder` or `CodeGenFolder` parameter is set. For example, if `pwd` is `'/mywork'`, the result is `'/mywork/mymodelsimcache'`.
- `'../test/mymodelgencode'` is a path relative to `pwd`. If `pwd` is `'/mywork'`, the result is `'/test/mymodelgencode'`.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CodeGenFolder', *myCodeGenFolder*);`

`CodeGenFolderStructure` — Specify generated code folder structure
`Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) | `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder`

Specify the layout of subfolders within the generated code folder:

- `Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) – Place generated code in subfolders within a model-specific folder.
- `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder` – If models are configured for different target environments, place generated code for each model in a separate subfolder. The name of the subfolder corresponds to the target environment.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', *myCacheFolder*, ... 'CodeGenFolder', *myCodeGenFolder*, ... 'CodeGenFolderStructure', ... Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);`

`keepPreviousPath` — Keep previous folder paths on MATLAB path
false (default) | true

Specify whether to keep the previous values of `CacheFolder` and `CodeGenFolder` on the MATLAB path:

- `true` – Keep previous folder path values on MATLAB path.

- `false` (default) – Remove previous older path values from MATLAB path.

Option for `reset`, `set`, or `setConfig`.

Example: `Simulink.fileGenControl('reset', 'keepPreviousPath', true);`

`createDir` — Create folders for file generation
false (default) | true

Specify whether to create folders for file generation if the folders do not exist:

- `true` – Create folders for file generation.
- `false` (default) – Do not create folders for file generation.

Option for `set` or `setConfig`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', *myCacheFolder*, 'CodeGenFolder', *myCodeGenFolder*, 'keepPreviousPath', true, 'createDir',true);`

Avoid Naming Conflicts

Using `Simulink.fileGenControl` to set `CacheFolder` and `CodeGenFolder` adds the specified folders to your MATLAB search path. This function has the same potential for introducing a naming conflict as using `addpath` to add folders to the search path. For example, a naming conflict occurs if the folder that you specify for `CacheFolder` or `CodeGenFolder` contains a model file with the same name as an open model. For more information, see “What Is the MATLAB Search Path?” (MATLAB) and “Files and Folders that MATLAB Accesses” (MATLAB).

To use a nondefault location for the simulation cache folder or code generation folder:

1. Delete any potentially conflicting artifacts that exist in:

   - The current working folder, `pwd`.
   - The nondefault simulation cache and code generation folders that you intend to use.
2. Specify the nondefault locations for the simulation cache and code generation folders by using `Simulink.fileGenControl` or Simulink preferences.

Output Arguments

`cfg` — Current values of file generation control parameters

object handle

Instance of a `Simulink.FileGenConfig` object, which contains the current values of file generation control parameters.

See Also

“Simulation cache folder” | “Code generation folder” | Code generation folder structure

Topics

“Manage Build Process Folders” (Simulink Coder)
“Share Simulation Builds for Faster Simulations”

Introduced in R2010b

Simulink.findBlocks

Find blocks in Simulink models

Syntax

```
bl = Simulink.findBlocks(sys)
bl = Simulink.findBlocks(sys,options)
bl = Simulink.findBlocks(sys,Param1,Value1,...,ParamN,ValueN)
bl = Simulink.findBlocks(sys,Param1,Value1,...,ParamN,ValueN,
options)
```

Description

`bl = Simulink.findBlocks(sys)` returns handles to all blocks in the model or subsystem `sys`.

`bl = Simulink.findBlocks(sys,options)` finds blocks that match the criteria specified by a `Simulink.FindOptions` object.

`bl = Simulink.findBlocks(sys,Param1,Value1,...,ParamN,ValueN)` finds blocks whose block parameters have the specified values.

`bl = Simulink.findBlocks(sys,Param1,Value1,...,ParamN,ValueN,options)` finds blocks whose parameters have the specified values and that match the criteria specified by a `FindOptions` object.

Examples

Find Blocks in a Model

Return handles for all blocks in the model `vdp`.

```
load_system(vdp);
bl = Simulink.findBlocks('vdp')
bl =

    5.0001
   83.0001
   86.0001
    6.0001
    7.0001
    8.0001
    9.0001
   10.0001
   11.0001
   12.0001
   13.0001
   14.0001
   15.0001
   16.0001
```

Return block names.

```
bl = getfullname(Simulink.findBlocks('vdp'))
```

```
bl =

  12×1 cell array

    {'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 block handles for the block whose name is `Mu`.

```
Simulink.findBlocks('vdp','Name','Mu')
```

```
ans =

    8.0001
```

Input Arguments

`sys` — Model or subsystem to find blocks in
character vector | string array

Model or subsystem to find blocks in, specified as a character vector or string array.

Example: `'vdp'` `"f14/Aircraft Dynamics Model"`

`options` — Search constraints
`Simulink.FindOptions` object

Search constraints, specified as a `Simulink.FindOptions` object.

Output Arguments

`bl` — Search results
array of handles

Search results, returned as an array of handles.

See Also

`Simulink.FindOptions` | `Simulink.allBlockDiagrams` | `Simulink.findBlocksOfType`

Topics

“Model Parameters” on page 6-2
“Block-Specific Parameters” on page 6-128

Introduced in R2018a

Simulink.findBlocksOfType

Find specified type of block in Simulink models

Syntax

```
bl = Simulink.findBlocksOfType(sys,type)
bl = Simulink.findBlocksOfType(sys,type,options)
bl = Simulink.findBlocksOfType(sys,
type,Param1,Value1,...,ParamN,ValueN)
bl = Simulink.findBlocksofType(sys,
type,Param1,Value1,...,ParamN,ValueN,options)
```

Description

`bl = Simulink.findBlocksOfType(sys,type)` returns handles to all blocks of the specified type in the model or subsystem `sys`.

`bl = Simulink.findBlocksOfType(sys,type,options)` matches the criteria specified by a `FindOptions` object.

`bl = Simulink.findBlocksOfType(sys, type,Param1,Value1,...,ParamN,ValueN)` finds blocks whose parameters have the specified values.

`bl = Simulink.findBlocksofType(sys, type,Param1,Value1,...,ParamN,ValueN,options)` finds blocks whose parameters have the specified values and that match the criteria specified by a `FindOptions` object.

Examples

Find Blocks of a Type in Model

Find all blocks of type Gain in the model `vdp`.

```
load_system('vdp');
Simulink.findBlocksOfType('vdp','Gain')

ans =

    7.0001
```

To return block names instead of handles, use `getfullname`.

```
getfullname(Simulink.findBlocksOfType('vdp','Gain'))

ans =

    'vdp/Mu'
```

Find Blocks of a Type Using Search Options

Load the model `sldemo_clutch`. Then, create a `FindOptions` object and use it to constrain the search of GoTo blocks in the model to the Unlocked system.

```
load_system('sldemo_clutch');
f = Simulink.FindOptions('SearchDepth',1);
bl = Simulink.findBlocksOfType('sldemo_clutch/Unlocked','Goto',f)

bl =

  166.0001
  167.0001
```

Input Arguments

sys — Model or subsystem to find blocks in

character vector | string array

Model or subsystem to find blocks in, specified as a character vector or string array.

Example: `'vdp'` `"f14/Aircraft Dynamics Model"`

type — Block type

character vector | string array

Block type, specified as a character vector or string array. Use `get_param` with the `'BlockType'` parameter to get the block type.

`options` — Search constraints
`simulink.FindOptions` object

Search constraints, specified as a `Simulink.FindOptions` object.

Example: `Simulink.FindOptions('SearchDepth',1)`

Output Arguments

`bl` — Search results
array of handles

Search results, returned as an array of handles.

See Also

`Simulink.FindOptions` | `Simulink.allBlockDiagrams` | `Simulink.findBlocks`

Topics

"Model Parameters" on page 6-2
"Block-Specific Parameters" on page 6-128

Introduced in R2018a

Simulink.findIntEnumType

Find enumeration classes defined by `Simulink.defineIntEnumType`

Syntax

```
result = Simulink.findIntEnumType(typeName)
result = Simulink.findIntEnumType()
```

Description

`result = Simulink.findIntEnumType(typeName)` returns the `meta.class` object for class `type` that is defined by `Simulink.defineIntEnumType`. Use the returned `meta.class` object to query attributes of the enumeration class. If the class does not exist, the function returns an empty `meta.class` object.

`result = Simulink.findIntEnumType()` returns `meta.class` objects for all enumeration classes that are defined by `Simulink.defineIntEnumType`. Use the returned `meta.class` objects to query attributes of the enumeration classes.

Examples

Find a Specific Dynamic Enumerated Data Type

Define an enumeration type.

```
Simulink.defineIntEnumType('myEnumType', {'e1', 'e2'}, [1 2]);
```

Check for the enumeration type that you have created .

```
myResult = Simulink.findIntEnumType('myEnumType')
```

```
myResult = 

  class with properties:
```

```
                     Name: 'myEnumType'
              Description: ''
      DetailedDescription: ''
                   Hidden: 0
                   Sealed: 0
                 Abstract: 0
              Enumeration: 1
          ConstructOnLoad: 0
         HandleCompatible: 0
          InferiorClasses: {[1×1 meta.class]}
        ContainingPackage: [0×0 meta.package]
     RestrictsSubclassing: 0
             PropertyList: [0×1 meta.property]
               MethodList: [150×1 meta.method]
                EventList: [0×1 meta.event]
    EnumerationMemberList: [2×1 meta.EnumeratedValue]
           SuperclassList: [1×1 meta.class]
```

Find All Dynamic Enumerated Data Types

Define two enumeration types.

```
Simulink.defineIntEnumType('myEnumType1', {'e1', 'e2'}, [1 2]);
Simulink.defineIntEnumType('myEnumType2', {'e3', 'e4'}, [3 4]);
```

Check for the enumeration types that you have created.

```
myResult = Simulink.findIntEnumType()
```

Input Arguments

typeName — Name of enumeration class
character vector or string

Name of a specific enumeration class that is defined by `Simulink.defineIntEnumType`, specified as a character vector or string.

Example: 'myEnumType'

Data Types: char | string

Output Arguments

`result` — Search results

array of `meta.class` objects

Search result, returned as an array of `meta.class` objects. If there are no enumeration classes, the array is empty.

See Also

`enumeration` | `Simulink.clearIntEnumType` | `Simulink.defineIntEnumType`

Introduced in R2018b

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');
```

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 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');
```

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 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'
```

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 2-696.

`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.

1. 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;
   ```
2. 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.
3. 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.

4 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 eumeration 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.

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.

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 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.sldd'

Example: 'myDict'

Example: `fullfile('subfolder','myDict.sldd')`

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.
`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

- If a MATLAB Function block or Stateflow chart in your model uses an imported enumeration or structure type, configure the model configuration parameters to include (`#include`) the type definition from your external header file. See “Control Imported Bus and Enumeration Type Definitions” (for a MATLAB Function block) and “Integrate Custom C/C++ Code for Simulation” (Stateflow) and “Integrate Custom Structures in Stateflow Charts” (Stateflow) (for a chart).
- 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	Loads all ancestors of the referenced model not previously loaded. If any ancestor model does not appear on the MATLAB path, the conversion fails. 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.
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 `Simulink.SimulationData.Dataset` 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 logsout.

1 Load the MAT-file.

2 Convert logsout to a dataset called myModel_dataset. (The elements information will be different for your data.)

```
dataset = logsout.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');
```

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')
```

To get the SimState of a block that is in a referenced model, specify the full path of the block relative to the root model.

```
rootModel = 'sldemo_fuelsys_dd';
opt = struct('SaveFinalState','on','SaveCompleteFinalSimState','on','StopTime','1');
simOut = sim(rootModel,opt);
x = simOut.xFinal;
blockPath = 'sldemo_fuelsys_dd/Fuel Rate Controller|sldemo_fuelsys_dd_controller/contro
chartState = getBlockSimState(x,blockPath)
```

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 `-regexps` *`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 `-regexps` *`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

Package: Simulink.sdi

Add one or more signals to existing run

Syntax

```
sigIDs = Simulink.sdi.addToRun(runID,'vars',var,var2,...,varn)
sigIDs = Simulink.sdi.addToRun(runID,'namevalue',sourceNames,
dataValues)
```

Description

`sigIDs = Simulink.sdi.addToRun(runID,'vars',var,var2,...,varn)` adds the data in the variables `var,var2,...,varn` to the run corresponding to the `runID` and returns the signal IDs for the signals added to the run.

`sigIDs = Simulink.sdi.addToRun(runID,'namevalue',sourceNames, dataValues)` adds the data in the cell array `dataValues` to the run corresponding to the `runID` and returns the signal IDs for the signals added to the run. The `sourceNames` argument specifies names to use for the source of the data in `dataValues` in the signal metadata.

Examples

Add Workspace Data to Run

This example shows how to use `Simulink.sdi.addToRun` to add workspace data to a run in the Simulation Data Inspector.

Generate Workspace Data

Generate workspace data to add to a simulation run in place of measured data, input data, or any other data that you want to associate with the simulation.

```
time = linspace(0, 60, 201);
cos_vals = 2*cos(2*pi/6.8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'cosine';
```

Simulate Model

Simulate the `slexAircraftExample` model to create a run containing the simulation outputs.

```
load_system('slexAircraftExample');
sim('slexAircraftExample','SaveFormat','Dataset');
```

Add Workspace Data to Simulation Run

Add the workspace data to the run. Then, view the results in the Simulation Data Inspector.

```
% Get run ID
count = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(count);

% Add data to run
sigIDs = Simulink.sdi.addToRun(runID,'vars',cos_ts);

Simulink.sdi.view
```

Add States Data to a Run

The model `slexAircraftExample` is configured to log outputs, states, and time data. The output data automatically logs to the Simulation Data Inspector as well as the base workspace, but the states data does not. To bring the states data into the Simulation Data Inspector, you can record the data, or you can add it to the run created by simulating the model. This example shows how to add logged states data to a Simulation Data Inspector run programmatically.

Simulate the Model and Get States Data

Simulate the model using the `sim` function with `'ReturnWorkspaceOutputs'` set to `'on'`. Select the states data, `xout`, from the simulation outputs.

```
load_system('slexAircraftExample')
```

```
simOut = sim('slexAircraftExample','ReturnWorkspaceOutputs','on',...
    'SaveFormat','Dataset');

% Get states data from simulation output
states = simOut.xout;
```

Get the Run ID

Because the outputs data automatically logs to the Simulation Data Inspector, a run is created upon simulating `slexAircraftExample`. Get the run ID for the run using the Simulation Data Inspector programmatic interface.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Add States Data to the Run

Add the states data to the run with the output data.

```
sigIDs = Simulink.sdi.addToRun(runID,'namevalue',{'States'},{states});
```

Input Arguments

`runID` — Run ID to add data to

scalar

Run ID for the signal you want to add data to. The Simulation Data Inspector assigns a unique run ID when it creates a run. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` and `Simulink.sdi.getRunIDByIndex`.

`var` — Data to add to run

variable

Workspace data to add to the run. `Simulink.sdi.addToRun` supports all loading and logging data formats, including `timeseries` and `Simulink.SimulationData.Dataset`. Provide one or more `var` inputs when you specify `'vars'` as the second argument.

Example: `myData`

`sourceNames` — Cell array of names for signal metadata

cell array of character vectors

Names to use as the source of the data in the metadata for the added signals. Provide a `sourceNames` input when you specify `'namevalue'` as the second argument.

Example: `{'speed','position'}`

`dataValues` — Cell array of data to add to run
cell array

Cell array of data to add to the run. Provide a `dataValues` input when you specify `'namevalue'` as the second argument.

Example: `{sig1,sig2}`

Output Arguments

`sigIDs` — Matrix containing signal IDs for added signals
matrix

Matrix of signal IDs for signals added to the run.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.copyRun` | `Simulink.sdi.createRun` | `Simulink.sdi.createRunOrAddToStreamedRun` | `add`

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 worker repository 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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an `spmd` code block, load the `slexAircraftExample` model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use `parfor` to run the seven simulations in parallel. Select the value for `Ts` for each simulation, and modify the value of `Ts` in the model workspace. Then, run the simulation and build an array of `Simulink.sdi.WorkerRun` objects to access the data with the Simulation Data Inspector. After the `parfor` loop, use another `spmd` segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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

Package: Simulink.sdi

Clear all data from the Simulation Data Inspector

Syntax

```
Simulink.sdi.clear
```

Description

`Simulink.sdi.clear` clears all data from the Simulation Data Inspector. `Simulink.sdi.clear` does not affect preferences or settings you have configured in the Simulation Data Inspector. Use `Simulink.sdi.clearPreferences` to reset the Simulation Data Inspector preferences to their default values.

Examples

Save a Simulation Data Inspector Session

This example shows how to create, save, and load a Simulation Data Inspector session. The example uses data logging to populate the Simulation Data Inspector with data and then uses the Simulation Data Inspector's programmatic interface to create plots to visualize the data. After saving the data and visualization settings in a session, the Simulation Data Inspector repository is emptied in order to demonstrate how to load the session.

Create Simulation Data

This example logs the `Stick`, `alpha, rad`, and `q, rad/sec` signals to generate simulation data using the model `slexAircraftExample` and creates two runs. The first uses a sine input, and the second has a square wave input.

```
% Ensure you start with an empty Simulation Data Inspector repository
Simulink.sdi.clear
```

```
% Load system
load_system('slexAircraftExample')

% Configure signals to log
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Change Pilot signal to sine
set_param('slexAircraftExample/Pilot','WaveForm','sine')

% Simulate model
sim('slexAircraftExample')

% Change Pilot signal to square
set_param('slexAircraftExample/Pilot','WaveForm','square')

% Simulate model
sim('slexAircraftExample')
```

Access Simulation Data

Use the Simulation Data Inspector programmatic interface to access the simulation data so you can create plots to visualize the signals.

```
% Get run objects
runIDs = Simulink.sdi.getAllRunIDs;
sineRunID = runIDs(end-1);
squareRunID = runIDs(end);

sineRun = Simulink.sdi.getRun(sineRunID);
squareRun = Simulink.sdi.getRun(squareRunID);

% Get signal objects
sineOut = sineRun.getSignalByIndex(1);
sineIn = sineRun.getSignalByIndex(3);

squareOut = squareRun.getSignalByIndex(1);
squareIn = squareRun.getSignalByIndex(3);
```

Create Plots in the Simulation Data Inspector

Use the programmatic interface to visualize the signal data from the two simulation runs. You can set the plot layout and plot signals on specific subplots.

```
% Set subplot layout
Simulink.sdi.setSubPlotLayout(2,1)

% Plot sine data on top plot
sineIn.plotOnSubPlot(1,1,true)
sineOut.plotOnSubPlot(1,1,true)

% Plot square wave data on bottom plot
squareIn.plotOnSubPlot(2,1,true)
squareOut.plotOnSubPlot(2,1,true)
```

Save a Simulation Data Inspector Session

First, view the plots you just created. Then, save the Simulation Data Inspector session as an MLDATX-file to recover your data along with your preference selections and plots.

```
% View the visualized data in the Simulation Data Inspector
Simulink.sdi.view

% Save the Simulation Data Inspector session
Simulink.sdi.save('myData.mldatx')
```

Load a Simulation Data Inspector Session

First, clear the Simulation Data Inspector repository with `Simulink.sdi.clear` and reset visualization settings with `Simulink.sdi.clearPreferences`. Then, you can load the session to see how the data and settings are preserved.

```
% Clear Simulation Data Inspector repository and preferences
Simulink.sdi.clear
Simulink.sdi.clearPreferences

% Load session file to view data
Simulink.sdi.load('myData.mldatx');
```

See Also

`Simulink.sdi.clearPreferences` | `Simulink.sdi.deleteRun` | `Simulink.sdi.deleteSignal`

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);
fuelRun = Simulink.sdi.getRun(runID);
fuelRun.name

ans =
'sldemo_fuelsys Run 1'

% Clear preferences to reset the run naming rule
Simulink.sdi.clearPreferences

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'
```

See Also

`Simulink.sdi.setMarkersOn` | `Simulink.sdi.setRunNamingRule` | `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
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.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 = sinusoidsRun.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 = sinusoidsRun.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.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,Name,Value)
```

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,Name,Value)` compares the simulation runs corresponding to `runID1` and `runID2` using the options specified by one or more `Name,Value` pair arguments 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 analyze the comparison results with the Simulation Data Inspector programmatic interface.

Generate Runs of Simulation Data

Simulate the model with different `Desired relative slip` values to create runs of simulation data to analyze with the Simulation Data Inspector programmatic interface.

```
% Open model
load_system('ex_sldemo_absbrake')

% Set the desired slip ratio 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 with a Global Tolerance

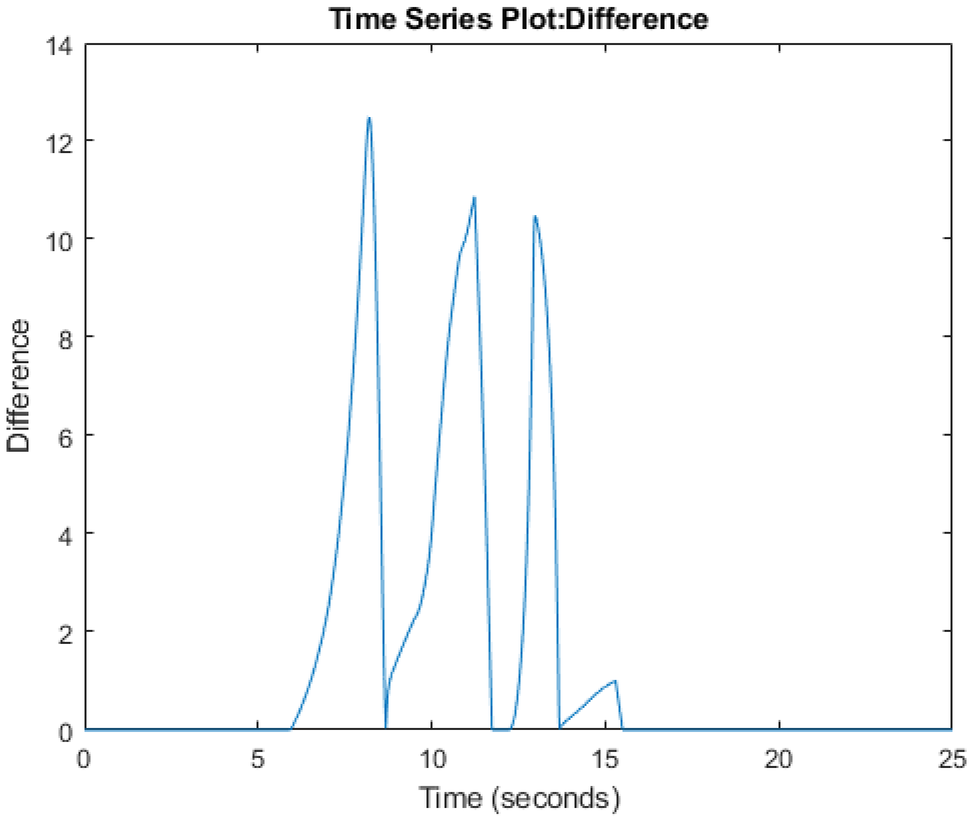
Get the run IDs for the runs you just created with the `Simulink.sdi.getAllRunIDs` function. Then, compare the runs using a global relative tolerance and a global time tolerance to analyze whether your data meets specifications.

```
% 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,'reltol',0.2,'timetol',0.5);
```

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.

Setup

Load the model, and mark the `q, rad/sec` and `alpha, rad` signals for logging. Then, simulate the model to create the baseline run.

```
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Simulate system
sim('slexAircraftExample')
```

Modify Model Parameter

Modify the model parameter Ts in the model workspace to change the time constant of the input low-pass filter.

```
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',1)

% Simulate again
sim('slexAircraftExample')
```

Compare Runs and Inspect Results

Use the Simulink.sdi.compareRuns function to compare the data from the simulations. Then, inspect the match property of the signal result to see whether the signals fell within the default tolerance of 0.

```
% 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
sig1Result1 = diffRun1.getResultByIndex(1);
sig2Result1 = diffRun1.getResultByIndex(2);

% Check whether signals matched
sig1Result1.Match
```

```
ans = logical
   0
```

```
sig2Result1.Match
```

```
ans = logical
   0
```

Compare Runs with Signal Tolerances

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

```
% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sigID2 = run1.getSignalIDByIndex(2);

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

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = diffRun2.getResultByIndex(1);
sig2Result2 = diffRun2.getResultByIndex(2);

% Check the result
sig1Result2.Match
```

```
ans = logical
   1
```

```
sig2Result2.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
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
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
```

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.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'abstol',x,'align',alignOpts`

`align` — Signal alignment options

array

Array specifying alignment algorithms to use for pairing signals from 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
`Simulink.sdi.AlignType.SID`	Simulink identifier “Locate Diagram Components Using Simulink Identifiers”

Value	Aligns By
`Simulink.sdi.AlignType.SignalName`	Signal name
`Simulink.sdi.AlignType.DataSource`	Path of the variable in the MATLAB workspace

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

`abstol` — Absolute tolerance for comparison

0 (default) | scalar

Positive-valued global absolute tolerance used for all signals in the run comparison. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: `0.5`

Data Types: `double`

`reltol` — Relative tolerance for comparison

0 (default) | scalar

Positive-valued global relative tolerance used for all signals in the run comparison. The relative tolerance is expressed as a fractional multiplier. For example, `0.1` specifies a 10 percent tolerance. For more information about how the relative tolerance is applied in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: `0.1`

Data Types: `double`

`timetol` — Time tolerance for comparison

0 (default) | scalar

Positive-valued global time tolerance used for all signals in the run comparison. Specify the time tolerance with units of seconds. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: `0.2`

Data Types: `double`

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

Package: Simulink.sdi

Compare data from two signals

Syntax

```
diff = Simulink.sdi.compareSignals(sigID1,sigID2)
```

Description

`diff = Simulink.sdi.compareSignals(sigID1,sigID2)` compares the signals corresponding to the signal IDs `sigID1` and `sigID2` and returns the results in a `Simulink.sdi.DiffSignalResult` object. For more information on how the comparison results are computed, see “How the Simulation Data Inspector Compares Data”.

Examples

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of the input and output signals for a control system. 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
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = aircraftRun.getSignalIDByIndex(1);
signalID2 = aircraftRun.getSignalIDByIndex(2);

if (aircraftRun.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (aircraftRun.isValidSignalID(signalID1) && aircraftRun.isValidSignalID(signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signals match within tolerance
    match = sigDiff.match
end
```

```
match = logical
   0
```

Compare Signals from Different Runs

This example shows how to compare signals from different simulation runs using the Simulation Data Inspector's `Simulink.sdi.compareSignals` function. When you only have one signal of interest to compare, using a signal comparison returns the `Simulink.sdi.diffSignalResult` object with the comparison data directly.

Generate Simulation Data

Use the `slexAircraftExample` model to generate simulation runs. Between the runs, change the time constant of the input filter.

```
% Load example model
load_system('slexAircraftExample')
```

```
% Mark the alpha, rad signal for streaming
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Simulate system
sim('slexAircraftExample')

% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',0.2)

% Simulate again
sim('slexAircraftExample')
```

Get Signal IDs for the Signal Comparison

Create run objects using the run IDs, and then use `getSignalIDByIndex` to get the signal IDs to pass to `Simulink.sdi.compareSignals`.

```
% Get run data
runIDs = Simulink.sdi.getAllRunIDs;

runID1 = runIDs(end-1);
runID2 = runIDs(end);

run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

sigID1 = run1.getSignalIDByIndex(1);
sigID2 = run2.getSignalIDByIndex(1);
```

Compare Signals

Compare the signals, and open the Simulation Data Inspector to view the results.

```
diffResult = Simulink.sdi.compareSignals(sigID1,sigID2);

Simulink.sdi.view
```

Input Arguments

sigID1 — Signal ID of Baseline signal
scalar

Signal ID for the **Baseline** signal. The Simulation Data Inspector assigns signal and run IDs when you create a run for your data. You can access the signal ID from a `Simulink.sdi.Run` object with the `getSignalIDByIndex` method.

`sigID2` — Signal ID of Compare to signal

scalar

Signal ID for the **Compare to** signal. The Simulation Data Inspector assigns signal and run IDs when you create a run for your data. You can access the signal ID from a `Simulink.sdi.Run` object with the `getSignalIDByIndex` method.

Output Arguments

`diff` — Comparison results

'Simulink.sdi.diffSignalResult'

`Simulink.sdi.DiffSignalResult` object containing the results of the signal comparison.

See Also

`Simulink.sdi.DiffSignalResult` | `Simulink.sdi.Run` | `Simulink.sdi.compareRuns`

Topics

"Inspect and Compare Data Programmatically"
"Compare Simulation Data"
"How the Simulation Data Inspector Compares Data"

Introduced in R2011b

Simulink.sdi.copyRun

Copy a Simulation Data Inspector run

Syntax

```
newRunID = Simulink.sdi.copyRun(runID)
[newRunID,runIndex] = Simulink.sdi.copyRun(runID)
[newRunID,runIndex,signalIDs] = Simulink.sdi.copyRun(runID)
```

Description

`newRunID = Simulink.sdi.copyRun(runID)` copies the run corresponding to `runID` and returns the run ID for the new run. The new run includes all the simulation data and metadata from the original run. You can modify the copy of the run by adding or deleting signals and metadata while still retaining the original run.

`[newRunID,runIndex] = Simulink.sdi.copyRun(runID)` copies the run corresponding to `runID` and returns the run ID and index in the Simulation Data Inspector repository for the new run. The new run includes all the simulation data and metadata from the original run.

`[newRunID,runIndex,signalIDs] = Simulink.sdi.copyRun(runID)` copies the run corresponding to `runID` and returns the signal IDs for the signals in the new run along with its run ID and index in the Simulation Data Inspector repository. The new run includes all the simulation data and metadata from the original run.

Examples

Compare a Subset of Signals

This example shows how to use `Simulink.sdi.copyRun` and `Simulink.sdi.deleteSignal` to create a copy of a run that contains a subset of the signals from the original run. You can use the copy to analyze and run comparisons on a subset of signals while still holding onto the original run that has all of the signals. For example, the model sldemo_fuelsys is configured to log ten signals. To compare the system's responses to different types of failures, you don't need to run the comparison on all of the logged signals. Deleting signals that do not represent the system's response before running the comparison saves processing time and simplifies the view of the results.

Create Runs

Load the model sldemo_fuelsys and run simulations to create runs in the Simulation Data Inspector. The first run simulates a failure of the throttle angle sensor, and the second run simulates a failure of the exhaust gas oxygen sensor.

```
load_system('sldemo_fuelsys')
modelWorkspace = get_param('sldemo_fuelsys','modelworkspace');
modelWorkspace.assignin('throttle_sw',0)
modelWorkspace.assignin('ego_sw',1)
sim('sldemo_fuelsys')

modelWorkspace.assignin('throttle_sw',1)
modelWorkspace.assignin('ego_sw',0)
sim('sldemo_fuelsys')
```

Copy the Run

Use the Simulation Data Inspector's programmatic interface to get Simulink.sdi.Run objects for the simulations, and then create copies of the runs.

```
% Get runs
runIDs = Simulink.sdi.getAllRunIDs;

runID1 = runIDs(end-1);
runID2 = runIDs(end);

run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

% Create a copy of each run, truncRun
[truncRun1,runIndex1,signalIDs1] = Simulink.sdi.copyRun(runID1);
[truncRun2,runIndex2,signalIDs2] = Simulink.sdi.copyRun(runID2);
```

Delete Signals in Run Copy

The sldemo_fuelsys model is configured to log the values of the fault switches along with several signals representing the system's response. When you compare the system's response when a throttle angle sensor fails to its response when an exhaust gas oxygen sensor fails, comparing the fault switch states does not provide new information. Therefore, delete the switch signals before running the comparison to eliminate unnecessary computations.

```
Simulink.sdi.deleteSignal(signalIDs1(1))
Simulink.sdi.deleteSignal(signalIDs1(3))
Simulink.sdi.deleteSignal(signalIDs1(5))
Simulink.sdi.deleteSignal(signalIDs1(8))

Simulink.sdi.deleteSignal(signalIDs2(1))
Simulink.sdi.deleteSignal(signalIDs2(3))
Simulink.sdi.deleteSignal(signalIDs2(5))
Simulink.sdi.deleteSignal(signalIDs2(8))
```

Compare Truncated Runs

You can use the truncated runs you created with `Simulink.sdi.copyRun` and `Simulink.sdi.deleteSignal` to perform a comparison of the system's response to different types of failures. Then, open the Simulation Data Inspector to view the comparison results.

```
truncRunDiff = Simulink.sdi.compareRuns(truncRun1,truncRun2);

Simulink.sdi.view
```

Input Arguments

`runID` — Numeric run identifier

scalar

Run ID for the run you want to copy. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

Output Arguments

`newRunID` — Run ID for the copy

scalar

Run ID for the copy of the run.

`runIndex` — Run index for the copy

scalar

Index of the copy in the Simulation Data Inspector repository.

`signalIDs` — Numeric identifiers for the signals in the copy

matrix

Matrix containing the signal IDs for the copies of signals created in the copy of the run.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.deleteRun` | `Simulink.sdi.deleteSignal`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2011b

Simulink.sdi.copyRunViewSettings

Copy line style and color for signals from one run to another

Syntax

```
sigIDs = Simulink.sdi.copyRunViewSettings(run1,run2,plot)
```

Description

`sigIDs = Simulink.sdi.copyRunViewSettings(run1,run2,plot)` copies the line style and color specifications from `runID1` to `runID2` for matched signals. You can specify `run1` and `run2` with their run ID or as a `Simulink.sdi.Run` object. If `plot` is specified as `true`, `Simulink.sdi.copyRunViewSettings` also changes signal parameters in both runs so that aligned signals that are plotted come from the `run2`. The function returns an array of signal identifiers for the signals that the Simulation Data Inspector aligned between the two runs. To learn more about how the Simulation Data Inspector aligns signals between runs, see “How the Simulation Data Inspector Compares Data”.

Examples

Copy View Settings to a Run

This example shows how to copy the view settings for aligned signals from one run to another.

Simulate Your Model and get Run Object

Simulate the `vdp` model to create a run of data to visualize.

```
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
sim('vdp')
```

```
runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify View Settings for Signals

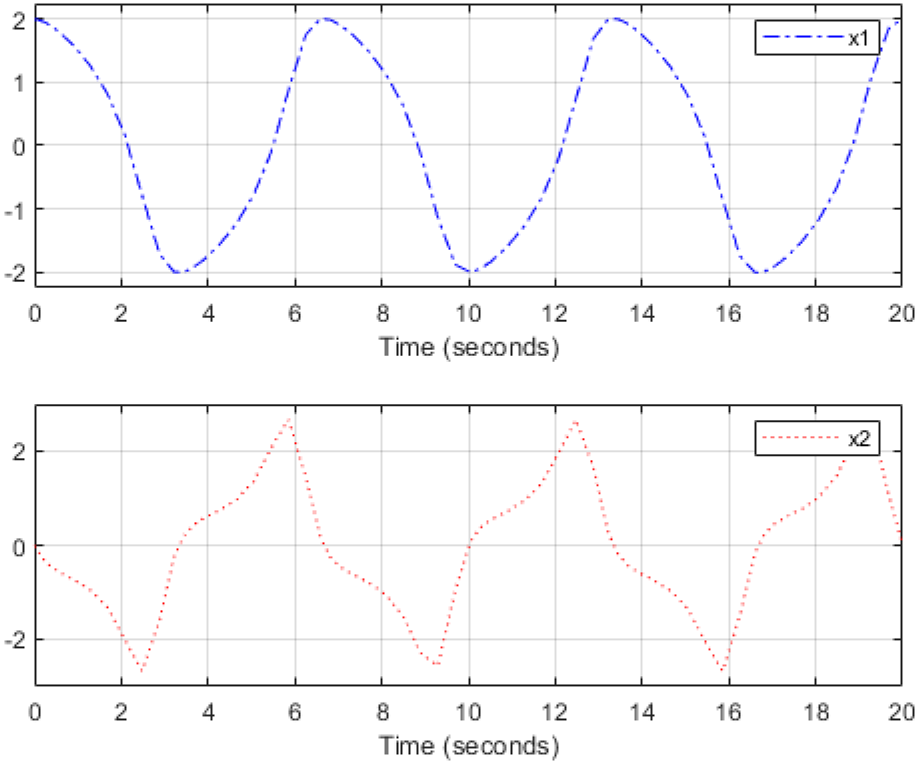
Use the `Simulink.sdi.Run` object to access the signals in the run. Then, modify the signals' view settings, and plot them in the Simulation Data Inspector. Open the Simulation Data Inspector and use `Simulink.sdi.snapshot` to view the results.

```
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

Capture a Snapshot from the Simulation Data Inspector

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```
snap = Simulink.sdi.CustomSnapshot;

snap.Rows = 2;
snap.YRange = {[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```



Copy the View Settings to a New Simulation Run

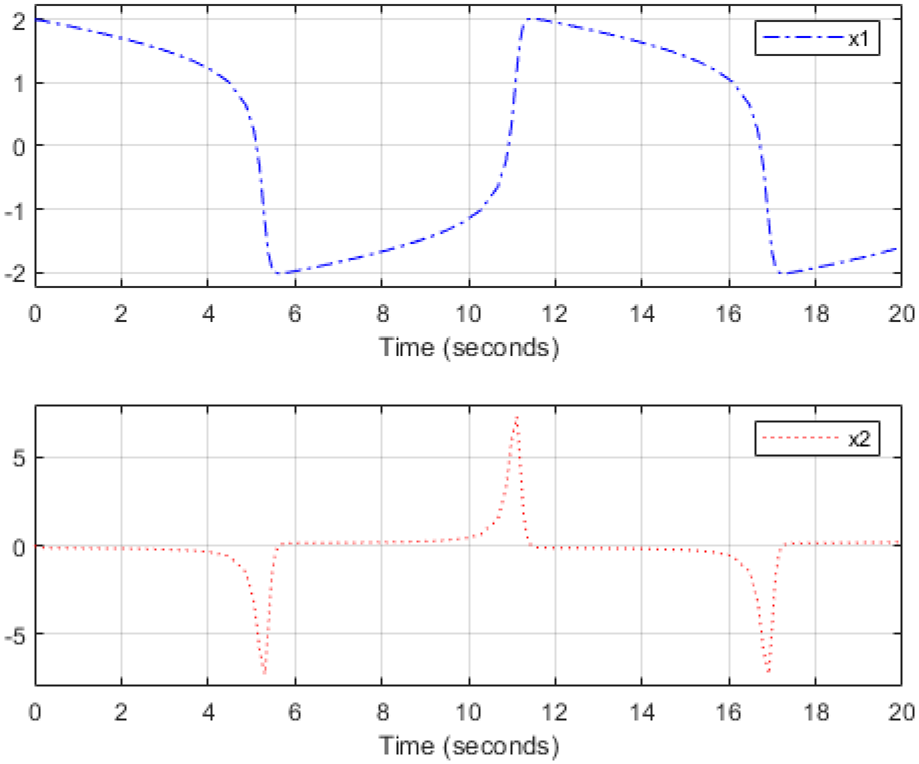
Simulate the model again, with a different Mu value. Then, visualize the new run by copying the view settings from the first run. Specify the plot input as true to plot the signals from the new run.

```
set_param('vdp/Mu','Gain','5')
sim('vdp')

runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);

sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,true);
```

Capture a Snapshot of the New Simulation Run

Use the `Simulink.sdi.CustomSnapshot` object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```
snap.clearSignals
snap.YRange = {[-2.25 2.25],[-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```



Input Arguments

run1 — Simulation Data Inspector run ID for source run

scalar | `'Simulink.sdi.Run'` object

Run with the view settings you want to copy specified with its run ID or `Simulink.sdi.Run` object. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

run2 — Simulation Data Inspector run ID for destination run

scalar | `'Simulink.sdi.Run'` object

Run you want to copy the view settings to, specified with its run ID or `Simulink.sdi.Run` object. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

`plot` — Specify whether to update plotted signals

`true` | `false`

Specifies whether the Simulation Data Inspector changes the plot settings in the runs corresponding to `run1` and `run2`.

- When `plot` is `true`, the Simulation Data Inspector modifies the signal parameters so that the aligned signals that are plotted come from `run2`.
- When `plot` is `false`, the Simulation Data Inspector does not change which signals are plotted.

Data Types: `logical`

Output Arguments

`sigIDs` — Signal IDs for aligned signals

matrix

Matrix containing the signal IDs for signals in `run2` that aligned with signals in `run1` and had view settings modified.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.Signal` | `Simulink.sdi.copyRun` | `Simulink.sdi.setMarkersOn` | `Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2016a

Simulink.sdi.createRun

Package: Simulink.sdi

Create a run in the Simulation Data Inspector

Syntax

```
runID = Simulink.sdi.createRun
runID = Simulink.sdi.createRun(runName)
runID = Simulink.sdi.createRun(var)

runID = Simulink.sdi.createRun(runName,'vars',var,var2,...,varn)
runID = Simulink.sdi.createRun(runName,'namevalue',sourceNames,
sigValues)
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 repository and returns the run ID. You can use `Simulink.sdi.getRun` to get a `Simulink.sdi.Run` object for the run, which allows you to add metadata and signals to the run.

`runID = Simulink.sdi.createRun(runName)` creates an empty run named `runName`.

`runID = Simulink.sdi.createRun(var)` creates a run named `var` containing the data in the workspace variable `var`.

`runID = Simulink.sdi.createRun(runName,'vars',var,var2,...,varn)` creates a run named `runName` containing data from one or more variables in the base workspace. The signals in the run take their names from the variable names.

runID = Simulink.sdi.createRun(runName,'namevalue',sourceNames, sigValues) creates a run named runName with the data in the cell array sigValues. The cell array of sourceNames specifies names to use as the source for the sigValues data.

runID = Simulink.sdi.createRun(runName,'file',filename) creates a run with data from the MAT, CSV, MDF, or Microsoft Excel file specified by filename.

[runID,runIndex] = Simulink.sdi.createRun(___) returns the run ID and run index for the run created in the Simulation Data Inspector repository.

[runID,runIndex,signalIDs] = Simulink.sdi.createRun(___) returns the run ID, run index within the Simulation Data Inspector, and the signal IDs for the signals in the run.

Examples

Create a Run in the Simulation Data Inspector

This example shows several ways to create a run in the Simulation Data Inspector for your data. You can create a run from simulation outputs, workspace data, and from a file.

Create Data

You can create Simulation Data Inspector runs from workspace data or from a file. The namevalue syntax for Simulink.sdi.createRun can be useful for hierarchical data. Create example data to use in each scenario.

```
% Create timeseries workspace data
time = linspace(0,20,101);

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 Dataset workspace data
```

```
sinusoids_ds = Simulink.SimulationData.Dataset;
sinusoids_ds = sinusoids_ds.add(cos_ts);
sinusoids_ds = sinusoids_ds.add(sine_ts);

doubSine = 2*sine_ts;
doubCos = 2*cos_ts;

doubSinusoids_ds = Simulink.SimulationData.Dataset;
doubSinusoids_ds = doubSinusoids_ds.add(doubSine);
doubSinusoids_ds = doubSinusoids_ds.add(doubCos);

% Save workspace data to a MAT-file
save sinusoids.mat sine_ts cos_ts
```

Create a Run with a Run Object

You can create a run object using the `Simulink.sdi.Run` object's `create` method and then add data to the run using the `add` method. The `add` method supports all loading and logging data formats.

```
% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals of different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars',sine_ts,cos_ts);
```

Create a Run for a Variable

When you have only one signal of interest that you want to inspect and compare in the Simulation Data Inspector you can use a simple syntax to create the run. The run takes the same name as the variable.

```
runID = Simulink.sdi.createRun(sine_ts);
```

Create a Named Empty Run

With this syntax, you can create an empty run and specify its name. Then, you can use the returned runID to add data to the run.

```
runID = Simulink.sdi.createRun('My Waves');
signalID = Simulink.sdi.addToRun(runID,'vars',cos_ts);
```

Create a Run from Multiple Workspace Variables

When your signals of interest are in multiple variables in your workspace, use the 'vars' syntax. You can also use this syntax to provide a custom name for a run created from a single variable.

```
runID = Simulink.sdi.createRun(' My Sinusoids','vars',sine_ts,cos_ts);
```

Create a Run and Specify the Source for the Data

Providing a name for the source of the run data can be helpful, particularly with hierarchical data. Use the 'namevalue' syntax to specify data source names.

```
runID = Simulink.sdi.createRun('Waves','namevalue',{'Sinusoids',...
    'BigSinusoids'},{sinusoids_ds,doubSinusoids_ds});
```

Create a Run from Data in a File

You can create a run in the Simulation Data Inspector directly from a file of data using the 'file' syntax.

```
runID = Simulink.sdi.createRun('Wave Data','file','sinusoids.mat');
```

View Runs in the Simulation Data Inspector

Look at the runs in the Inspect pane to see the results from each run creation method. You can select a run or a signal to view its metadata.

```
Simulink.sdi.view
```

Create a Simulation Data Inspector Run and Access Signal Data

This example shows how to access signal data when you create a run in the Simulation Data Inspector.

Generate Data for Run

For this example, create timeseries data for sine and cosine signals.

```
% Create timeseries workspace data
time = linspace(0, 20, 101);

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 Run and Return Signal IDs

You can use the `Simulink.sdi.createRun` syntax with multiple return arguments to get the signal IDs more directly instead of accessing the signal IDs through a `Simulink.sdi.Run` object.

```
[runID,runIndex,sigIDs] = Simulink.sdi.createRun('Sinusoids','vars',...
    sine_ts,cos_ts);

cosID = sigIDs(2);
cosSig = Simulink.sdi.getSignal(cosID);
```

Modify Signal Properties and View in the Simulation Data Inspector

You can use the `Simulink.sdi.Signal` object to view and modify signal properties and to plot signals in the Simulation Data Inspector.

```
cosSig.Checked = true;
cosSig.AbsTol = 0.05;
Simulink.sdi.view
cosSig.Name
```

```
ans =

    'Cosine, T=8'
```

Input Arguments

`runName` — Name for the run

character vector

Name for the run in the Simulation Data Inspector.

Example: `'Baseline Simulation'`

`var` — Variable name
variable

Variable in the base workspace containing the data you want to create a run for in the Simulation Data Inspector. The Simulation Data Inspector requires an associated time vector for your data. `Simulink.sdi.createRun` supports all loading and logging data formats, including `timeseries` and `Simulink.SimulationData.Dataset`.

Provide one `var` argument when `var` is the only argument and one or more `var` arguments when you specify `'vars'` for the second argument.

Example: `myData`

`sourceNames` — Cell array of signal names
cell array

Cell array of character vectors in which each element is the name of the source for data in the run. Provide a `sigNames` input when you specify `'namevalue'` for the second argument.

Example: `{'sig1','sig2'}`

`sigValues` — Signal data
cell array

Cell array of variables containing signal data for the run. Provide a `sigValues` input when you specify `'namevalue'` for the second argument.

Example: `{var1,var2}`

`filename` — Name of file containing data for run
character vector

File name of the file containing your run data. Provide a `filename` input when you specify `'file'` for the second argument. You can create a run from a MAT, CSV, or Microsoft Excel file. You can also create a run from an MDF-file with an mdf, mf4, mf3, data, or dat file extension.

Example: `'simulation.mat'`

Output Arguments

`runID` — Run identifier
scalar

Run identifier for the new run.

`runIndex` — Run index
scalar

Index of the new run in the Simulation Data Inspector repository.

`signalIDs` — Signal IDs for signals in run
vector

Vector containing the signal IDs for the signals in the run.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.addToRun` | `Simulink.sdi.copyRun` | `Simulink.sdi.createRunOrAddToStreamedRun` | `Simulink.sdi.deleteRun` | `Simulink.sdi.getRun`

Topics

"Inspect and Compare Data Programmatically"
"View Data with the Simulation Data Inspector"

Introduced in R2011b

Simulink.sdi.createRunOrAddToStreamedRun

Package: Simulink.sdi

Create a single run for all simulation outputs

Syntax

```
runID = Simulink.sdi.createRunOrAddToStreamedRun(mdl,runName,
varSources,varValues)
```

Description

`runID = Simulink.sdi.createRunOrAddToStreamedRun(mdl,runName, varSources,varValues)` creates a run with the data `varValues` if no run exists in the Simulation Data Inspector repository for the model `mdl`. If one or more runs for the model `mdl` exist in the Simulation Data Inspector repository, the function adds `varValues` to the most recent run associated with `mdl`. The run is named according to `runName`, and the sources for the data in `varValues` are named according to `varSources`.

Examples

Add Signals to a Run

This example shows how to use `Simulink.sdi.createRunOrAddToStreamedRun` to add data to an existing run for a model. In this example, you add logged states data to the run created through simulation.

Simulate the Model

Simulate the model to generate data. The model `slexAircraftExample` is configured to log outputs, so the Simulation Data Inspector automatically creates a run with the logged outport data. Using this simulation syntax, `out` contains the outport data (`yout`) and the states data (`xout`).

```
load_system('slexAircraftExample')
out = sim('slexAircraftExample','ReturnWorkspaceOutputs','on',...
            'SaveFormat','Dataset');
```

Add Logged States Data to Run

The Simulation Data Inspector automatically created a run for the logged outport data. Add the logged state data to the existing run using `Simulink.sdi.createRunOrAddToStreamedRun`.

```
Simulink.sdi.createRunOrAddToStreamedRun('slexAircraftExample','Run 1',...
              {'out'},{out});
```

Open the Simulation Data Inspector to View Results

Using `Simulink.sdi.createRunOrAddToStreamedRun` avoids redundancy in the data shown in the Simulation Data Inspector. Using `Simulink.sdi.createRun` to bring the states data into the Simulation Data Inspector creates a second run. `Simulink.sdi.addToRun` creates a duplicate signal from the outport data. Using `Simulink.sdi.createRunOrAddToStreamedRun`, you can include all simulation data in a single run without duplicating any signals.

```
Simulink.sdi.view
```

Input Arguments

`mdl` — Name of model that created simulation data

character vector

Name of the model the simulation data is from, specified as a character vector.

Example: `'sldemo_absbrake'`

`runName` — Name for the run

character vector

Name for the new or augmented run. If `Simulink.sdi.createRunOrAddToStreamedRun` adds data to an existing run, the run is renamed according to `runName`.

Example: `'Run 1'`

varSources — Names to use for the sources of data

cell array of character vectors

Names for the sources of the data in varValues.

Example: {'sig1','sig2'}

varValues — Data to add to run

cell array

Cell array of data to incorporate into the run. Simulink.sdi.createRunOrAddToStreamedRun supports data in all logging and loading formats, including timeseries and Simulink.SimulationData.Dataset.

Example: {sig1,sig2}

Output Arguments

runID — Run identifier

scalar

Run identifier for the new or augmented run.

See Also

Simulink.sdi.Run | Simulink.sdi.addToRun | Simulink.sdi.createRun | Simulink.sdi.getAllRunIDs | Simulink.sdi.getRun | Simulink.sdi.getRunCount | Simulink.sdi.isValidRunID | Simulink.sdi.view

Topics

“Inspect and Compare Data Programmatically”
“View Data with the Simulation Data Inspector”

Introduced in R2017a

Simulink.sdi.deleteRun

Package: Simulink.sdi

Delete a run from the Simulation Data Inspector repository

Syntax

```
Simulink.sdi.deleteRun(runID)
```

Description

`Simulink.sdi.deleteRun(runID)` deletes the run corresponding to `runID`. When you delete a run, the indices of all runs following the deleted run change to account for the change in the run count. Deleting a run does not change any run IDs.

Examples

Delete a Run

You can delete a run from the Simulation Data Inspector repository to free up memory space or to declutter your workspace from data you do not need.

```
% Load and simulate sldemo_fuelsys model
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')

% Get the run ID for the run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

```
% Delete the run
Simulink.sdi.deleteRun(runID)
```

Input Arguments

`runID` — Run identifier

scalar

Run ID for the run you want to delete. You can get the run ID for a run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

See Also

`Simulink.sdi.clear` | `Simulink.sdi.copyRun` | `Simulink.sdi.deleteSignal` | `Simulink.sdi.getAllRunIDs` | `Simulink.sdi.getArchiveRunLimit` | `Simulink.sdi.getRunIDByIndex` | `Simulink.sdi.setArchiveRunLimit`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2011b

Simulink.sdi.deleteSignal

Package: Simulink.sdi

Delete a signal from the Simulation Data Inspector repository

Syntax

```
Simulink.sdi.deleteSignal(sigID)
```

Description

`Simulink.sdi.deleteSignal(sigID)` deletes the signal corresponding to `sigID` from the Simulation Data Inspector repository.

Examples

Compare a Subset of Signals

This example shows how to use `Simulink.sdi.copyRun` and `Simulink.sdi.deleteSignal` to create a copy of a run that contains a subset of the signals from the original run. You can use the copy to analyze and run comparisons on a subset of signals while still holding onto the original run that has all of the signals. For example, the model `sldemo_fuelsys` is configured to log ten signals. To compare the system's responses to different types of failures, you don't need to run the comparison on all of the logged signals. Deleting signals that do not represent the system's response before running the comparison saves processing time and simplifies the view of the results.

Create Runs

Load the model `sldemo_fuelsys` and run simulations to create runs in the Simulation Data Inspector. The first run simulates a failure of the throttle angle sensor, and the second run simulates a failure of the exhaust gas oxygen sensor.

```
load_system('sldemo_fuelsys')
modelWorkspace = get_param('sldemo_fuelsys','modelworkspace');
modelWorkspace.assignin('throttle_sw',0)
modelWorkspace.assignin('ego_sw',1)
sim('sldemo_fuelsys')

modelWorkspace.assignin('throttle_sw',1)
modelWorkspace.assignin('ego_sw',0)
sim('sldemo_fuelsys')
```

Copy the Run

Use the Simulation Data Inspector's programmatic interface to get `Simulink.sdi.Run` objects for the simulations, and then create copies of the runs.

```
% Get runs
runIDs = Simulink.sdi.getAllRunIDs;

runID1 = runIDs(end-1);
runID2 = runIDs(end);

run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

% Create a copy of each run, truncRun
[truncRun1,runIndex1,signalIDs1] = Simulink.sdi.copyRun(runID1);
[truncRun2,runIndex2,signalIDs2] = Simulink.sdi.copyRun(runID2);
```

Delete Signals in Run Copy

The `sldemo_fuelsys` model is configured to log the values of the fault switches along with several signals representing the system's response. When you compare the system's response when a throttle angle sensor fails to its response when an exhaust gas oxygen sensor fails, comparing the fault switch states does not provide new information. Therefore, delete the switch signals before running the comparison to eliminate unnecessary computations.

```
Simulink.sdi.deleteSignal(signalIDs1(1))
Simulink.sdi.deleteSignal(signalIDs1(3))
Simulink.sdi.deleteSignal(signalIDs1(5))
Simulink.sdi.deleteSignal(signalIDs1(8))

Simulink.sdi.deleteSignal(signalIDs2(1))
Simulink.sdi.deleteSignal(signalIDs2(3))
Simulink.sdi.deleteSignal(signalIDs2(5))
Simulink.sdi.deleteSignal(signalIDs2(8))
```

Compare Truncated Runs

You can use the truncated runs you created with `Simulink.sdi.copyRun` and `Simulink.sdi.deleteSignal` to perform a comparison of the system's response to different types of failures. Then, open the Simulation Data Inspector to view the comparison results.

```
truncRunDiff = Simulink.sdi.compareRuns(truncRun1,truncRun2);

Simulink.sdi.view
```

Input Arguments

`sigID` — Signal ID

scalar

Unique number identifying the signal within the Simulation Data Inspector repository. You can get the signal ID for a signal as a return from `Simulink.sdi.createRun` or using the `Simulink.sdi.Run` object's methods.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.Signal` | `Simulink.sdi.copyRun` | `Simulink.sdi.createRun`

Topics

“Inspect and Compare Data Programmatically”
“Organize Your Simulation Data Inspector Workspace”

Introduced in R2016a

Simulink.sdi.enablePCTSupport

Control how the Simulation Data Inspector works with the Parallel Computing Toolbox

Syntax

```
Simulink.sdi.enablePCTSupport(mode)
```

Description

`Simulink.sdi.enablePCTSupport(mode)` enables support for automatic data import from parallel workers into the Simulation Data Inspector, according to the mode specified by `mode`. You can configure the Simulation Data Inspector to import no worker data, only data from local workers, or data from all workers — local and remote. You can also configure the parallel worker support as manual, where you manually select runs to import to the Simulation Data Inspector using the `Simulink.sdi.sendWorkerRunToClient` function. By default, the Simulation Data Inspector automatically imports runs from local workers.

Examples

Enable Parallel Support for All Workers

Configure Simulation Data Inspector parallel worker support to import the output automatically from both local and remote workers.

```
Simulink.sdi.enablePCTSupport('all')
```

Disable Support for Parallel Workers

To prevent the output from any Parallel Computing Toolbox workers from automatically importing to the Simulation Data Inspector, disable Parallel Computing Toolbox support.

```
Simulink.sdi.enablePCTSupport('none')
```

Manually Send Runs from Parallel Workers to the Simulation Data Inspector

This example shows how to use `Simulink.sdi.sendWorkerRunToClient` to send runs created using parallel workers manually to the Simulation Data Inspector.

Setup

This example runs several simulations of the `vdp` model, varying the value of the gain, `Mu`. To set up for the parallel simulation, define a vector of `Mu` values and configure the Simulation Data Inspector for manual Parallel Computing Toolbox support.

```
% Enable manual Parallel Computing Toolbox support
Simulink.sdi.enablePCTSupport('manual');

% Choose several Mu values
MuVals = [1 2 3 4];
```

Initialize Parallel Workers

Use `parpool` to start a pool of four parallel workers. This example calls `parpool` inside an if statement so you only create a parallel pool if you don't already have one. You can use `spmd` to run initialization code common to all workers. For example, load the `vdp` model and select signals to log to runs that we can send to the Simulation Data Inspector on the client MATLAB. To avoid data concurrency issues when simulating with `sim` in `parfor`, create a temporary directory on each worker. After the simulations complete, another spmd block deletes the temporary directories.

```
p = gcp('nocreate');

if isempty(p)

    parpool(4);

end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd
```

```
    % Load system and select signals to log
    load_system('vdp')
    Simulink.sdi.markSignalForStreaming('vdp/x1',1,'on')
    Simulink.sdi.markSignalForStreaming('vdp/x2',1,'on')

    % Create temporary directory for simulation on worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations with parfor

To stream data from parallel workers to the Simulation Data Inspector, you have to run parallel simulations using parfor. Each worker runs a vdp simulation with a different value of Mu. Simulink cannot access the contents of the parfor loop, so the variable MuVal is defined in the worker's workspace, where the vdp model can see it, using assignin.

```
parfor (index = 1:4)

    % Set value of Mu in the worker's base workspace
    assignin('base','MuVal',MuVals(index));

    % Modify the value of Mu in the model and simulate
    set_param('vdp/Mu','Gain','MuVal')
    sim('vdp')
```

Access Data and Send Run to Client MATLAB

You can use the Simulation Data Inspector programmatic interface on the worker the same way you would in the client MATLAB. This example creates a Simulink.sdi.Run object and attaches the value of Mu used in the simulation with the Tag property.

```
    % Attach metadata to the run
    IDs = Simulink.sdi.getAllRunIDs;
    lastIndex = length(IDs);
    runID = Simulink.sdi.getRunIDByIndex(lastIndex);
    parRun = Simulink.sdi.getRun(runID);
    parRun.Tag = strcat('Mu = ',num2str(MuVals(index)));

    % Send the run to the Simulation Data Inspector on the client MATLAB
    Simulink.sdi.sendWorkerRunToClient

end
```

Close Temporary Directories and View Runs in the Simulation Data Inspector

Use another `spmd` section to delete the temporary directories created on the workers once the simulations complete. In each simulation, `Simulink.sdi.sendWorkerRunToClient` imported runs from all the workers into the Simulation Data Inspector. You can view the data and check the run properties to see the value of `Mu` used during simulation.

```
spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

end

Simulink.sdi.view
```

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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')
```

```
% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an `spmd` code block, load the `slexAircraftExample` model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use `parfor` to run the seven simulations in parallel. Select the value for `Ts` for each simulation, and modify the value of `Ts` in the model workspace. Then, run the simulation and build an array of `Simulink.sdi.WorkerRun` objects to access the data with the Simulation Data Inspector. After the `parfor` loop, use another `spmd` segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);
    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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
```

Input Arguments

`mode` — Prallel worker support mode

`'local'` (default) | `'none'` | `'all'` | `'manual'`

Mode of Simulation Data Inspector support for importing runs from parallel workers.

- `'local'` — The default behavior configures automatic import for runs generated on local workers.
- `'none'` — Disables parallel worker support. No runs from local or remote workers import to the Simulation Data Inspector.
- `'all'` — Enables automatic import for runs created from local and remote workers.
- `'manual'` — Configures support for manual import of runs created by parallel workers using the `Simulink.sdi.sendWorkerRunToClient` function.

Data Types: `char` | `string`

See Also

`Simulink.sdi.WorkerRun` | `Simulink.sdi.isPCTSupportEnabled` | `Simulink.sdi.sendWorkerRunToClient`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2017b

Simulink.sdi.exportRun

Export run data to a Simulink.SimulationData.Dataset object

Syntax

```
dataset = Simulink.sdi.exportRun(runID)
```

Description

`dataset = Simulink.sdi.exportRun(runID)` creates a `Simulink.SimulationData.Dataset` object in the base workspace with the data in the Simulation Data Inspector run identified by `runID`.

Examples

Export Run Data

This example shows how to export data from a run in the Simulation Data Inspector to a `Simulink.SimulationData.Dataset` object in the base workspace you can use to further process your data. The method you choose to export your run depends on the processing you do in your script. If you have a run object for the run, you can use the `export` method to create a `Simulink.SimulationData.Dataset` object with the run's data in the base workspace. If you do not have a run object, use the `Simulink.sdi.exportRun` function to export the run to the workspace.

Export Run Using `Simulink.sdi.exportRun`

Use the `Simulink.sdi.export` function when your workflow does not include creating a run object.

```
% Load vdp model
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 vdp model
sim('vdp')

% Get run ID for simulation run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Export run
simDataset = Simulink.sdi.exportRun(runID);
```

Export Run Using export Method

When you already have a `Simulink.sdi.Run` object for your run, you can use the `export` method to create a `Simulink.SimulationData.Dataset` object in the base workspace for further processing of the data.

```
% Load vdp model
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);
vdpRun = Simulink.sdi.getRun(runID);

% Get signal ids for signals
sigID1 = vdpRun.getSignalIDByIndex(1);
sigID2 = vdpRun.getSignalIDByIndex(2);

% Compare signals
```

```
diffResult = Simulink.sdi.compareSignals(sigID1,sigID2);
diffResult.match

ans = logical
   0
```

```
% Export run
simDataset = vdpRun.export;
```

Input Arguments

runID — Run identifier
scalar

Run identifier for the run you want to export to a `Simulink.SimulationData.Dataset`. The Simulation Data Inspector assigns a unique run ID when it creates a run. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` and `Simulink.sdi.getRunIDByIndex`.

Output Arguments

dataset — Dataset containing run data
`Simulink.SimulationData.Dataset`

`Simulink.SimulationData.Dataset` object containing the data from the run identified by `runID`.

See Also

`Simulink.SimulationData.Dataset` | `Simulink.sdi.Run` | `Simulink.sdi.getAllRunIDs` | `Simulink.sdi.getRunIDByIndex` | `Simulink.sdi.save`

Topics

"Inspect and Compare Data Programmatically"
"Save and Share Simulation Data Inspector Data and Views"

Introduced in R2017a

Simulink.sdi.getAllRunIDs

Package: Simulink.sdi

Get all Simulation Data Inspector run identifiers

Syntax

```
runIDs = Simulink.sdi.getAllRunIDs
```

Description

`runIDs = Simulink.sdi.getAllRunIDs` returns a matrix of the run identifiers for all runs in the Simulation Data Inspector repository.

Examples

Get Run ID for a Simulation

Many workflows that use the Simulation Data Inspector programmatic interface start with obtaining the ID for a simulation run. This example shows two different methods to use the programmatic interface to get the run ID for a run. You can use the run ID to create a `Simulink.sdi.Run` object to access run data and metadata, or you can use the run ID for a comparison.

Simulate a Model to Create a Run

The model `sldemo_fuelsys` is already configured for logging. When you simulate the model, the Simulation Data Inspector automatically creates a run and assigns it a run ID.

```
% Load and simulate system
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')
```

Get Run ID Using Simulink.sdi.getAllRunIDs

`Simulink.sdi.getAllRunIDs` returns an array of all run IDs for runs in the Simulation Data Inspector repository in order, with the most recently created run at the end.

```
% Get runID for most recent run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Get Run ID Using Simulink.sdi.getRunIDByIndex

You can also use `Simulink.sdi.getRunCount` and `Simulink.sdi.getRunIDByIndex` to get the run ID for a run. This method is useful if you also want to use `count` as a counting variable to index through the runs in the Simulation Data Inspector repository.

```
count = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(count);
```

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.

Setup

Load the model, and mark the `q, rad/sec` and `alpha, rad` signals for logging. Then, simulate the model to create the baseline run.

```
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Simulate system
sim('slexAircraftExample')
```

Modify Model Parameter

Modify the model parameter `Ts` in the model workspace to change the time constant of the input low-pass filter.

```
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',1)

% Simulate again
sim('slexAircraftExample')
```

Compare Runs and Inspect Results

Use the `Simulink.sdi.compareRuns` function to compare the data from the simulations. Then, inspect the `match` property of the signal result to see whether the signals fell within the default tolerance of 0.

```
% 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
sig1Result1 = diffRun1.getResultByIndex(1);
sig2Result1 = diffRun1.getResultByIndex(2);

% Check whether signals matched
sig1Result1.Match
```

```
ans = logical
   0
```

```
sig2Result1.Match
```

```
ans = logical
   0
```

Compare Runs with Signal Tolerances

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

```
% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sigID2 = run1.getSignalIDByIndex(2);

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

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = diffRun2.getResultByIndex(1);
sig2Result2 = diffRun2.getResultByIndex(2);

% Check the result
sig1Result2.Match

ans = logical
   1


sig2Result2.Match

ans = logical
   1
```

Output Arguments

runIDs — Matrix of Simulation Data Inspector run IDs
matrix

Matrix of run IDs in the Simulation Data Inspector repository.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.compareRuns` | `Simulink.sdi.copyRun` | `Simulink.sdi.copyRunViewSettings` | `Simulink.sdi.deleteRun` | `Simulink.sdi.exportRun` | `Simulink.sdi.getRun` | `Simulink.sdi.getRunCount` | `Simulink.sdi.isValidRunID`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2017a

Simulink.sdi.getArchiveRunLimit

Package: Simulink.sdi

Determine configured run limit for Simulation Data Inspector archive

Syntax

```
limit = Simulink.sdi.getArchiveRunLimit
```

Description

`limit = Simulink.sdi.getArchiveRunLimit` returns the limit configured for the number of runs the Simulation Data Inspector stores in the archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive on a first-in, first-out basis. When another run enters the archive, the Simulation Data Inspector deletes the run that has been in the archive the longest. A `limit` of `-1` indicates the archive has no limit for the number of runs it stores. A `limit` of `0` means that the archive cannot contain any runs.

Tip To retain data for only the current run, configure the Simulation Data Inspector to automatically archive runs and set the archive run limit to `0`.

Examples

Configure the Simulation Data Inspector to Retain Only the Current Run

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.

```
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;
```

Set the archive limit to `0` and configure the Simulation Data Inspector to automatically archive simulation runs.

```
setArchiveRunLimit(0)
setAutoArchiveMode(true)
```

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

```
Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)
```

Output Arguments

`limit` — Maximum number of runs to store in the Simulation Data Inspector archive

scalar

Limit for the number of runs stored in the Simulation Data Inspector archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive, on a first-in, first-out basis. A `limit` of `-1` indicates there is no limit on the runs stored in the archive. A `limit` of `0` indicates that the archive cannot contain any runs.

See Also

`Simulink.sdi.getAutoArchiveMode` | `Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode`

Topics

"Iterate Model Design Using the Simulation Data Inspector"

Introduced in R2018b

Simulink.sdi.getAutoArchiveMode

Package: Simulink.sdi

Determine if the Simulation Data Inspector is configured to automatically archive

Syntax

```
mode = Simulink.sdi.getAutoArchiveMode
```

Description

`mode = Simulink.sdi.getAutoArchiveMode` returns a logical value that indicates whether the Simulation Data Inspector is configured to automatically archive simulation runs. When `mode` is `true`, the Simulation Data Inspector automatically archives simulation runs. When `mode` is `false`, the Simulation Data Inspector does not automatically archive.

Examples

Configure the Simulation Data Inspector to Retain Only the Current Run

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.

```
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;
```

Set the archive limit to `0` and configure the Simulation Data Inspector to automatically archive simulation runs.

```
setArchiveRunLimit(0)
setAutoArchiveMode(true)
```

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

```
Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)
```

Output Arguments

`mode` — Logical indication of Simulation Data Inspector automatic archive configuration
logical

Logical value that indicates whether the Simulation Data Inspector is configured to automatically archive simulation runs.

- `true` — The Simulation Data Inspector automatically archives simulation runs.
- `false` — The Simulation Data Inspector does not automatically archive simulation runs.

See Also

`Simulink.sdi.getArchiveRunLimit` | `Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode`

Topics

"Iterate Model Design Using the Simulation Data Inspector"

Introduced in R2018b

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

Package: Simulink.sdi

Get a `Simulink.sdi.Run` object to access data

Syntax

```
run = Simulink.sdi.getRun(runID)
```

Description

`run = Simulink.sdi.getRun(runID)` returns a `Simulink.sdi.Run` object that provides access to the data in the run corresponding to the `runID`. The Simulation Data Inspector assigns run IDs when it creates a run. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

Examples

Get a Run Object for Simulation Data

Many workflows using the Simulation Data Inspector programmatic interface start with acquiring a `Simulink.sdi.Run` object for your simulation data.

```
% Load and simulate system
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')

% Get runID for most recent run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get run object
run = Simulink.sdi.getRun(runID);
```

You can use the Simulink.sdi.Run object to access signal data, add data, and inspect run metadata.

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 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
fuelRun = Simulink.sdi.getRun(runID);

% Check signal count of the run
fuelRun.signalCount

ans = int32
    15

% Get signal objects for the signals in the run
signal1 = fuelRun.getSignalByIndex(4);
signal2 = fuelRun.getSignalByIndex(9);
signal3 = fuelRun.getSignalByIndex(10);

% Create subplot 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
```

Input Arguments

`runID` — Numeric run identifier
scalar

Run ID for the run you want a `Simulink.sdi.Run` object for. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for a run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

Output Arguments

`run` — `Simulink.sdi.Run` object
'`Simulink.sdi.Run`'

`Simulink.sdi.Run` object for the run corresponding to the run ID.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.getAllRunIDs` | `Simulink.sdi.getRunIDByIndex`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2011b

Simulink.sdi.getRunCount

Package: Simulink.sdi

Get number of runs in Simulation Data Inspector repository

Syntax

```
count = Simulink.sdi.getRunCount
```

Description

`count = Simulink.sdi.getRunCount` returns the number of runs in the Simulation Data Inspector repository. You can use the run count to loop over all runs in the Simulation Data Inspector repository to modify run or signal properties. For example, you could add an absolute tolerance to a signal in every run.

Examples

Modify Parameter for Several Runs

This example shows how to modify a parameter for all the runs in the Simulation Data Inspector programmatically.

Generate Runs

Load the `vdp` model and mark the `x1` and `x2` signals for logging. Then, run several simulations.

```
% Clear all data from the Simulation Data Inspector repository
Simulink.sdi.clear

% Load the model and mark signals of interest for streaming
load_system('vdp')
Simulink.sdi.markSignalForStreaming('vdp/x1',1,'on')
Simulink.sdi.markSignalForStreaming('vdp/x2',1,'on')

% Simulate the model with several Mu values
for gain = 1:5
    gainVal = num2str(gain);
    set_param('vdp/Mu','Gain',gainVal)
    sim('vdp')
end
```

Use Simulink.sdi.getRunCount to Assign Tolerance to x1 Signals

```
count = Simulink.sdi.getRunCount;

for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
    vdpRun = Simulink.sdi.getRun(runID);
    sig = vdpRun.getSignalByIndex(1);
    sig.AbsTol = 0.1;
end

% Open the Simulation Data Inspector to view your data
Simulink.sdi.view
```

Output Arguments

count — Number of runs
scalar

Number of runs in the Simulation Data Inspector repository.

See Also

Simulink.sdi.Run | Simulink.sdi.Signal | Simulink.sdi.getAllRunIDs | Simulink.sdi.getRun | Simulink.sdi.getRunIDByIndex

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.getRunIDByIndex

Package: Simulink.sdi

Use Simulation Data Inspector run index to get run ID

Syntax

```
runID = Simulink.sdi.getRunIDByIndex(index)
```

Description

`runID = Simulink.sdi.getRunIDByIndex(index)` returns the run ID for the run with the specified `index` in the Simulation Data Inspector repository.

Examples

Get Run ID for a Simulation

Many workflows that use the Simulation Data Inspector programmatic interface start with obtaining the ID for a simulation run. This example shows two different methods to use the programmatic interface to get the run ID for a run. You can use the run ID to create a `Simulink.sdi.Run` object to access run data and metadata, or you can use the run ID for a comparison.

Simulate a Model to Create a Run

The model `sldemo_fuelsys` is already configured for logging. When you simulate the model, the Simulation Data Inspector automatically creates a run and assigns it a run ID.

```
% Load and simulate system
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')
```

Get Run ID Using `Simulink.sdi.getAllRunIDs`

`Simulink.sdi.getAllRunIDs` returns an array of all run IDs for runs in the Simulation Data Inspector repository in order, with the most recently created run at the end.

```
% Get runID for most recent run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Get Run ID Using `Simulink.sdi.getRunIDByIndex`

You can also use `Simulink.sdi.getRunCount` and `Simulink.sdi.getRunIDByIndex` to get the run ID for a run. This method is useful if you also want to use `count` as a counting variable to index through the runs in the Simulation Data Inspector repository.

```
count = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(count);
```

Input Arguments

`index` — Run index in Simulation Data Inspector
integer

Positive, whole number index of the run in the Simulation Data Inspector repository.

Example: 3

Output Arguments

`runID` — Numeric run identifier
scalar

Numeric run identification assigned by the Simulation Data Inspector.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.compareRuns` | `Simulink.sdi.copyRun` | `Simulink.sdi.deleteRun` | `Simulink.sdi.getRun` | `Simulink.sdi.getRunCount` | `Simulink.sdi.isValidRunID`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2011b

Simulink.sdi.getRunNamingRule

Package: Simulink.sdi

Get the Simulation Data Inspector rule for naming runs

Syntax

`namingRule = Simulink.sdi.getRunNamingRule`

Description

`namingRule = Simulink.sdi.getRunNamingRule` returns the run naming rule as a character vector. The run naming rule can contain one or more tokens that update for each run, for example, `<run_index>`. The run naming rule applies to runs automatically created through simulating a model in Simulink.

Examples

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);
fuelRun = Simulink.sdi.getRun(runID);
fuelRun.name

ans =
'sldemo_fuelsys Run 1'

% Clear preferences to reset the run naming rule
Simulink.sdi.clearPreferences

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'
```

Output Arguments

namingRule — Naming rule for Simulation Data Inspector runs

character vector

Character vector that specifies the naming rule the Simulation Data Inspector uses to name the runs automatically created through simulating a Simulink model. The run naming rule can contain any of the following tokens that represent information pulled for each run:

- `<run_index>` – Run's index in the Simulation Data Inspector repository.
- `<model_name>` – Name of the model simulated to create the run.
- `<time_stamp>` – Start time for the simulation that created the run.
- `<sim_mode>` – Simulation mode used for the simulation that created the run.

Alternatives

You can view the run naming rule using the Simulation Data Inspector UI. You can find the **New Run** options under the Simulation Data Inspector **Preferences** menu.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.resetRunNamingRule` | `Simulink.sdi.setRunNamingRule`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2015a

Simulink.sdi.getSignal

Package: Simulink.sdi

Get Simulink.sdi.Signal object for a signal

Syntax

```
signalObj = Simulink.sdi.getSignal(sigID)
```

Description

`signalObj = Simulink.sdi.getSignal(sigID)` returns a `Simulink.sdi.Signal` object for the signal in the Simulation Data Inspector repository that corresponds to the signal ID, `sigID`. The `Simulink.sdi.Signal` object manages signal data and metadata and allows you to view and modify signal properties.

Examples

Create a Simulation Data Inspector Run and Access Signal Data

This example shows how to access signal data when you create a run in the Simulation Data Inspector.

Generate Data for Run

For this example, create timeseries data for sine and cosine signals.

```
% Create timeseries workspace data
time = linspace(0, 20, 101);

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 Run and Return Signal IDs

You can use the `Simulink.sdi.createRun` syntax with multiple return arguments to get the signal IDs more directly instead of accessing the signal IDs through a `Simulink.sdi.Run` object.

```
[runID,runIndex,sigIDs] = Simulink.sdi.createRun('Sinusoids','vars',...
    sine_ts,cos_ts);

cosID = sigIDs(2);
cosSig = Simulink.sdi.getSignal(cosID);
```

Modify Signal Properties and View in the Simulation Data Inspector

You can use the `Simulink.sdi.Signal` object to view and modify signal properties and to plot signals in the Simulation Data Inspector.

```
cosSig.Checked = true;
cosSig.AbsTol = 0.05;
Simulink.sdi.view
cosSig.Name
```

```
ans =

    'Cosine, T=8'
```

Input Arguments

sigID — Signal ID

scalar

Signal identifier. The Simulation Data Inspector assigns signal IDs to signals when a run is created. You can get the signal ID for a signal as a return from `Simulink.sdi.createRun` or using the `Simulink.sdi.Run` object's methods.

Output Arguments

`signalObj` — Simulink.sdi.Signal object
'Simulink.sdi.Signal' object

Simulink.sdi.Signal object for the signal corresponding to `sigID`.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.Signal` | `Simulink.sdi.createRun` | `getSignalIDByIndex`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2011b

Simulink.sdi.isPCTSupportEnabled

Determine status and mode for Parallel Computing Toolbox support

Syntax

```
[enabled,mode] = Simulink.sdi.isPCTSupportEnabled
```

Description

`[enabled,mode] = Simulink.sdi.isPCTSupportEnabled` returns `enabled`, a logical indication of whether support for the Parallel Computing Toolbox is enabled, and `mode`, the mode of support enabled.

Examples

Check Status of Parallel Worker Support

Before running code that depends on whether automatic import of runs created by parallel workers is enabled, you can use the `Simulink.sdi.isPCTSupportEnabled` function to check the support status. The default behavior for the Simulation Data Inspector enables parallel worker support in `local` mode. In `local` mode, only runs created on local workers automatically import into the Simulation Data Inspector.

```
[enabled, mode] = Simulink.sdi.isPCTSupportEnabled
```

```
enabled =

  logical

   1


mode =
'local'
```

Output Arguments

`enabled` — Logical indicator of parallel worker support
logical

Logical indication of parallel worker support.

- `1` indicates that support for parallel workers is enabled.
- `0` indicates that support for parallel workers is not enabled.

`mode` — Parallel worker support mode
local (default)

Mode of Parallel Computing Toolbox support.

- `'local'` — Runs generated on local workers automatically import to the Simulation Data Inspector.
- `'none'` — Parallel worker support is disabled.
- `'all'` — Runs created from local and remote workers automatically import to the Simulation Data Inspector.
- `'manual'` — Support for manual import of runs created by parallel workers using the `Simulink.sdi.sendWorkerRunToClient` function.

See Also

`Simulink.sdi.WorkerRun` | `Simulink.sdi.enablePCTSupport` | `Simulink.sdi.sendWorkerRunToClient`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2017b

Simulink.sdi.isValidRunID

Package: Simulink.sdi

Determine whether a run ID is valid

Syntax

```
valid = Simulink.sdi.isValidRunID(runID)
```

Description

`valid = Simulink.sdi.isValidRunID(runID)` returns `true` if `runID` corresponds to a run in the Simulation Data Inspector repository.

Examples

Check Run ID Validity

This example shows how to check whether a run ID is valid. You can use `Simulink.sdi.isValidRunID` to ensure you have valid data throughout your script.

Create a Simulation Run

Simulate the model `sldemo_fuelsys` to create a run in the Simulation Data Inspector, and use `Simulink.sdi.getAllRunIDs` to get its run ID.

```
% Simulate model
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')

% Get run ID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Check Run ID Validity

Check to verify that the Simulation Data Inspector has a run corresponding to the run ID.

```
Simulink.sdi.isValidRunID(runID)
```

```
ans = logical
   1
```

Delete the Run and Check Validity

You can delete runs to clear out memory space or clean up the Simulation Data Inspector UI. When you delete a run, the run ID for that run becomes invalid.

```
Simulink.sdi.deleteRun(runID)
```

```
Simulink.sdi.isValidRunID(runID)
```

```
ans = logical
   0
```

Input Arguments

`runID` — Simulation Data Inspector run identifier
scalar

Unique numeric identification for the run. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

Output Arguments

`valid` — Run validity indicator
logical

Run validity indicator. When `valid` is `true`, the `runID` is valid. When `valid` is `false`, the `runID` is invalid.

See Also

`Simulink.sdi.compareRuns` | `Simulink.sdi.createRun` | `Simulink.sdi.deleteRun` | `Simulink.sdi.getAllRunIDs` | `Simulink.sdi.getRunIDByIndex`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.load

Package: Simulink.sdi

Load a Simulation Data Inspector session or view

Syntax

```
valid = Simulink.sdi.load(fileName)
```

Description

`valid = Simulink.sdi.load(fileName)` loads the data in the MLDATX-file or MAT-file specified by `fileName` and returns `1` when `fileName` is a valid file. A return value of `0` indicates that the file specified by `fileName` is invalid and cannot be loaded into the Simulation Data Inspector. You can use `Simulink.sdi.load` to load Simulation Data Inspector sessions and views. A view includes preferences and visualization options but does not save data. A session saves data along with preference selections and plot configurations.

Examples

Save a Simulation Data Inspector Session

This example shows how to create, save, and load a Simulation Data Inspector session. The example uses data logging to populate the Simulation Data Inspector with data and then uses the Simulation Data Inspector's programmatic interface to create plots to visualize the data. After saving the data and visualization settings in a session, the Simulation Data Inspector repository is emptied in order to demonstrate how to load the session.

Create Simulation Data

This example logs the `Stick`, `alpha, rad`, and `q, rad/sec` signals to generate simulation data using the model `slexAircraftExample` and creates two runs. The first uses a sine input, and the second has a square wave input.

```
% Ensure you start with an empty Simulation Data Inspector repository
Simulink.sdi.clear

% Load system
load_system('slexAircraftExample')

% Configure signals to log
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Change Pilot signal to sine
set_param('slexAircraftExample/Pilot','WaveForm','sine')

% Simulate model
sim('slexAircraftExample')

% Change Pilot signal to square
set_param('slexAircraftExample/Pilot','WaveForm','square')

% Simulate model
sim('slexAircraftExample')
```

Access Simulation Data

Use the Simulation Data Inspector programmatic interface to access the simulation data so you can create plots to visualize the signals.

```
% Get run objects
runIDs = Simulink.sdi.getAllRunIDs;
sineRunID = runIDs(end-1);
squareRunID = runIDs(end);

sineRun = Simulink.sdi.getRun(sineRunID);
squareRun = Simulink.sdi.getRun(squareRunID);

% Get signal objects
sineOut = sineRun.getSignalByIndex(1);
sineIn = sineRun.getSignalByIndex(3);
```

```
squareOut = squareRun.getSignalByIndex(1);
squareIn = squareRun.getSignalByIndex(3);
```

Create Plots in the Simulation Data Inspector

Use the programmatic interface to visualize the signal data from the two simulation runs. You can set the plot layout and plot signals on specific subplots.

```
% Set subplot layout
Simulink.sdi.setSubPlotLayout(2,1)

% Plot sine data on top plot
sineIn.plotOnSubPlot(1,1,true)
sineOut.plotOnSubPlot(1,1,true)

% Plot square wave data on bottom plot
squareIn.plotOnSubPlot(2,1,true)
squareOut.plotOnSubPlot(2,1,true)
```

Save a Simulation Data Inspector Session

First, view the plots you just created. Then, save the Simulation Data Inspector session as an MLDATX-file to recover your data along with your preference selections and plots.

```
% View the visualized data in the Simulation Data Inspector
Simulink.sdi.view

% Save the Simulation Data Inspector session
Simulink.sdi.save('myData.mldatx')
```

Load a Simulation Data Inspector Session

First, clear the Simulation Data Inspector repository with `Simulink.sdi.clear` and reset visualization settings with `Simulink.sdi.clearPreferences`. Then, you can load the session to see how the data and settings are preserved.

```
% Clear Simulation Data Inspector repository and preferences
Simulink.sdi.clear
Simulink.sdi.clearPreferences
```

```
% Load session file to view data
Simulink.sdi.load('myData.mldatx');
```

Input Arguments

fileName — Name of file to load
character vector

Name of the file to load with the session or view data.

Example: 'myData.mldatx'

Example: 'myData.mat'

Output Arguments

valid — File validity indicator
logical

Validity indicator for the file. When the file specified by fileName is valid, valid is 1. A valid value of 0 indicates an invalid file.

See Also

Simulink.sdi.close | Simulink.sdi.createRun | Simulink.sdi.save

Topics

"Inspect and Compare Data Programmatically"
"View Data with the Simulation Data Inspector"

Introduced in R2011b

Simulink.sdi.markSignalForStreaming

Package: Simulink.sdi

Turn logging on or off for a signal

Syntax

```
Simulink.sdi.markSignalForStreaming(block,portIndex,log)
Simulink.sdi.markSignalForStreaming(portHandle,log)
Simulink.sdi.markSignalForStreaming(lineHandle,log)
```

Description

`Simulink.sdi.markSignalForStreaming(block,portIndex,log)` marks the signal on the specified `portIndex` of the specified `block` for logging when you specify `log` as `'on'`. To stop logging a signal, specify `log` as `'off'`.

`Simulink.sdi.markSignalForStreaming(portHandle,log)` marks the signal on the port specified by `portHandle` for logging when you specify `log` as `'on'`. To stop logging a signal, specify `log` as `'off'`.

`Simulink.sdi.markSignalForStreaming(lineHandle,log)` marks the signal with the specified `lineHandle` for logging when you specify `log` as `'on'`. To stop logging a signal, specify `log` as `'off'`.

Examples

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of the input and output signals for a control system. 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
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = aircraftRun.getSignalIDByIndex(1);
signalID2 = aircraftRun.getSignalIDByIndex(2);

if (aircraftRun.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (aircraftRun.isValidSignalID(signalID1) && aircraftRun.isValidSignalID(signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signals match within tolerance
    match = sigDiff.match
end
```

```
match = logical
   0
```

Mark Signals for Logging with Port Handles

This example shows how to mark signals for logging using port handles.

Load Model and Mark Signals for Streaming

User get_param to get the port handles for the blocks with your signals of interest. Then, use the handle to mark the desired signals for logging.

```
load_system('vdp')

% Get port handles
x1_handles = get_param('vdp/x1','PortHandles');
x1 = x1_handles.Outport(1);
x2_handles = get_param('vdp/x2','PortHandles');
x2 = x2_handles.Outport(1);

% Mark signals for streaming
Simulink.sdi.markSignalForStreaming(x1,'on');
Simulink.sdi.markSignalForStreaming(x2,'on');
```

Simulate Model and View Signals in the Simulation Data Inspector

Simulate the model and then open the Simulation Data Inspector to view the logged signals.

```
sim('vdp')

Simulink.sdi.view
```

Mark Signals for Logging with Line Handles

This example shows how to mark signals for logging using their line handles.

Load System and Mark Signals for Logging

Load a model and use get_param to get handles for the signals in the model. Then, use the line handles to mark signals of interest for logging.

```
load_system('slexAircraftExample')

lines = get_param('slexAircraftExample','Lines');

sig1handle = lines(1).Handle;
sig2handle = lines(2).Handle;
```

```
Simulink.sdi.markSignalForStreaming(sig1handle,'on')
Simulink.sdi.markSignalForStreaming(sig2handle,'on')
```

Simulate Model and View Signals

Simulate the model and view the signals marked for logging in the Simulation Data Inspector.

```
sim('slexAircraftExample')
```

```
Simulink.sdi.view
```

Input Arguments

block — Source block path or handle

character vector

Block path for the block with the desired signal connected to one of its outports.

Example: `'slexAircraftExample/Pilot'`

portIndex — Source block output port index

integer

Index of the port connected to the signal you want to mark for streaming.

Example: `'slexAircraftExample/Pilot'`

log — Logging state

'on' | 'off'

Logging state desired for signal.

- `'on'` -- Turn logging on for a signal.
- `'off'` -- Turn logging off for a signal.

portHandle — Source block output port handle

handle

Port handle for the source block's output port that connects to the signal.

Example: `x1_handles.Outport(1)`

`lineHandle` — Signal line handle
handle

Line handle for the signal.

Example: `lines(1).Handle`

See Also

`Simulink.HMI.InstrumentedSignals` | `Simulink.sdi.createRunOrAddToStreamedRun` | `Simulink.sdi.getAllRunIDs` | `Simulink.sdi.getRunIDByIndex`

Topics

"Inspect and Compare Data Programmatically"
"View Data with the Simulation Data Inspector"

Introduced in R2015b

Simulink.sdi.report

Package: Simulink.sdi

Generate a Simulation Data Inspector report

Syntax

```
Simulink.sdi.report
Simulink.sdi.report(Name,Value)
```

Description

`Simulink.sdi.report` creates a Simulation Data Inspector report of the plotted data in the **Inspect** pane of 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 in the Simulation Data Inspector.

Examples

Generate a Simulation Data Inspector Report Programmatically

This example shows how to create reports using the Simulation Data Inspector programmatic interface. You can create a report for plotted signals in the Inspect pane or for comparison data in the Compare pane. This example first generates data by simulating a model, then shows how to create an `Inspect Signals` report. To run the example exactly as shown, ensure that the Simulation Data Inspector repository starts empty with the `Simulink.sdi.clear` function.

Generate Data

This example generates data using model `ex_sldemo_absbrake` with two different desired slip ratios.

```
% Ensure Simulation Data Inspector is empty
Simulink.sdi.clear

% Open model
load_system('ex_sldemo_absbrake')

% Set slip ratio and simulate model
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.24')
sim('ex_sldemo_absbrake')

% Set new slip ratio and simulate model again
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.25')
sim('ex_sldemo_absbrake')
```

Plot Signals in the Inspect Pane

The `Inspect Signals` report includes all signals plotted in the graphical viewing area of the Inspect pane and all displayed metadata for the plotted signals.

```
% Get Simulink.sdi.Run objects
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end-1);
runID2 = runIDs(end);

run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

% Get Simulink.sdi.Signal objects for slp signal
run1_slp = run1.getSignalByIndex(4);
run2_slp = run2.getSignalByIndex(4);

% Plot slp signals
run1_slp.plotOnSubPlot(1, 1, true)
run2_slp.plotOnSubPlot(1, 1, true)
```

Create a Report of Signals Plotted in Inspect Pane

You can include more data in the report by adding more columns using the Simulation Data Inspector UI, or you can specify the information you want in the report programmatically with Name-Value pairs and the enumeration class `Simulink.sdi.SignalMetaData`. This example shows how to specify the data in the report programmatically.

```
% Specify report parameters
reportType = 'Inspect Signals';
```

```
reportName = 'Data_Report.html';

signalMetadata = [Simulink.sdi.SignalMetaData.Run, ...
    Simulink.sdi.SignalMetaData.Line, ...
    Simulink.sdi.SignalMetaData.BlockName, ...
    Simulink.sdi.SignalMetaData.SignalName];

Simulink.sdi.report('ReportToCreate', reportType, 'ReportOutputFile', ...
    reportName, 'ColumnsToReport', signalMetadata);
```

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ReportToCreate','Compare Runs','ShortenBlockPath',true`

`ReportToCreate` — Information to include in the report

`'Inspect Signals'` (default) | `'Compare Runs'`

Simulation Data Inspector pane to capture in the report.

- `'Inspect Signals'` -- Create a report of the information on the **Inspect** pane.
- `'Compare Runs'` -- Create a report of the information on the **Compare** pane.

Example: `'ReportToCreate','Inspect Signals'`

Example: `'ReportToCreate','Compare Runs'`

`ReportOutputFolder` — Folder where report is saved

`<current working folder>/slprj/sdi` (default) | `path`

Folder where the report is saved. Specify the path to the folder where you want to save the report as a character vector.

Example: `'ReportOutputFolder','C:\Users\user1\Desktop'`

ReportOutputFile — Report file name

'SDI_report.html' (default) | character vector

File name for report.

Example: 'ReportOutputFile','MyReport.html'

PreventOverwritingFile — Report overwrite prevention

true (default) | false

File overwrite protection for the report. File overwrite protection prevents the Simulation Data Inspector from overwriting an existing file by appending the file name with a number that increments for subsequent reports. When you disable file overwrite protection, the Simulation Data Inspector overwrites the existing report file unless you specify a unique file name.

- true enables file overwrite protection.
- false disables file overwrite protection.

Example: 'PreventOverwritingFile',true

Example: 'PreventOverwritingFile',false

ColumnsToReport — Signal metadata to include in report

array

Signal metadata to include in report. By default, the Inspect Signals report includes the block path, name, line style and color, and data source parameters for each plotted signal. The Compare Runs report includes the signal name, absolute tolerance, relative tolerance, and maximum difference metadata by default.

Specify metadata to include as an array, using the enumeration class Simulink.sdi.SignalMetaData. For example, to include the name of the simulation run and signal name, create an array like signal_metadata:

```
signal_metadata = [Simulink.sdi.SignalMetaData.Run,...
                   Simulink.sdi.SignalMetaData.SignalName];
```

Then, specify ColumnsToReport as signal_metadata in the name-value pair:

```
Simulink.sdi.report('ColumnsToReport',signal_metadata)
```

The table summarizes the metadata available for Inspect Signals report.

Column Value	Description
SignalName (default)	Signal name
Line (default)	Signal line style and color
SID	Simulink identifier For more information about SIDs, see “Locate Diagram Components Using Simulink Identifiers”
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
AbsTol	Absolute tolerance for the signal
RelTol	Relative tolerance for the signal
OverrideGlobalTol	Property that specifies whether signal tolerances take priority over global tolerances
TimeTol	Time tolerance for the signal
InterpMethod	Interpolation method
SyncMethod	Synchronization method used to coordinate signals for comparison

Column Value	Description
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

The table provides a summary of the metadata 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)	Absolute tolerance for the **Baseline** signal
RelTol (default)	Relative tolerance for the **Baseline** signal
MaxDifference	The maximum difference between the **Baseline** and **Compare To** signals
OverrideGlobalTol	Property that specifies whether the **Baseline** signal tolerances take priority over global tolerances
TimeTol	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

Column Value	Description
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
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

Column Value	Description
`Channel1`	Index of the **Baseline** within its matrix
`Channel2`	Index of the **Compare To** within its matrix
`InterpMethod`	Interpolation method for the **Baseline** signal
`SyncMethod`	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
`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

Example: `'ColumnsToReport',metadata`

`ShortenBlockPath` — Block path length

`true` (default) | `false`

Block path length.

- `true` -– Report shortened block path length.
- `false` -– Include the full block path in the report.

Example: `'ShortenBlockPath',false`

`LaunchReport` — Open report when created

`true` (default) | `false`

Open report when created.

- `true` –– Open the report when it is created.
- `false` –– Do not open the report upon creation.

Example: `'LaunchReport',false`

`SignalsToReport` — Signals to include in a Compare Runs report

`'ReportOnlyMismatchedSignals'` (default) | `'ReportAllSignals'`

Signals to include in a `Compare Runs` report.

- `ReportOnlyMismatchedSignals` –– Include only signals that fall out of tolerance.
- `ReportAllSignals` –– Include all signals.

Example: `'SignalsToReport','ReportAllSignals'`

See Also

`Simulink.sdi.Signal` | `Simulink.sdi.compareRuns` |
`Simulink.sdi.compareSignals` | `Simulink.sdi.createRun`

Topics

"Inspect and Compare Data Programmatically"
"Create Plots Using the Simulation Data Inspector"
"Save and Share Simulation Data Inspector Data and Views"

Introduced in R2011b

Simulink.sdi.resetRunNamingRule

Package: Simulink.sdi

Revert the Simulation Data Inspector run naming rule to default

Syntax

```
Simulink.sdi.resetRunNamingRule
```

Description

`Simulink.sdi.resetRunNamingRule` resets the run naming rule the Simulation Data Inspector uses to assign a name to runs created through simulating a Simulink model to its default `'Run <run_index>: <model_name>'`.

Examples

Modify then Reset Run Naming Rule

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, reset the run naming rule to its default, 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 1'

% Reset the run naming rule
Simulink.sdi.resetRunNamingRule

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'
```

Alternatives

You can reset the run naming rule to its default using the Simulation Data Inspector UI. Use the **Restore Defaults** button on the **New Run** tab under Simulation Data Inspector **Preferences**.

`Simulink.sdi.clearPreferences` restores the run naming rule, along with all other Simulation Data Inspector preferences.

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 R2015a

Simulink.sdi.save

Package: Simulink.sdi

Save Simulation Data Inspector session

Syntax

```
Simulink.sdi.save(fileName)
```

Description

`Simulink.sdi.save(fileName)` saves all runs, signals, and plot settings as a Simulation Data Inspector session in the file `fileName`.

Examples

Save a Simulation Data Inspector Session

This example shows how to create, save, and load a Simulation Data Inspector session. The example uses data logging to populate the Simulation Data Inspector with data and then uses the Simulation Data Inspector's programmatic interface to create plots to visualize the data. After saving the data and visualization settings in a session, the Simulation Data Inspector repository is emptied in order to demonstrate how to load the session.

Create Simulation Data

This example logs the `Stick`, `alpha, rad`, and `q, rad/sec` signals to generate simulation data using the model `slexAircraftExample` and creates two runs. The first uses a sine input, and the second has a square wave input.

```
% Ensure you start with an empty Simulation Data Inspector repository
Simulink.sdi.clear
```

```
% Load system
load_system('slexAircraftExample')

% Configure signals to log
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Change Pilot signal to sine
set_param('slexAircraftExample/Pilot','WaveForm','sine')

% Simulate model
sim('slexAircraftExample')

% Change Pilot signal to square
set_param('slexAircraftExample/Pilot','WaveForm','square')

% Simulate model
sim('slexAircraftExample')
```

Access Simulation Data

Use the Simulation Data Inspector programmatic interface to access the simulation data so you can create plots to visualize the signals.

```
% Get run objects
runIDs = Simulink.sdi.getAllRunIDs;
sineRunID = runIDs(end-1);
squareRunID = runIDs(end);

sineRun = Simulink.sdi.getRun(sineRunID);
squareRun = Simulink.sdi.getRun(squareRunID);

% Get signal objects
sineOut = sineRun.getSignalByIndex(1);
sineIn = sineRun.getSignalByIndex(3);

squareOut = squareRun.getSignalByIndex(1);
squareIn = squareRun.getSignalByIndex(3);
```

Create Plots in the Simulation Data Inspector

Use the programmatic interface to visualize the signal data from the two simulation runs. You can set the plot layout and plot signals on specific subplots.

```
% Set subplot layout
Simulink.sdi.setSubPlotLayout(2,1)

% Plot sine data on top plot
sineIn.plotOnSubPlot(1,1,true)
sineOut.plotOnSubPlot(1,1,true)

% Plot square wave data on bottom plot
squareIn.plotOnSubPlot(2,1,true)
squareOut.plotOnSubPlot(2,1,true)
```

Save a Simulation Data Inspector Session

First, view the plots you just created. Then, save the Simulation Data Inspector session as an MLDATX-file to recover your data along with your preference selections and plots.

```
% View the visualized data in the Simulation Data Inspector
Simulink.sdi.view

% Save the Simulation Data Inspector session
Simulink.sdi.save('myData.mldatx')
```

Load a Simulation Data Inspector Session

First, clear the Simulation Data Inspector repository with `Simulink.sdi.clear` and reset visualization settings with `Simulink.sdi.clearPreferences`. Then, you can load the session to see how the data and settings are preserved.

```
% Clear Simulation Data Inspector repository and preferences
Simulink.sdi.clear
Simulink.sdi.clearPreferences

% Load session file to view data
Simulink.sdi.load('myData.mldatx');
```

Input Arguments

`fileName` — File name

character vector

Name for the session file.

Example: `'myData.mldatx'`

See Also

`Simulink.sdi.close` | `Simulink.sdi.load`

Topics

“Inspect and Compare Data Programmatically”
“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2011b

Simulink.sdi.sendWorkerRunToClient

Package: Simulink.sdi

Send run created on parallel workers to the Simulation Data Inspector

Syntax

```
Simulink.sdi.sendWorkerRunToClient
Simulink.sdi.sendWorkerRunToClient(run)
```

Description

`Simulink.sdi.sendWorkerRunToClient` sends the run most recently generated by the worker to the client MATLAB and imports the run to the Simulation Data Inspector.

`Simulink.sdi.sendWorkerRunToClient(run)` sends the run corresponding to `run` to the client MATLAB and imports the run to the Simulation Data Inspector.

Examples

Manually Send Runs from Parallel Workers to the Simulation Data Inspector

This example shows how to use `Simulink.sdi.sendWorkerRunToClient` to send runs created using parallel workers manually to the Simulation Data Inspector.

Setup

This example runs several simulations of the `vdp` model, varying the value of the gain, `Mu`. To set up for the parallel simulation, define a vector of `Mu` values and configure the Simulation Data Inspector for manual Parallel Computing Toolbox support.

```
% Enable manual Parallel Computing Toolbox support
Simulink.sdi.enablePCTSupport('manual');
```

```
% Choose several Mu values
MuVals = [1 2 3 4];
```

Initialize Parallel Workers

Use `parpool` to start a pool of four parallel workers. This example calls `parpool` inside an if statement so you only create a parallel pool if you don't already have one. You can use `spmd` to run initialization code common to all workers. For example, load the `vdp` model and select signals to log to runs that we can send to the Simulation Data Inspector on the client MATLAB. To avoid data concurrency issues when simulating with `sim` in `parfor`, create a temporary directory on each worker. After the simulations complete, another spmd block deletes the temporary directories.

```
p = gcp('nocreate');

if isempty(p)

    parpool(4);

end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('vdp')
    Simulink.sdi.markSignalForStreaming('vdp/x1',1,'on')
    Simulink.sdi.markSignalForStreaming('vdp/x2',1,'on')

    % Create temporary directory for simulation on worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations with `parfor`

To stream data from parallel workers to the Simulation Data Inspector, you have to run parallel simulations using `parfor`. Each worker runs a `vdp` simulation with a different value of Mu. Simulink cannot access the contents of the parfor loop, so the variable MuVal is defined in the worker's workspace, where the vdp model can see it, using assignin.

```
parfor (index = 1:4)

    % Set value of Mu in the worker's base workspace
    assignin('base','MuVal',MuVals(index));

    % Modify the value of Mu in the model and simulate
    set_param('vdp/Mu','Gain','MuVal')
    sim('vdp')
```

Access Data and Send Run to Client MATLAB

You can use the Simulation Data Inspector programmatic interface on the worker the same way you would in the client MATLAB. This example creates a Simulink.sdi.Run object and attaches the value of Mu used in the simulation with the Tag property.

```
    % Attach metadata to the run
    IDs = Simulink.sdi.getAllRunIDs;
    lastIndex = length(IDs);
    runID = Simulink.sdi.getRunIDByIndex(lastIndex);
    parRun = Simulink.sdi.getRun(runID);
    parRun.Tag = strcat('Mu = ',num2str(MuVals(index)));

    % Send the run to the Simulation Data Inspector on the client MATLAB
    Simulink.sdi.sendWorkerRunToClient

end
```

Close Temporary Directories and View Runs in the Simulation Data Inspector

Use another spmd section to delete the temporary directories created on the workers once the simulations complete. In each simulation, Simulink.sdi.sendWorkerRunToClient imported runs from all the workers into the Simulation Data Inspector. You can view the data and check the run properties to see the value of Mu used during simulation.

```
spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

end

Simulink.sdi.view
```

Input Arguments

run — Run ID or `Simulink.sdi.Run` object

runID | `Simulink.sdi.Run` object

Run ID or `Simulink.sdi.Run` object corresponding to the run you want to import into the Simulation Data Inspector.

See Also

`Simulink.sdi.WorkerRun` | `Simulink.sdi.cleanupWorkerResources` | `Simulink.sdi.enablePCTSupport` | `Simulink.sdi.isPCTSupportEnabled`

Topics

"Inspect and Compare Data Programmatically"

Introduced in R2018a

Simulink.sdi.setArchiveRunLimit

Package: Simulink.sdi

Specify a limit for the number of runs stored in the Simulation Data Inspector archive

Syntax

```
Simulink.sdi.setArchiveRunLimit(limit)
```

Description

`Simulink.sdi.setArchiveRunLimit(limit)` sets the Simulation Data Inspector archive limit as specified by `limit`. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive on a first-in, first-out basis. When another run enters the archive, the Simulation Data Inspector deletes the run that has been in the archive the longest. A `limit` of `-1` indicates that the archive has no limit for the number of runs it stores. A `limit` of `0` means that the archive cannot contain any runs.

Tip To retain data for only the current run, configure the Simulation Data Inspector to automatically archive runs and set the archive run limit to `0`.

Examples

Configure the Simulation Data Inspector to Retain Only the Current Run

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.

```
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;
```

Set the archive limit to `0` and configure the Simulation Data Inspector to automatically archive simulation runs.

```
setArchiveRunLimit(0)
setAutoArchiveMode(true)
```

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

```
Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)
```

Input Arguments

`limit` — Maximum number of runs to store in the Simulation Data Inspector archive

scalar

Limit for the number of runs stored in the Simulation Data Inspector archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive, on a first-in, first-out basis. A `limit` of `-1` indicates that there is no limit on the runs stored in the archive. A `limit` of `0` indicates that the archive cannot contain any runs.

Example: `-1`

Example: `10`

See Also

`Simulink.sdi.getArchiveRunLimit` | `Simulink.sdi.getAutoArchiveMode` | `Simulink.sdi.setAutoArchiveMode`

Topics

“Iterate Model Design Using the Simulation Data Inspector”

Introduced in R2018b

Simulink.sdi.setAutoArchiveMode

Package: Simulink.sdi

Specify whether the Simulation Data Inspector automatically archives simulation runs

Syntax

```
Simulink.sdi.setAutoArchiveMode(mode)
```

Description

`Simulink.sdi.setAutoArchiveMode(mode)` configures the automatic archive behavior of the Simulation Data Inspector, according to `mode`. The Simulation Data Inspector automatically archives simulation runs when you specify `mode` as `true`. When you specify `mode` as `false`, the Simulation Data Inspector does not automatically archive.

Examples

Configure the Simulation Data Inspector to Retain Only the Current Run

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.

```
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;
```

Set the archive limit to `0` and configure the Simulation Data Inspector to automatically archive simulation runs.

```
setArchiveRunLimit(0)
setAutoArchiveMode(true)
```

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

```
Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)
```

Input Arguments

`mode` — Logical specification of Simulation Data Inspector automatic archive behavior

`true` (default) | `false`

Logical value that specifies whether the Simulation Data Inspector automatically archives simulation runs.

- `true` — The Simulation Data Inspector automatically archives simulation runs.
- `false` — The Simulation Data Inspector does not automatically archive simulation runs.

Example: `false`

Data Types: `logical`

See Also

`Simulink.sdi.getArchiveRunLimit` | `Simulink.sdi.getAutoArchiveMode` | `Simulink.sdi.setArchiveRunLimit`

Topics

"Iterate Model Design Using the Simulation Data Inspector"

Introduced in R2018b

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

Package: Simulink.sdi

Specify the Simulation Data Inspector run naming rule

Syntax

`Simulink.sdi.setRunNamingRule('rule')`

Description

`Simulink.sdi.setRunNamingRule('rule')` sets the Simulation Data Inspector rule for naming runs created by simulating a Simulink model.

Examples

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);
```

```
fuelRun = Simulink.sdi.getRun(runID);
fuelRun.name

ans =
'sldemo_fuelsys Run 1'

% Clear preferences to reset the run naming rule
Simulink.sdi.clearPreferences

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'
```

Input Arguments

'rule' — Simulation Data Inspector run naming rule

'Run <run_index>: <model_name> (default) | character vector

Simulation Data Inspector run naming rule for runs created by simulating a Simulink model. The character vector specifying the run naming rule can include plain text and any of the following tokens that represent data pulled from each run:

- `<run_index>` – Run's index in the Simulation Data Inspector repository.
- `<model_name>` – Name of the model simulated to create the run.
- `<time_stamp>` – Start time for the simulation that created the run.
- `<sim_mode>` – Simulation mode used for the simulation that created the run.

Example: `'<time_stamp> Simulation <run_index>: <model_name>'`

Example: `'<model_name> - <run_index>'`

Alternatives

You can modify the run naming rule using the Simulation Data Inspector UI in the **Preferences** menu. You can rename a run by modifying the `Name` property of its `Simulink.sdi.Run` object.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.getRunNamingRule` | `Simulink.sdi.resetRunNamingRule`

Topics

"Inspect and Compare Data Programmatically"
"Organize Your Simulation Data Inspector Workspace"

Introduced in R2011b

Simulink.sdi.setRunOverwrite

Package: Simulink.sdi

(Removed) Enable and disable run overwrite mode for a Simulation Data Inspector run

Note The `Simulink.sdi.setRunOverwrite` function has been removed. You can configure the Overwrite Run mode behavior using the `Simulink.sdi.setAutoArchiveMode` and `Simulink.sdi.setArchiveRunLimit` functions. For further details, see “Compatibility Considerations”.

Syntax

```
Simulink.sdi.setRunOverwrite(runID,overwrite)
```

Description

`Simulink.sdi.setRunOverwrite(runID,overwrite)` configures run overwrite mode for the run corresponding to `runID`, according to the value specified for `overwrite`.

Examples

Enable and Disable Run Overwrite for a Run

This example shows how to enable and disable run overwrite mode for a run in the Simulation Data Inspector. Run overwrite mode can help you avoid creating a large amount of intermediate data in an iterative design workflow.

Load the `sldemo_fuelsys` model, and then run a simulation to create a run in the Simulation Data Inspector. Use the Simulation Data Inspector programmatic interface to access the run ID for the run.

```
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')
```

```
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Enable run overwrite for the run you created to start your iterative design process. When run overwrite is enabled for a run, the Simulation Data Inspector replaces that run's data with new simulation data for every subsequent simulation until you disable run overwrite mode for that run.

```
Simulink.sdi.setRunOverwrite(runID,true)
```

When you get to a stopping point or a meaningful intermediate stage and you want to retain the data for a run, you can disable run overwrite mode for the run.

```
Simulink.sdi.setRunOverwrite(runID,false)
```

Input Arguments

`runID` — Numeric run identifier

scalar

Run ID for the run you want to set run overwrite for. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for a run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

`overwrite` — Run overwrite enable/disable

true | false

Logical value for run overwrite mode. Specify `overwrite` as `true` to enable run overwrite mode for a run, and set `overwrite` to `false` to disable run overwrite mode for a run.

Compatibility Considerations

`Simulink.sdi.setRunOverwrite` function has been removed

Errors starting in R2018b

The `Simulink.sdi.setRunOverwrite` function is removed in R2018b with the introduction of the Simulation Data Inspector archive. You can configure the Overwrite Run mode behavior using the new `Simulink.sdi.setAutoArchiveMode` and

`Simulink.sdi.setArchiveRunLimit` functions. Enable the Simulation Data Inspector to automatically archive simulation runs, and set the archive limit to `0`.

```
Simulink.sdi.setAutoArchiveMode(true);
Simulink.sdi.setArchiveRunLimit(0);
```

With these settings, the Simulation Data Inspector retains the data for the current simulation run and discards the data for the previous simulation run. The view in the Simulation Data Inspector also updates to show signals in the current run, with consistent line styles and colors.

You can disable each aspect of the behavior separately. To retain prior run data, set the archive limit to a nonzero value. If you want no limit on the number of runs stored in the archive, you can specify `-1` as the archive limit. When you disable the **Automatically archive** setting, the Simulation Data Inspector does not move the prior run to the archive or update the view to show signals from the current run.

See Also

`Simulink.sdi.getArchiveRunLimit` | `Simulink.sdi.getAutoArchiveMode` | `Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode`

Topics

“Iterate Model Design Using the Simulation Data Inspector”

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
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.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 = sinusoidsRun.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 = sinusoidsRun.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')
```

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');
```

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.snapshot

Package: Simulink.sdi

Capture contents of Simulation Data Inspector plots

Syntax

```
fig = Simulink.sdi.snapshot
[fig,image] = Simulink.sdi.snapshot
Simulink.sdi.snapshot(Name,Value)
Simulink.sdi.snapshot(Name,Value)
Simulink.sdi.snapshot(Name,Value)
```

Description

`fig = Simulink.sdi.snapshot` creates a figure of the plotting area in the open Simulation Data Inspector session with the figure handle `fig`.

`[fig,image] = Simulink.sdi.snapshot` creates a figure of the plotting area in the open Simulation Data Inspector session with the figure handle `fig` and returns the image data in the array, `image`.

`Simulink.sdi.snapshot(Name,Value)` captures an image of the Simulation Data Inspector plots according to the options specified by name-value pairs.

`fig` = `Simulink.sdi.snapshot(Name,Value)` captures an image of the Simulation Data Inspector plots according to the options specified by name-value pairs. This syntax returns the figure handle, `fig`, if a figure is created.

[`fig`, `image`] = `Simulink.sdi.snapshot(Name,Value)` captures an image of the Simulation Data Inspector plots according to the options specified by name-value pairs. This syntax returns the figure handle, `fig`, and an array of image data, `image`, when appropriate for the specified options.

Examples

Copy View Settings to a Run

This example shows how to copy the view settings for aligned signals from one run to another.

Simulate Your Model and get Run Object

Simulate the `vdp` model to create a run of data to visualize.

```
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
sim('vdp')

runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify View Settings for Signals

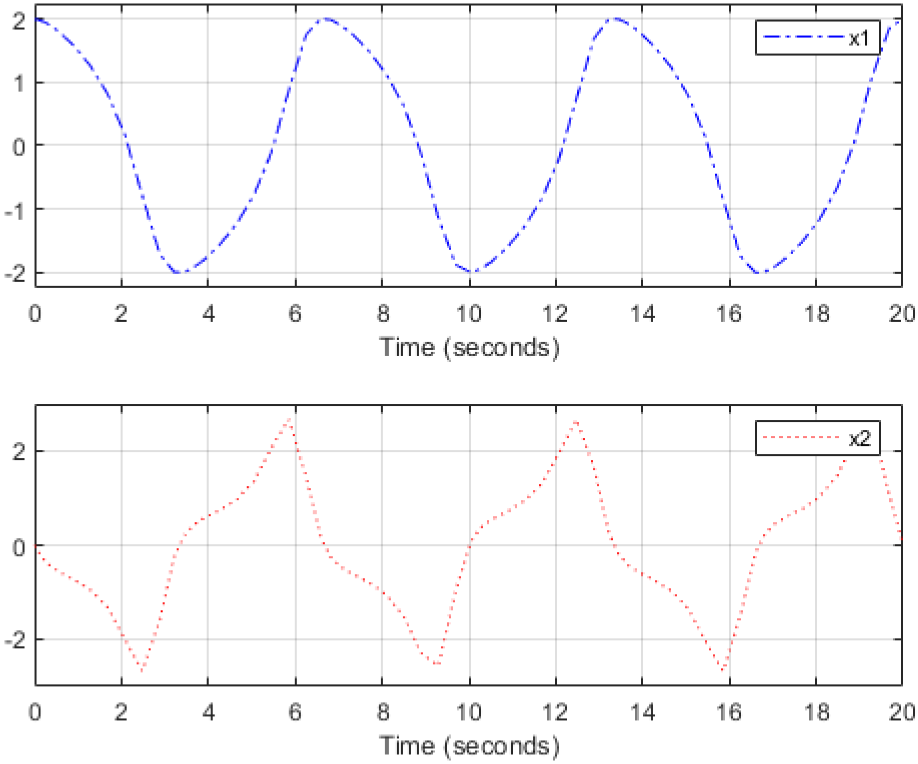
Use the `Simulink.sdi.Run` object to access the signals in the run. Then, modify the signals' view settings, and plot them in the Simulation Data Inspector. Open the Simulation Data Inspector and use `Simulink.sdi.snapshot` to view the results.

```
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

Capture a Snapshot from the Simulation Data Inspector

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```
snap = Simulink.sdi.CustomSnapshot;

snap.Rows = 2;
snap.YRange = {[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```



Copy the View Settings to a New Simulation Run

Simulate the model again, with a different `Mu` value. Then, visualize the new run by copying the view settings from the first run. Specify the `plot` input as `true` to plot the signals from the new run.

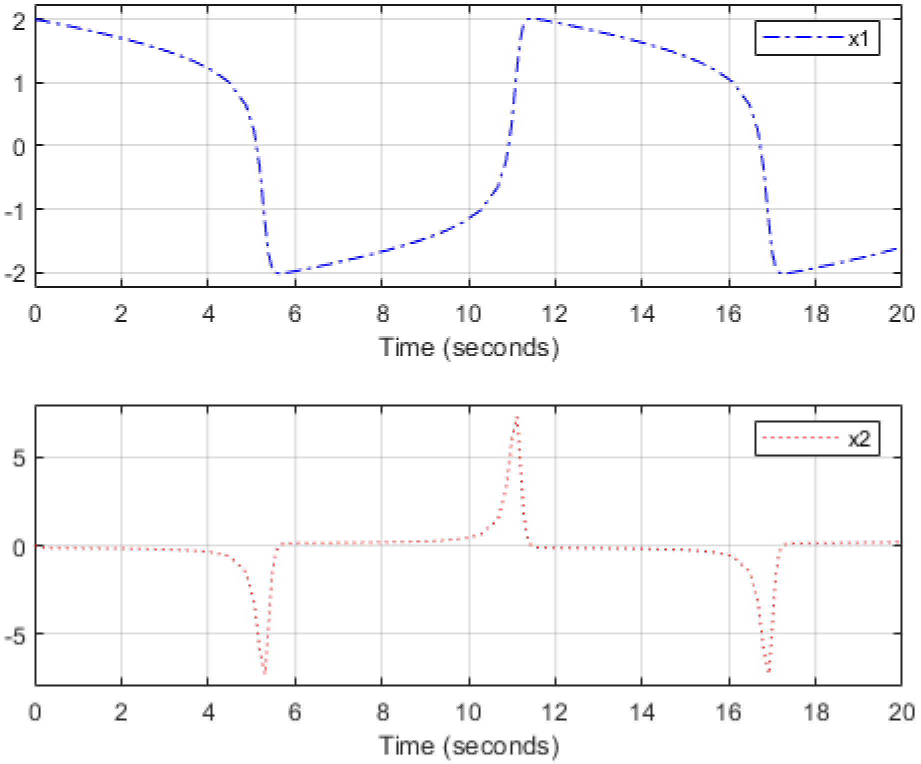
```
set_param('vdp/Mu','Gain','5')
sim('vdp')
```

```
runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);

sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,true);
```

Capture a Snapshot of the New Simulation Run

Use the `Simulink.sdi.CustomSnapshot` object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```
snap.clearSignals
snap.YRange = {[-2.25 2.25],[-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```



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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'to','figure','props',{'Name','My Data'}`

`from` — Content to include in the snapshot

`'opened'` (default) | `'active'` | `'comparison'` | `'custom'`

Content to include in the snapshot.

- `'opened'` — Include all subplots in the graphical viewing area of the open Simulation Data Inspector session.
- `'active'` — Include only the active (selected) subplot in the open Simulation Data Inspector session.
- `'comparison'` — Include the comparison plots for the selected comparison run or signal in the open Simulation Data Inspector session.
- `'custom'` — Include contents specified by the `settings` name-value pair `Simulink.sdi.CustomSnapshot` object. You can use the `'from','custom'` option to create a snapshot without opening the Simulation Data Inspector or affecting your open Simulation Data Inspector session. Include a `settings` name-value pair when you specify `'from','custom'`.

Example: `'from','comparison'`

Data Types: `char` | `string`

`to` — Type of snapshot to create

`'image'` (default) | `'figure'` | `'file'` | `'clipboard'`

Type of snapshot to create.

- `'image'` — Create a figure and return the figure handle and an array of image data. When you specify `'to','image'`, the `fig` and `image` outputs both have value.
- `'figure'` — Create a figure and return the figure handle. When you specify `'to','figure'` the `fig` output has value, and the `image` output is empty.
- `'file'` — Save to a PNG file with the name specified by the `filename` name-value pair. If you do not specify a `filename` name-value pair, the file is named `plots.png`. When you specify `'to','file'`, the `fig` and `image` outputs are both empty.
- `'clipboard'` — Copy the plots to your system clipboard. From the clipboard, you can paste the image into another program such as Microsoft Word. When you specify `'to','clipboard'`, the `fig` and `image` outputs are both empty.

Example: `'to','file'`

Data Types: `char` | `string`

`filename` — Name for image file

`'plots.png'` (default) | character array | string

Name of the image file to store the snapshot when you specify `'to','file'`.

Example: `'filename','MyImage.png'`

Data Types: `char` | `string`

`props` — Properties to customize the figure

cell array

Figure properties, specified as a cell array. You can include settings for the figure properties described in Figure Properties.

Example: `'props',{'Name','MyData','NumberTitle','off'}`

Data Types: `char` | `string`

`settings` — Custom snapshot settings

`Simulink.sdi.CustomSnapshot`

`Simulink.sdi.CustomSnapshot` object specifying custom snapshot settings. You can use the `settings` name-value pair to specify the dimensions of the image in pixels, subplot layout, and limits for the *x*- and *y*-axes.

Example: `'settings',customSnap`

Data Types: `char` | `string`

Output Arguments

`fig` — Figure handle

figure handle

Handle for the figure. When a figure is not created with your specified options, the `fig` output is empty.

`image` — Image data

array

Array of image data. The `image` output has value when you use `Simulink.sdi.snapshot` without any input arguments or without a `to` name-value pair and when you specify `'to','image'`.

See Also

`Simulink.sdi.CustomSnapshot` | `Simulink.sdi.Signal` | `Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.setMarkersOn` | `Simulink.sdi.setSubPlotLayout` | `Simulink.sdi.view`

Topics

"Inspect and Compare Data Programmatically"
"Create Plots Using the Simulation Data Inspector"

Introduced in R2018a

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
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.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 = sinusoidsRun.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 = sinusoidsRun.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')
```

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"
"View Data with the Simulation Data Inspector"

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.

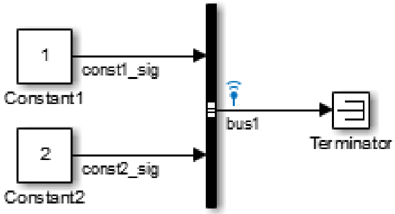
`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries( busObj,cellOfTimeseries)` creates a structure that matches the attributes of the bus object `busObj` and sets the values of structure leaf nodes using a cell array of MATLAB `timeseries` objects `cellOfTimeseries`.

`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 `logsout`.



View the logged data.

```
logsout
```

```
logsout =

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(logsout.bus1)
```

```
struct_of_ts =
    const1-sig: [1x1 timeseries]
    const2_sig: [1x1 timeseries]
```

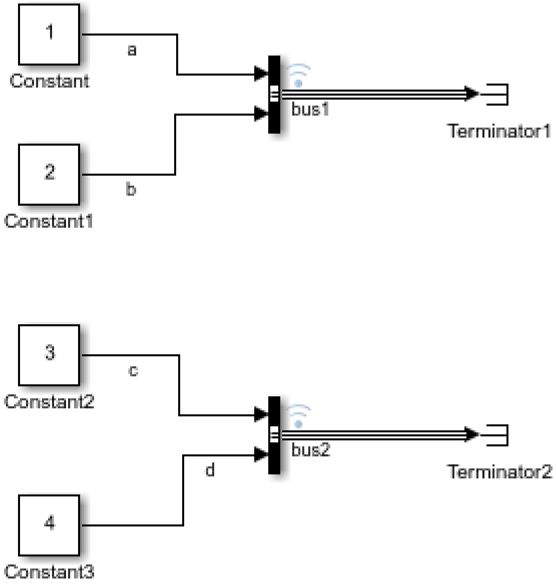
Structure Based on Bus Object and a Partial Structure of Timeseries Data

Create a structure of MATLAB `timeseries` objects based on a `Simulink.Bus` object and a partial structure of MATLAB `timeseries` objects. Then, load the structure into another model as simulation input.

Create a Structure of Timeseries

Open the `ex_logstructTimeSeries` model, and run a simulation to create the structure to load with signal logging.

```
open_system('ex_log_structTimeSeries')
sim('ex_log_structTimeSeries')
```



View Simulation Output

View the logged data.

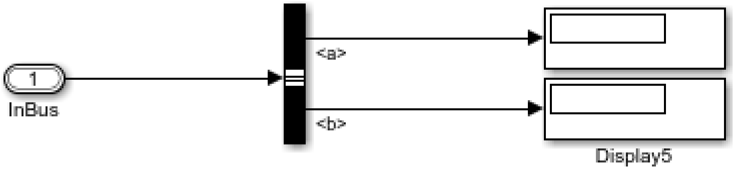
```
ex_log_structTimeSeries_logsout
```

```
ex_log_structTimeSeries_logsout = 

Simulink.SimulationData.Dataset 'ex_log_structTimeSeries_logsout' 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 Model

Open the model that uses the logged simulation data as input.

```
open_system('ex_load_structTimeSeries_Bus')
```



Use `Simulink.SimulationData.createStructOfTimeseries` to Create Simulation Input

Open the `ex_load_structTimeSeries_Bus` model Configuration Parameters and view the specification for the Input parameter on the Data Import/Export pane. The `ex_load_structTimeSeries_Bus` model uses a variable, `ex_load_struct_TimeSeries_inputBus` as an input. Use `Simulink.SimulationData.createStructOfTimeseries` to create `ex_load_structTimeSeries_inputBus` in the base workspace from the data in | ex_log_structTimeSeries_logsout.

```
ex_load_structTimeSeries_inputBus = ...
Simulink.SimulationData.createStructOfTimeseries...
('bus', ex_log_structTimeSeries_logsout.get(2).Values)
```

```
ex_load_structTimeSeries_inputBus = 

  struct with fields:

    a: [1x1 timeseries]
    b: [1x1 timeseries]
```

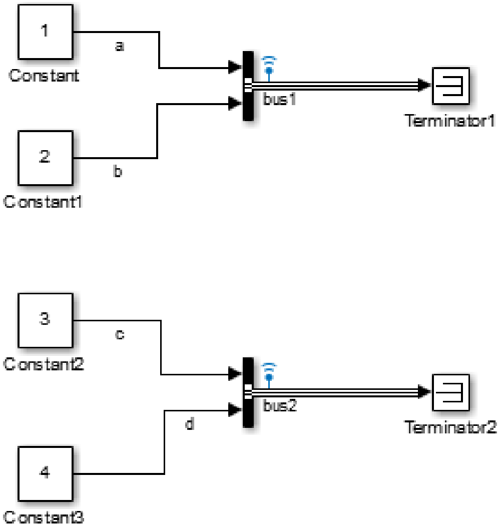
Structure to Use with an Array of Buses

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:



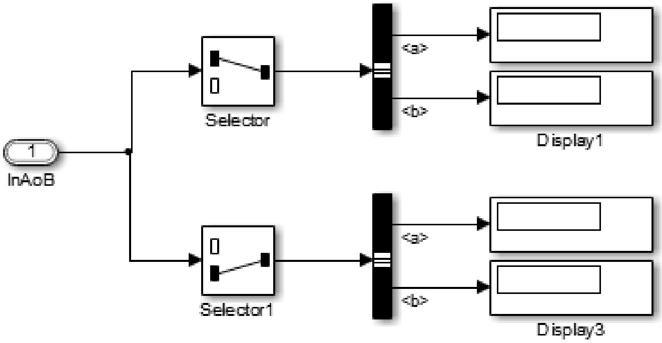
View the logged signal data.

```
ex_log_structTimeSeries_logsout
```

```
Simulink.SimulationData.Dataset 'ex_log_structTimeSeries_logsout' 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_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_logsout.get(1).Values.a,...
ex_log_structTimeSeries_logsout.get(1).Values.b,...
ex_log_structTimeSeries_logsout.get(2).Values.c,...
ex_log_structTimeSeries_logsout.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`.

See Also

`Simulink.Bus` | `Simulink.ModelDataLogs` | `Simulink.ModelDataLogs.convertToDataset` | `Simulink.TsArray`

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_element
s)
```

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&#x2019;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 (`logsout`) in the `out.mat` MAT-file.

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logsout');
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: [1×1 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');
```

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','logsout');
firstSig = sigLogRef{1}
```

```
ans =
  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
              Name: 'x1'
    PropagatedName: ''
         BlockPath: [1x1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 1
            Values: [1×1 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'  'logsout'
```

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`.

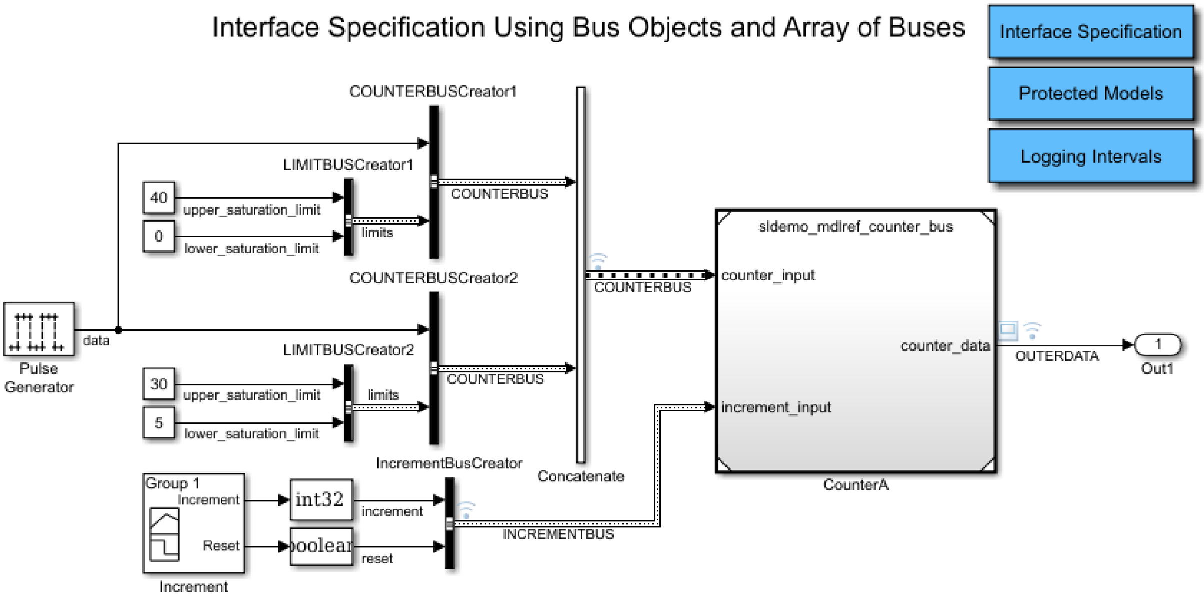
Examples

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_mdlref_bus')
sim('sldemo_mdlref_bus');
```



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_mdlref_bus/Concatenate
    2  [1x1 Signal]      OUTERDATA     sldemo_mdlref_bus/CounterA
    3  [1x1 Signal]      INCREMENTBUS  sldemo_mdlref_bus/IncrementBusCreator
    4  [1x1 Signal]      INNERDATA     ...erA|sldemo_mdlref_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 = 

  2×1 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_mdlref_bus.mdl`.

```
Simulink.SimulationData.signalLoggingSelector('sldemo_mdlref_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'
```

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 2-900

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 is 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 is 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.

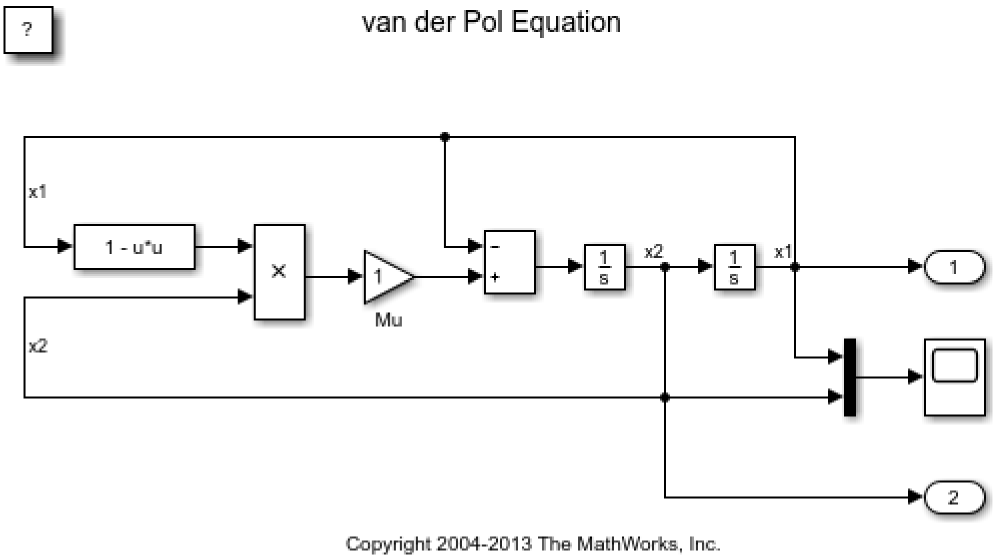
Examples

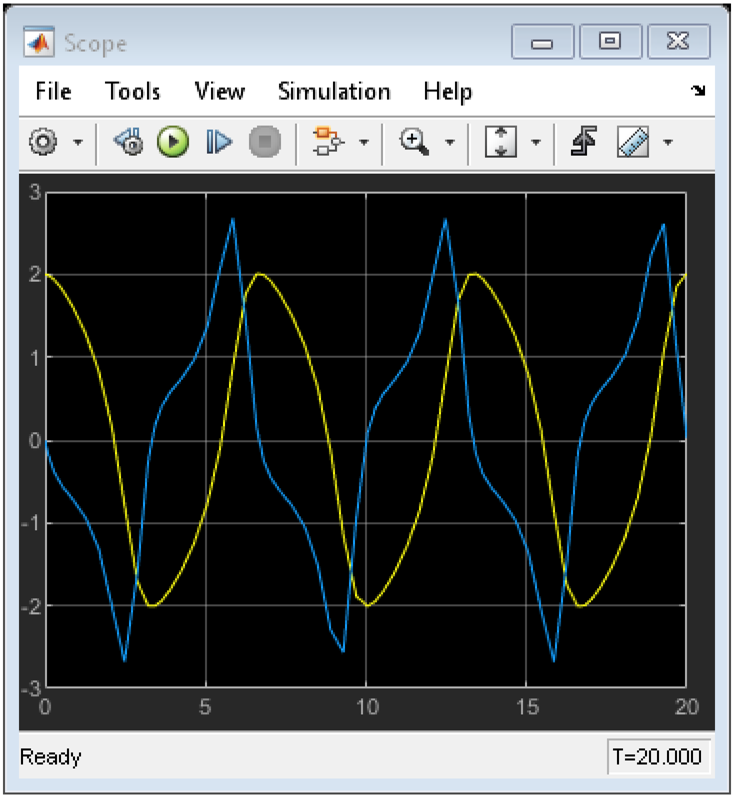
Retrieve Metadata From vdp Simulation

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');
```





Retrieve metadata information about this simulation using `mData`.

```
mData=simout.getSimulationMetadata()
```

```
mData = 

  SimulationMetadata 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.

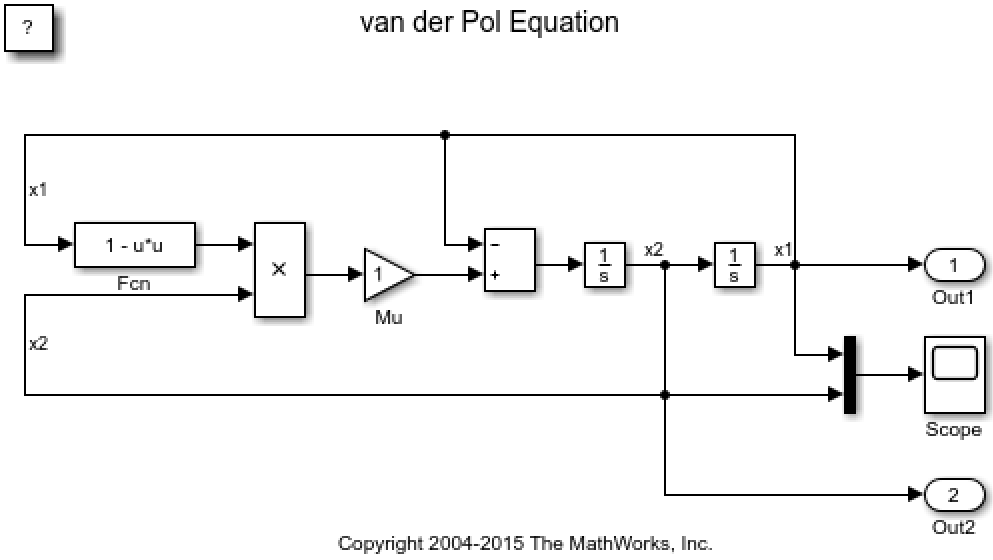
Examples

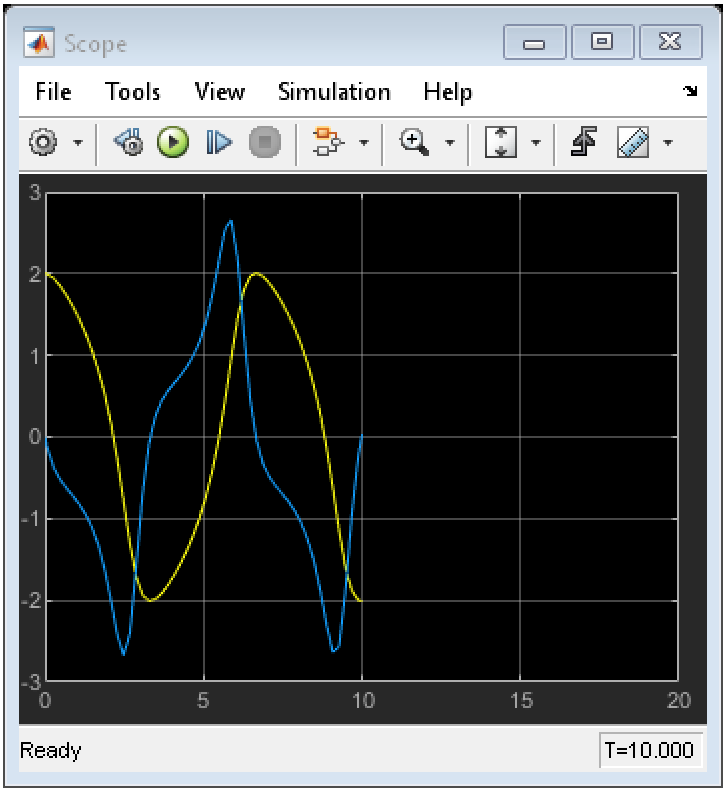
Store Data in SimulationMetadata Object of vdp Simulation

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');
```





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.

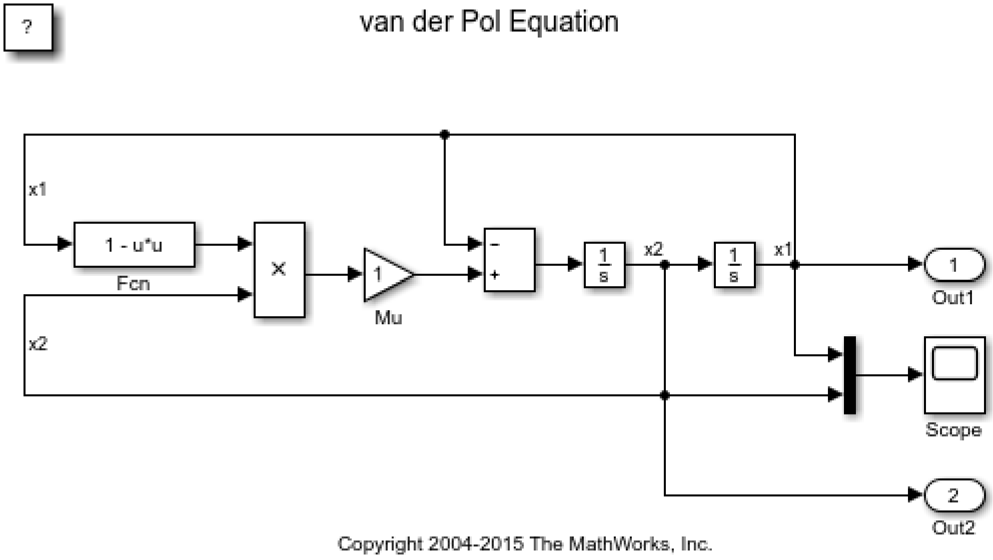
Examples

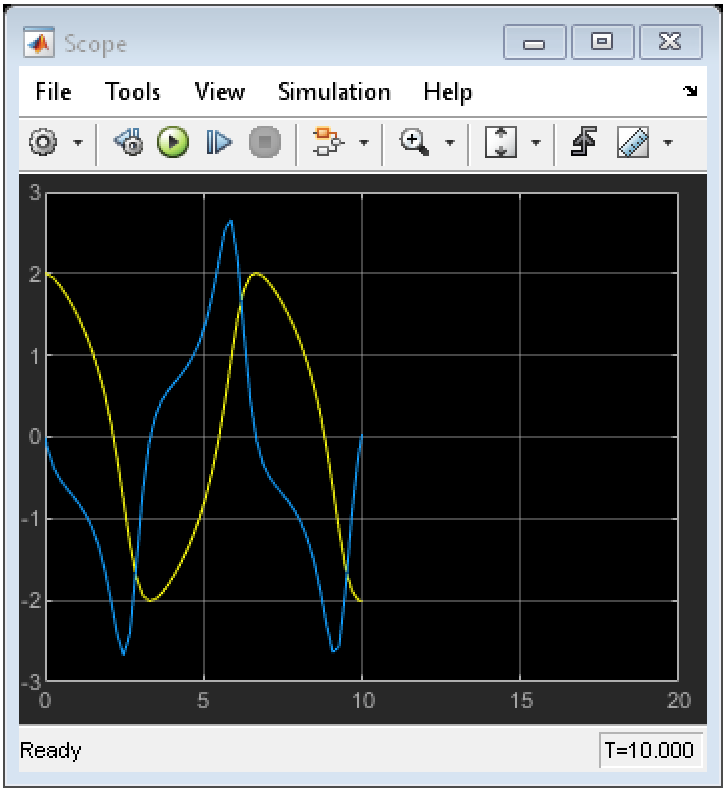
Store a Character Vector in SimulationMetadata Object of vdp Simulation

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');
```





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 is 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);
```

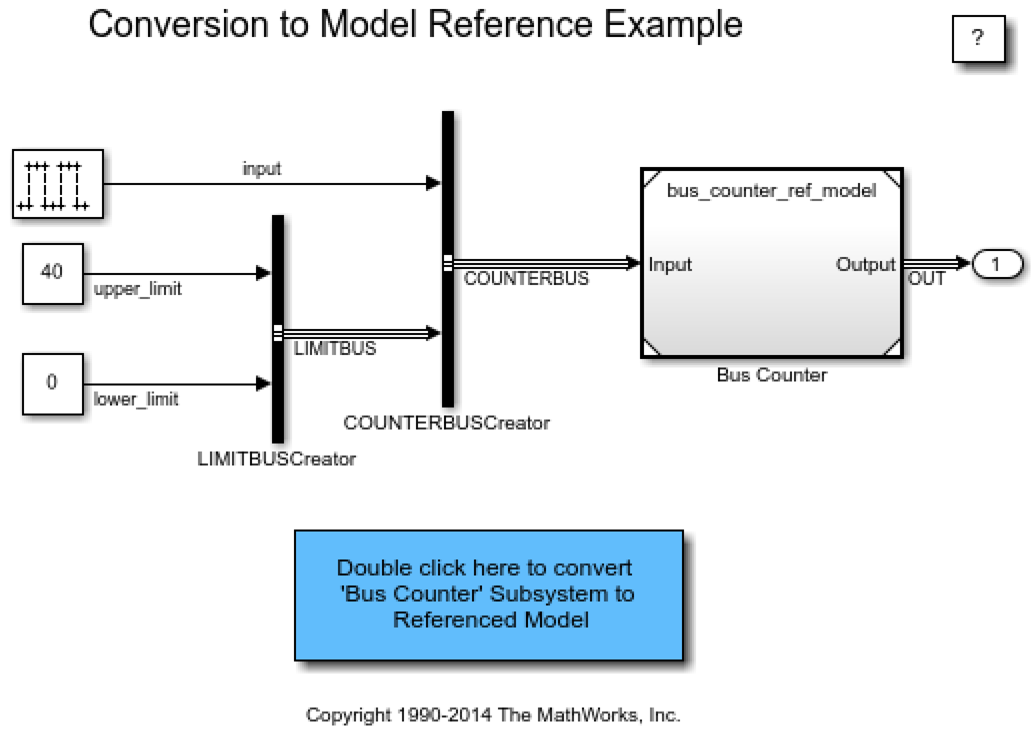
Perform the conversion using the advisor.

Convert Subsystem to Referenced Model

Convert the `Bus Counter` subsystem to a referenced model named `bus_counter_ref_model`.

```
open_system('sldemo_mdlref_conversion');
Simulink.SubSystem.convertToModelReference(...
   'sldemo_mdlref_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
?
input
40
upper_limit
0
lower_limit
LIMITBUS
LIMITBUSCreator
COUNTERBUS
COUNTERBUSCreator
bus_counter_ref_model
Input
Output
OUT
1
Bus Counter
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)
```

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 on which subsystems you can convert, see “Modify Referenced Models for Conditional Execution”.

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 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 a `'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 and 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 “Create 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_mdlref_conversion
“Convert Subsystems to Referenced Models”
“Model References”

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:

```
load_system('f14');
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: [4×1 uint32]
    MarkedUnique: 0


chksum1_details =

  struct with fields:

          ContentsChecksum: [1×1 struct]
         InterfaceChecksum: [1×1 struct]
     ContentsChecksumItems: [359×1 struct]
    InterfaceChecksumItems: [60×1 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');
```

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

slblocksearchdb.trainfrommodel

Train suggestion engine to improve quick insert results based on one model

Syntax

```
slblocksearchdb.trainfrommodel(model)
```

Description

`slblocksearchdb.trainfrommodel(model)` improves quick insert search results based on a single model.

Examples

Use a Single Model for Training

Train the suggestion engine to use a model to improve results. Assume `'MyModels'` is in the current folder. This code uses a relative path.

```
slblocksearchdb.trainfrommodel('MyModels/model.slx')
```

Input Arguments

`model` — Model for training the suggestion engine
model path

Model to train the suggestion engine, specified as a character vector or string scalar of the full model path or relative path. Include the `.slx` or `.mdl` extension

Example: `'H:/MyModels/model.slx'` `'mymodel.slx'`

See Also

`slblocksearchdb.trainfrommodelsindir` | `slblocksearchdb.untrainall` | `slblocksearchdb.untrainmodel` | `slblocksearchdb.untrainmodelsindir`

Topics

"Improve Quick Block Insert Results"

Introduced in R2018a

slblocksearchdb.trainfrommodelsindir

Train suggestion engine to improve quick insert results based on models in a folder

Syntax

```
slblocksearchdb.trainfrommodelsindir(folder)
slblocksearchdb.trainfrommodelsindir(folder,'exclude',exclusions)
```

Description

`slblocksearchdb.trainfrommodelsindir(folder)` improves search results based on the models in `folder`, recursively.

`slblocksearchdb.trainfrommodelsindir(folder,'exclude',exclusions)` excludes the specified models from updating the suggestion engine.

Examples

Train Suggestion Engine and Exclude Folders and Models

Create the cell array `exPath` for a folder you want to exclude and a model you want to exclude. To train the suggestion engine, use the cell array in the `slblocksearchdb.trainfrommodelsindir` command.

```
exPath = ('MyModels/testmodels' 'MyModels/myvdp.slx'}
slblocksearchdb.trainfrommodelsindir('MyModels','exclude',exPath)
```

The command uses the models in the folder `'MyModels'` with the exclusions you specified.

Input Arguments

`folder` — Folder whose models to use for training

folder path

Folder whose models to use for training the suggestion engine, specified as an absolute or relative path character vector or string scalar.

Example: `'H:/MyModels/trainingmodels'` `'MyModels'`

`exclusions` — Models to exclude from training models

cell array of character vectors | string array

Models to exclude from training the suggestion engine, specified as a cell array of character vectors or a string array. Specify folders or models to exclude as a full or relative path. Model names must include the file extension `.slx` or `.mdl`.

Example: `{'H:/MyModels/trainingmodels' 'MyModels/trainingmodels2' 'MyModels/myvdp.slx'}`

See Also

`slblocksearchdb.trainfrommodel` | `slblocksearchdb.untrainall` | `slblocksearchdb.untrainmodel` | `slblocksearchdb.untrainmodelsindir`

Topics

“Improve Quick Block Insert Results”

Introduced in R2018a

slblocksearchdb.untrainall

Remove the effects of all added models from the suggestion engine

Syntax

```
slblocksearchdb.untrainall
```

Description

`slblocksearchdb.untrainall` removes the models added to the suggestion engine to improve quick insert results. Use this function when you want to return the database to the default state.

Examples

Add Models and Remove All

Add some models to the suggestion engine.

```
slblocksearchdb.trainfrommodelsindir('MyModels')
slblocksearchdb.trainfrommmodel('C:/users/TrainingModels/mymodel.slx')
```

Remove all added models from the suggestion engine.

```
slblocksearchdb.untrainall
```

See Also

`slblocksearchdb.trainfrommodel` | `slblocksearchdb.trainfrommodelsindir` | `slblocksearchdb.untrainmodel` | `slblocksearchdb.untrainmodelsindir`

Topics

"Improve Quick Block Insert Results"

Introduced in R2018a

slblocksearchdb.untrainmodel

Remove the effect of a model from the suggestion engine

Syntax

```
slblocksearchdb.untrainmodel('model')
```

Description

`slblocksearchdb.untrainmodel('model')` removes the effects of a single model from the suggestion engine.

Examples

Remove a Model from the Suggestion Engine

Train the suggestion engine to use a model to improve results.

```
slblocksearchdb.trainfrommodel('MyModels/model.slx')
```

Remove the model from the suggestion engine.

```
slblocksearchdb.untrainmodel('MyModels/model.slx')
```

Input Arguments

`model` — Model to remove from the suggestion engine
model path

Model whose effects to remove from the suggestion engine, specified as the full or relative path character vector or string scalar. Include the `.slx` or `.mdl` extension.

Example: `'H:/MyModels/model.slx'`

See Also

`slblocksearchdb.trainfrommodel` | `slblocksearchdb.trainfrommodelsindir` | `slblocksearchdb.untrainall` | `slblocksearchdb.untrainmodelsindir`

Topics

"Improve Quick Block Insert Results"

Introduced in R2018a

slblocksearchdb.untrainmodelsindir

Remove the effects of models from the suggestion engine

Syntax

```
slblocksearchdb.untrainmodelsindir(folder)
slblocksearchdb.untrainmodelsindir(folder,'exclude',exclusions)
```

Description

`slblocksearchdb.untrainmodelsindir(folder)` removes models in `folder` from the suggestion engine, recursively.

`slblocksearchdb.untrainmodelsindir(folder,'exclude',exclusions)` excludes the specified models from updating the suggestion engine.

Examples

Remove the Effects of Models from Suggestion Engine

Create the cell array `exPath` for a folder and a model whose effects you do not want to remove from the suggestion engine. Then use the cell array in the `slblocksearchdb.untrainmodelsindir` command.

```
exPath = {'MyModels/subfolder' 'MyModels/myvdp.slx'}
slblocksearchdb.untrainmodelsindir('MyModels','exclude',exPath)
```

The command removes the models in the folder `'MyModels'` with the exclusions you specified.

Input Arguments

`folder` — Folder whose models to remove from the suggestion engine

folder path

Folder whose models to remove from the suggestion engine, specified as an absolute or relative path character vector or string scalar.

Example: `'H:/MyModels/trainingmodels'` `'MyModels'`

`exclusions` — Folders and models to exclude from removing

cell array of character vectors | string array

Folders or models to exclude from removing from the suggestion engine, specified as a cell array of character vectors or a string array. Specify the folders or models to exclude as a full or relative path. For models, include the `.slx` or `.mdl` extension.

Example: `{'H:/MyModels/trainingmodels' 'MyModels/trainingmodels2' 'MyModels/myvdp.slx'}`

See Also

`slblocksearchdb.trainfrommodel` | `slblocksearchdb.trainfrommodelsindir` | `slblocksearchdb.untrainall` | `slblocksearchdb.untrainmodel`

Topics

"Improve Quick Block Insert Results"

Introduced in R2018a

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 `spmd` 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_rtwintro`.

```
slbuild('rtwdemo_rtwintro')
% same operation as ...
% slbuild('rtwdemo_rtwintro','StandaloneRTWTarget')
```

For the GRT target, the coder generates the following code files and places them in folders `rtwdemo_rtwintro_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Files	Interface Files	Other Files

`rtwdemo_rtwintro.c`
`rtwdemo_rtwintro.h`
`rtwdemo_rtwintro_private.h`
`rtwdemo_rtwintrotypes.h`
	`rtwtypes.h`
	`multiword_types.h`
	`builtin_typeid_types.h`
		`rtmodel.h`	none

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_rtwintro.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_mdlreftop`, which refers to model `rtwdemo_mdlrefbot`, regardless of model checksums and parameter settings.

```
slbuild('rtwdemo_mdlreftop','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_rtwintro','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 *`model`*, using the current model configuration settings. If the *`model`* has not been loaded, `slbuild` 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

```
slbuild(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 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`

OpenBuildStatusAutomatically — Display build information in the Build Process Status Window

`false` (default) | `true`

Display build information in the **Build Process Status** window, specified as `true` or `false`. For more information about using the status window, see “View Build Process Status” (Simulink Coder).

The **Build Process Status** window support parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

Action	Specify
Display build information in the Build Process Status Window	`true`
No action	`false`

See Also

`rtwbuild` | `rtwrebuild`

Topics

“Model Reference Simulation Targets”
“What Is Acceleration?”
“Perform Acceleration”
“Share 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 `slCharacterEncoding` function to change the character encoding

1. Close all open models.
2. Use `slCharacterEncoding` 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_mdlrefs` for more information on finding all Model blocks and referenced models in a specified model.

Option	Valid Inputs	Output
`CountBlocks`	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.
`CountSF`	root model, library, or subsystem	Lists all unique Stateflow objects in the system and the number of occurrences of each.
`Sizes`	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.
`Libs`	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.
`CompileStats`	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.
`RTWBuildStats`	root model	Lists the same information as the `CompileStats` diagnostic. When issued with the second output argument `sRpt`, it captures the same statistics included in `CompileStats` 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')
```

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:

`target`	`'all'` (default)	Discretize all continuous blocks
	`'selected'`	Discretize only selected blocks in the model
	`'`*`full_blk_path`*`'`	Discretize specified block
`replace_with`	`'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
`put_into`	`'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
`prompt`	`'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

`slmdldiscui`

Topics

"Discretize a Model with the sldiscmdl Function"

Introduced before R2006a

slexpr

Generate expression to use in value of parameter object

Syntax

```
expressionOut = slexpr(expressionIn)
```

Description

`expressionOut = slexpr(expressionIn)` converts the MATLAB-syntax expression `expressionIn` to an object, `expressionOut`, that you can use to set the `Value` property of a parameter object (such as `Simulink.Parameter`). When you use multiple parameter objects to set block parameter values, you can use the expression to model mathematical relationships between the objects. For more information, see “Set Variable Value by Using a Mathematical Expression”.

Examples

Model Relationship Between Mass, Length, and Moment of Inertia of Metronome

In the base workspace, create three `Simulink.Parameter` objects that represent the mass, length, and moment of inertia of a pointlike metronome.

```
m = Simulink.Parameter;
r = Simulink.Parameter;
J = Simulink.Parameter;
```

Set the mass to 0.1 kg and the length to 1.0 m.

```
m.Value = 0.1;
r.Value = 1.0;
```

Set the value of the moment of inertia to the mass times the square of the length.

```
J.Value = slexpr('m*r^2');
```

Simulink preserves the expression, `m*r^2`. If you change the value of the mass or the length, Simulink recalculates the value of the moment of inertia.

Input Arguments

`expressionIn` — Target expression

string | character vector

Target expression, specified as a string or character vector.

Example: `"myParam + myOtherParam"`

Data Types: `char` | `string`

Output Arguments

`expressionOut` — Simulink representation of expression

`Simulink.data.Expression` object

Simulink representation of the target expression, returned as a `Simulink.data.Expression` object. A `Simulink.data.Expression` object has no use outside the `Value` property of a parameter object.

See Also

`Simulink.Parameter`

Topics

“Share and Reuse Block Parameter Values by Creating Variables”
“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

Introduced in R2018a

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

```
slLibraryBrowser
slLibraryBrowser('close')
```

Load the Library Browser and Get a Handle

```
libraryhandle = slLibraryBrowser('noshow')
```

See Also

`simulink` | `start_simulink`

Topics

“Build and Edit a Model in the Simulink Editor”
“Model Editing Environment”

Introduced in R2016a

slmdldiscui

Open Model Discretizer GUI

Syntax

```
slmdldiscui
slmdldiscui('model')
```

Description

`slmdldiscui` opens the Model Discretizer. A model does not need to be open.

`slmdldiscui('`*`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:

```
slmdldiscui('slexAircraftExample')
```

Open the Model Discretizer for the `discretizing` library:

```
slmdldiscui('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_name`*`ProfileData)`

Description

When you run a model with the profiler enabled, the simulation generates the data and saves it in the variable, *`model_name`*`ProfileData`. `slprofreport(`*`model_name`*` ProfileData)` generates a profiler report based on the data in *`model_name`*`ProfileData`, 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.

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 project 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');
```

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(filepath);
```

Add this new folder to the project.

```
 projectFile = addFile(project,folderpath);
```

Add this new folder to the project path.

```
folderpath = addPath(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 | string

Path of the folder to add relative to the project root folder, specified as a character vector or string. 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(filepath);
```

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);
```

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                         isLoaded
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

Project Property	Description
Files	Paths and names of project files
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(proj, file);
```

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 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 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 Simscape states to file

off (default) | on

Enable this parameter to save Simscape 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 “Compare Solvers”.

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

Option 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
`jacobianNumber`	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”.
`exceptionNumber`	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

“Solver Types”
“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
```

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

stringtype

Create string data type

Syntax

```
string = stringtype(maximum_length)
stringtype(maximum_length)
```

Description

`string = stringtype(maximum_length)` creates a Simulink string data type with a maximum length. Alternatively, you can also create string data types using the String Constant, String Concatenate, and Compose String blocks.

`stringtype(maximum_length)` creates a Simulink string data type with a maximum length that you can type directly on the MATLAB command line or in the **Output data type** parameter of the String Constant, String Concatenate, or Compose String block.

Examples

Create a String Data Type of Maximum Length 10

Create a string data type of maximum length 10.

```
h=stringtype(10)
```

```
h = 

  StringType with properties:

    MaximumLength: 10
      Description: ''
```

```
         DataScope: 'Auto'
        HeaderFile: ''
```

Input Arguments

`maximum_length` — Maximum length
scalar

Maximum length of string data type, specified as a scalar, from 1 to 32766. This value can be an integer, MATLAB variable, or MATLAB expression.

Data Types: `double`

Output Arguments

`string` — String data type object
scalar

String object, specified as a scalar.

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

Topics

“Simulink Strings”

Introduced in R2018a

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-x0,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 `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.
5	N/A	Not used.

No.	Default	Description
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 $(1 - 2^{-8}).2^{4}$ = 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., `logsout`).

`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 2-1047.

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_mdlrefs

Display graph of model referencing dependencies without library dependencies

`view_mdlrefs` opens the “Model Dependency Viewer” with library dependencies omitted. To open a specific configuration of the Model Dependency Viewer, use `depview` with programmatic options. The Model Dependency Viewer provides the same options regardless of how you open it.

Syntax

```
view_mdlrefs(sys)
```

Description

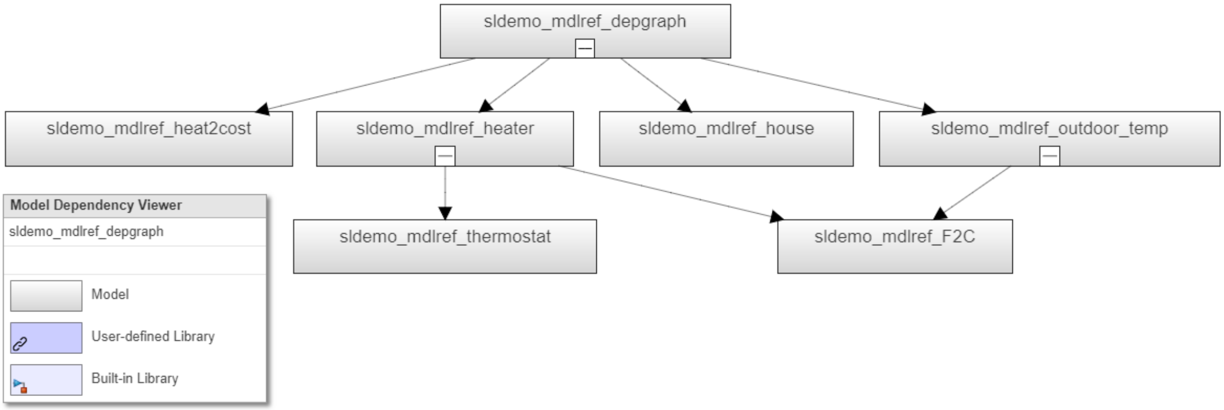
`view_mdlrefs(sys)` opens the Model Dependency Viewer, which displays a graph of model referencing dependencies for the specified model. The nodes in the graph represent Simulink models. The directed lines indicate model dependencies.

Examples

Open Model Dependency Viewer with Default Settings

Open the Model Dependency Viewer for the model `sldemo_mdlref_depgraph`.

```
view_mdlrefs('sldemo_mdlref_depgraph');
```



Input Arguments

sys — Model name or path

character vector

The full name or path of a model, specified as a character vector.

Data Types: `char`

See Also

Model | `depview` | `find_mdlrefs`

Topics

“Model References”
“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

3

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
	Promote a block icon image to the masked Subsystem

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.

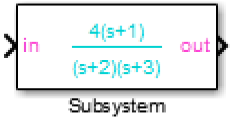
Examples

The following commands

```
color('cyan');
droots([-1], [-2 -3], 4)
color('magenta')
```

```
port_label('input',1,'in')
port_label('output',1,'out')
```

draw the following mask icon.



See Also

`droots` | `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.

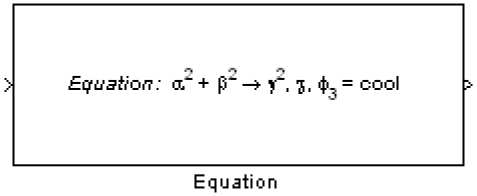
disp(text, 'texmode', 'on') allows you to use TeX formatting commands in *text*. The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See “Interpreter” (MATLAB) for information on the TeX formatting commands supported by Simulink software.

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.


Equation: $\alpha^2 + \beta^2 \rightarrow \gamma^2, \chi, \phi_3 = \mathbf{cool}$
Equation

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*.

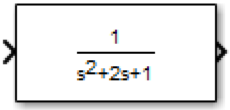
`dpoly(num, den, 'character')` specifies the name of the transfer function independent variable. The default is *s*.

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)
  ```

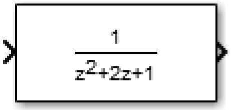
  For example, for `num = [0 0 1];` and `den = [1 2 1]` the icon looks like:

- To display a discrete transfer function in descending powers of *z*, enter

  ```
  dpoly(num, den, 'z')
  ```

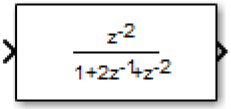
  For example, for `num = [0 0 1];` and `den = [1 2 1];` the icon looks like:



- 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:



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*.

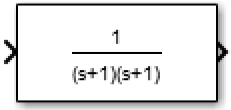
When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- 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(formatSpec, var)
```

Description

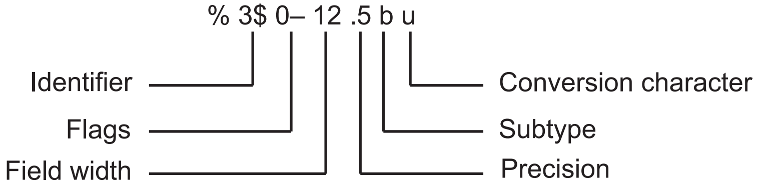
The `fprintf` command displays formatted text centered on the icon and can display *`formatSpec`* along with the contents of *`var`*.

Note While this `fprintf` function is identical in name to its corresponding MATLAB function, it provides only the functionality described on this page.

`formatSpec` can be a character vector in single quotes, or a string scalar.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).



Conversion Character

This table shows conversion characters to format numeric and character data as text.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a`–`f`
	`%X`	Same as `%x`, uppercase letters `A`–`F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters or strings	`%c`	Single character
	`%s`	Character vector or string array. The type of the output text is the same as the type of `formatSpec`.

Examples

The command

```
fprintf('Hello');
```

displays the text `'Hello'` on the icon.

The command

```
fprintf('Hello = %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
[*x*, *y*, *w*, *h*]	Position (*x*, *y*) and size (*w*, *h*) of the image where the position is relative to the lower-left corner of the mask. The image scales to fit the specified size.
`'center'`	Center of the mask
`'top-left'`	Top left corner of the mask, unscaled
`'bottom-left'`	Bottom left corner of the mask, unscaled
`'top-right'`	Top right corner of the mask, unscaled
`'bottom-right'`	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
`image('icon.jpg')`	Reads the icon image from a JPEG file named `icon.jpg` in the MATLAB path.

`[data, map]=image('label.gif');`
`pic=ind2rgb(data,map);`
	Reads and converts a GIF file, `label.gif`, to the appropriate matrix format.

`image(pic)`	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])
```

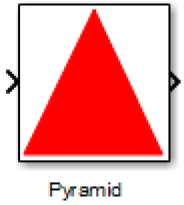
creates a red triangle on the mask's icon.

Examples

The command

```
patch([0 .5 1], [0 1 0], [1 0 0])
```

creates a red triangle on the mask's icon.



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 `inf`s, 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

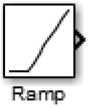
Note The plot command supports all numeric data types.

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 enable 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

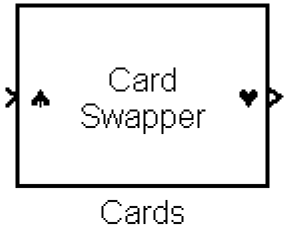
```
port_label('trigger','$\sqrt m$','interpreter','tex')
```

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

Option	Aligns
'bottom'	The bottom line of the text at the specified point
'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')
```

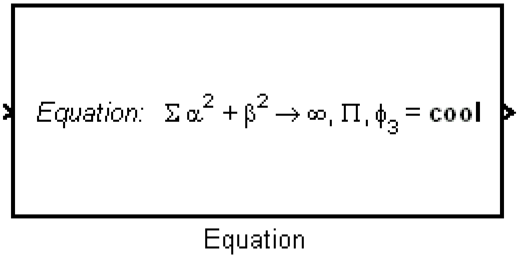
Equation in Mask Icon

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: $\Sigma\alpha^2 + \beta^2 \rightarrow \infty, \Pi, \phi_3 = \mathbf{cool}$
Equation

See Also

`disp` | `fprintf` | `port_label`

Introduced before R2006a

block_icon

Promote a block icon image to the masked Subsystem

Syntax

```
block_icon(BlockName)
```

Description

`block_icon(BlockName)` displays the underneath block icon image on the masked Subsystem icon. For more information, see slexblockicon.

Input Arguments

`BlockName` — Name of the underneath block
String

The name of the underneath block whose icon image you want to display on the Subsystem block that encapsulates the specified block. For more information, see slexblockicon

Data Types: `string`

See Also

`Simulink.Mask` | slexblockicon

Introduced in R2006b

4

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
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
zcbreak	Toggle breaking at nonsampled zero-crossing events

ashow

Show algebraic loop

Syntax

```
ashow <gcb | s:b | s#n | clear>
```

```
as <gcb | s:b | s#n | clear>
```

Arguments

`gcb`	Current block.
`s:b`	The block whose system index is `s` and block index is `b`.
`s#n`	The algebraic loop numbered `n` in system `s`.
`clear`	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
`atrace 0`	No information
`atrace 1`	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
`atrace 2`	Same as level 1
`atrace 3`	Level 2 plus Jacobian matrix used to solve loop
`atrace 4`	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`**`> [`*`mth`*`] [`**`tid:`***`TID`*`]`

`bafter <`**`s:`***`sysIdx`*` | `**`gcs`**`> [`*`mth`*`] [`**`tid:`***`TID`*`]`

`bafter `**`model`**` [`*`mth`*`] [`**`tid:`***`TID`*`]`

Arguments

`mid`	Method ID
`sysIdx``:`*`blkIdx`*	Block ID
`gcb`	Currently selected block
`sysIdx`	System ID
gcs	Currently selected system
`model`	Currently selected model
mth	A method name, e.g., `Outputs.Major`
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 *`mth`* 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> [mth] [tid:TID]
```

```
break <s:sysIdx | gcs> [mth] [tid:TID]
```

```
break model [mth] [tid:TID]
```

Arguments

mid	Method ID
sysIdx:*blkIdx*	Block ID
gcb	Currently selected block
sysIdx	System ID
gcs	Currently selected system
model	Currently selected model
mth	A method name, e.g., `Outputs.Major`
TID	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 times 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.

Note Do not use a model name instead of identifier `model` in the `break model` command.

The optional *`mth`* 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
```

```
bshow modelName 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:`**`m`*`id`*

`clear` *`id`*

`clear <`*`sysIdx:blkIdx`*` | `**`gcb`**`>`

Arguments

`mid`	Method ID
`id`	Breakpoint ID
`sysIdx``:`*`blkIdx`*	Block ID
`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
```

```
disp  modelName 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

el

elist **m:***mid* [**tid:***TID*]

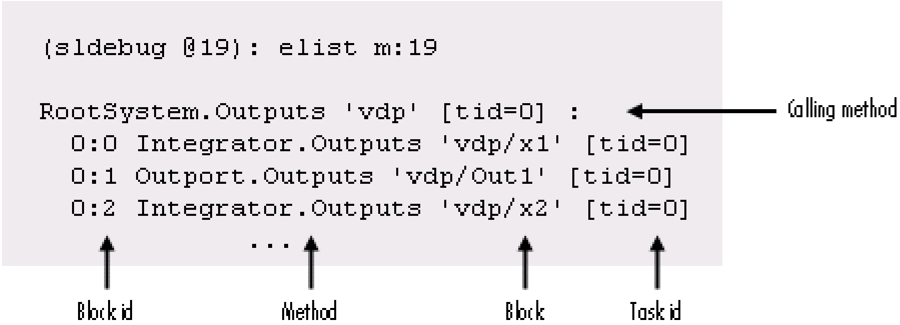
elist <**gcs** | **s:***sysIdx*> [*mth*] [**tid:***TID*]

elist <**gcb** | *sysIdx***:***blkIdx*> [*mth*] [**tid:***TID*]

Description

Instead of `elist`, you can use the short form of `el` 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] :
  0:0 Integrator.Outputs 'vdp/x1' [tid=0]
  0:1 Outport.Outputs 'vdp/Out1' [tid=0]
  0:2 Integrator.Outputs 'vdp/x2' [tid=0]
         ...
```

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

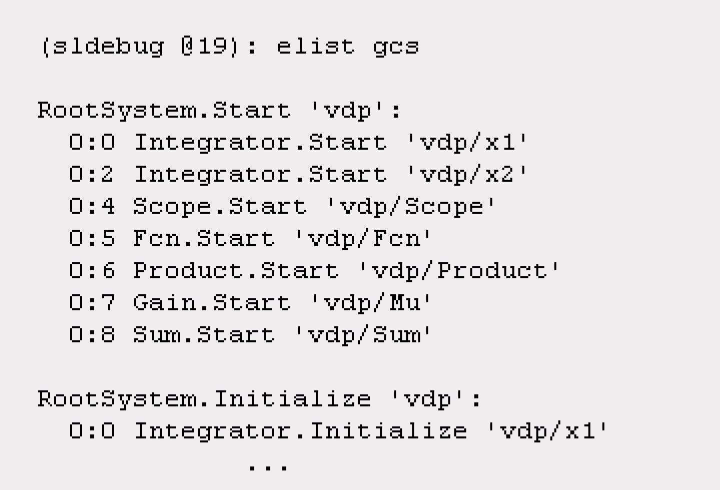
- The ID (*`sysIdx`*:*`blkIdx`*) of the block instance on which the method is invoked

  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.,

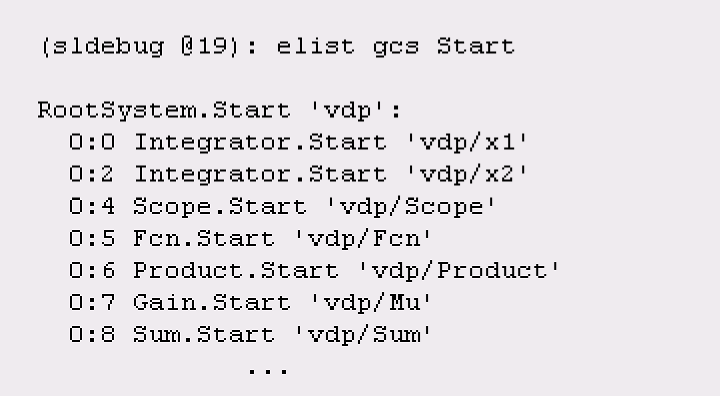


```
(sldebug @19): elist gcs

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 modelName 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` *`modelName s`*`:`*`b`* displays the I/O of the block system index is *`s`* and block index is *`b`* in the *`modelName`* model.

`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.

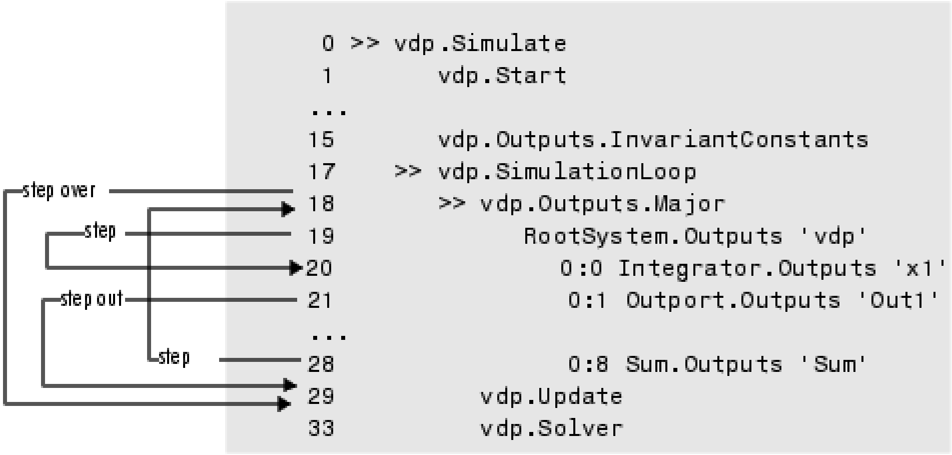
`s` is the short form of the command.

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`.



```
 0 >> vdp.Simulate
 1       vdp.Start
...
15       vdp.Outputs.InvariantConstants
17    >> vdp.SimulationLoop
18       >> vdp.Outputs.Major
19             RootSystem.Outputs 'vdp'
20                0:0 Integrator.Outputs 'x1'
21                 0:1  Outport.Outputs 'Out1'
...
28                 0:8 Sum.Outputs 'Sum'
29          vdp.Update
33          vdp.Solver
```

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 = 0.005777824246771424 ] [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

`slist`

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

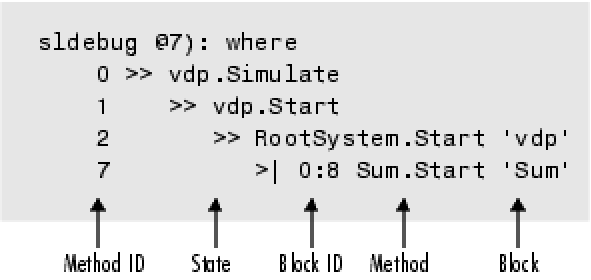
Syntax

```
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:

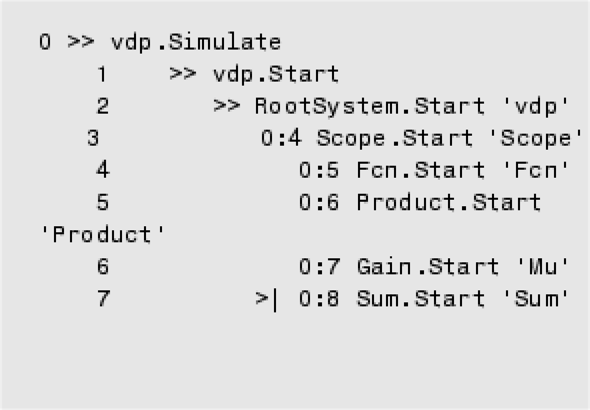
- Method ID

  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.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.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
Simulink.ModelDataLogs	Container for signal data logs of a model
Simulink.SimState.ModelSimState	Access SimState snapshot data
Simulink.MSFcnRunTimeBlock	Get run-time information about Level-2 MATLAB S-function block
Simulink.RunTimeBlock	Allow Level-2 MATLAB S-function and other MATLAB programs to get information about block while simulation is running

matlab.io.datastore.sdidatastore class

Package: matlab.io.datastore

Datastore for Simulation Data Inspector signals

Description

A `matlab.io.datastore.sdidatastore` object provides access to signals logged to the Simulation Data Inspector that are too large to fit into memory. An `sdidatastore` object references the data for a single signal. The `read` method loads the signal data referenced by an `sdidatastore` object in a chunk-wise manner such that each chunk always fits into memory. You can use an `sdidatastore` object to create a tall timetable for your signal data. For more information about working with tall arrays, see “Tall Arrays” (MATLAB).

Note `matlab.io.datastore.sdidatastore` does not support parallel computations. If you have a Parallel Computing Toolbox license, use `mapreducer(0)` to set the execution environment to the local MATLAB client before creating a tall timetable from the `matlab.io.datastore.sdidatastore`.

Construction

`ds = dsrObj.getAsDatastore(arg)` creates the `sdidatastore`, `ds`, for the signal in the `Simulink.sdi.DatasetRef` object selected by the search criterion `arg`. You can specify `arg` as an integer representing the index of the desired signal within the `Simulink.sdi.DatasetRef` object, or as a character vector containing the name of the signal.

`ds = matlab.io.datastore.sdidatastore(signalID)` creates the `sdidatastore`, `ds`, for the signal corresponding to the specified `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

Numeric signal identifier generated for the signal by the Simulation Data Inspector. You can get the signal ID for a signal using methods of the `Simulink.sdi.Run` object containing the signal or as a return from the `Simulink.sdi.createRun` function.

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 `sdidatastore`. The `Signal` property provides access to the signal data and metadata.

Methods

hasdata	Determine if data is available to read
preview	Return preview of data in sdidatstore
read	Read a chunk of data from an sdidatastore
readall	Read all data from an sdidatastore
reset	Reset the read position

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a Tall Timetable for a Simulation Data Inspector Signal

This example shows how to create a tall timetable for a signal in the Simulation Data Inspector repository. You can create the tall timetable using a `Simulink.sdi.Signal` object or by first creating a `matlab.io.datastore.sdidatastore` for the signal. You can use a `matlab.io.datastore.sdidatastore` to incrementally read and process signal data for signals that do not fit into memory. A tall timetable handles the data chunking and processing in the background. In general, you can work with tall timetables very similarly to how you work with in-memory data.

Create Data and Access Signal ID

Whether you create your tall timetable using a `Simulink.sdi.Signal` object or a `matlab.io.datastore.sdidatastore`, start by creating data and accessing the signal ID for a signal of interest. The `sldemo_fuelsys` model is configured to log signals which stream to the Simulation Data Inspector repository when you simulate the model.

```
open_system('sldemo_fuelsys')
sim('sldemo_fuelsys')
```

Then, use the Simulation Data Inspector programmatic interface to access the signal ID for a signal of interest. For example, access the `ego` signal.

```
runCount = Simulink.sdi.getRunCount;
latestRunID = Simulink.sdi.getRunIDByIndex(runCount);
latestRun = Simulink.sdi.getRun(latestRunID);

egoSigID = latestRun.getSignalIDByIndex(7);
```

Create a Tall Timetable Using a `matlab.io.datastore.sdidatastore`

In general, tall timetables are backed by datastores. Create a `matlab.io.datastore.sdidatastore` object to reference the signal data in the Simulation Data Inspector repository.

```
egoDs = matlab.io.datastore.sdidatastore(egoSigID);
```

Check the name of the datastore to verify you have the signal you expect.

```
egoDs.Name
```

```
ans = 
'ego'
```

Create a tall timetable from the `matlab.io.datastore.sdidatastore` to use for processing the signal data. When you have a Parallel Computing Toolbox™ license, you need to explicitly set the execution environment to the local MATLAB® session using `mapreducer` before creating the tall timetable. The `matlab.io.datastore.sdidatastore` object does not support parallel computations.

```
mapreducer(0);
```

```
egoTt = tall(egoDs)
```

```
egoTt =

  Mx1 tall timetable

         Time          Data
    ______________     ____

    0 sec               1  
    0.00056199 sec      1  
    0.0033719 sec       1  
    0.01 sec            1  
    0.02 sec            1  
    0.03 sec            1  
    0.04 sec            1  
    0.05 sec            1  
          :             :
          :             :
```

Create a Tall Timetable Using a `Simulink.sdi.Signal` Object

The `Simulink.sdi.Signal` class has a method to create a tall timetable directly, allowing you to skip the step of creating a datastore by creating it behind the scenes. Use the signal ID to access the `Simulink.sdi.Signal` object for the `ego` signal. Then, use the `getTable` method to create the tall timetable.

```
egoSig = Simulink.sdi.getSignal(egoSigID);

egoTt = egoSig.getAsTall

egoTt =

  Mx1 tall timetable

         Time          Data
    ______________    ____

    0 sec               1
    0.00056199 sec      1
    0.0033719 sec       1
    0.01 sec            1
    0.02 sec            1
    0.03 sec            1
    0.04 sec            1
    0.05 sec            1
          :             :
          :             :
```

Use the Tall Timetable to Process Your Signal Data

When you use the tall timetable egoTt, its underlying datastore reads chunks of data and passes them to the tall timetable to process. Neither the datastore nor the tall timetable retain any of the data in memory after processing. Also, the tall timetable defers processing for many operations. For example, calculate the mean value of the signal.

```
egoMean = mean(egoTt.Data)

egoMean =

  tall double

    ?
```

You can use the gather function to evaluate a variable and write its value to the workspace, or you can use the write function to write the results to disc. When you use gather, be sure the results fit into memory.

```
egoMean = gather(egoMean)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 5.9 sec
Evaluation completed in 6.2 sec

egoMean = 0.5008
```

When you perform multiple operations on a tall timetable, evaluation of the results for each step is deferred until you explicitly request the results with `write` or `gather`. During evaluation, MATLAB optimizes the number of passes it makes through the tall timetable, which can significantly speed up processing time for analyzing very large signals. For more information about working with tall arrays, see “Tall Arrays” (MATLAB).

Process Signal Data Using a `matlab.io.datastore.sdidatastore`

A `matlab.io.datastore.sdidatastore` references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the `matlab.io.datastore.sdidatastore` to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you. This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

Create a `matlab.io.datastore.sdidatastore` for a Signal

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```
runCount = Simulink.sdi.getRunCount;

latestRunID = Simulink.sdi.getRunIDByIndex(runCount);

latestRun = Simulink.sdi.getRun(latestRunID);

speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the `speed` signal.

```
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

Verify the Contents of the Datastore

Check the `Name` property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```
speedSDIds.Name
```

```
ans = 
'speed'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```
speedSDIds.preview
```

```
ans=10×1 timetable
         Time           Data
    ______________      ____

    0 sec               300 
    0.00056199 sec      300 
    0.0033719 sec       300 
    0.01 sec            300 
    0.02 sec            300 
    0.03 sec            300 
    0.04 sec            300 
    0.05 sec            300 
    0.055328 sec        300 
    0.055328 sec        300 
```

Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```
latestMax = [];

while speedSDIds.hasdata

    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);
```

```
end
```

```
latestMax
```

```
latestMax = 300
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for processing, rather than reading and processing the data incrementally with the `read` method. The `readall` method returns a `timetable` with all the signal data.

```
speedTimetable = speedSDIds.readall;
```

```
speedMax = max(speedTimetable.Data)
```

```
speedMax = 300
```

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.DatasetRef.getAsDatastore` | `Simulink.sdi.Signal` | `matlab.io.datastore.SimulationDatastore`

Topics

"Tall Arrays" (MATLAB)
"Datastore" (MATLAB)

Introduced in R2017b

matlab.io.datastore.sdidatastore.hasdata

Class: matlab.io.datastore.sdidatastore
Package: matlab.io.datastore

Determine if data is available to read

Syntax

```
tf = sdi_ds.hasdata
```

Description

`tf = sdi_ds.hasdata` returns logical `1` if the `matlab.io.datastore.sdidatastore`, `sdi_ds`, has data available to read. When `sdi_ds` does not have data available to read, `hasdata` returns `0`.

Output Arguments

`tf` — Data availability indication
logical

Logical indication of whether the `matlab.io.datastore.sdidatastore` has data available to read. When data is available, `tf` is `1`. When data is not available, `tf` is `0`.

Examples

Process Signal Data Using a `matlab.io.datastore.sdidatastore`

A `matlab.io.datastore.sdidatastore` references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the `matlab.io.datastore.sdidatastore` to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you.

This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

Create a `matlab.io.datastore.sdidatastore` for a Signal

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```
runCount = Simulink.sdi.getRunCount;

latestRunID = Simulink.sdi.getRunIDByIndex(runCount);

latestRun = Simulink.sdi.getRun(latestRunID);

speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the `speed` signal.

```
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

Verify the Contents of the Datastore

Check the `Name` property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```
speedSDIds.Name
```

```
ans = 
'speed'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```
speedSDIds.preview
```

```
ans=10×1 timetable
         Time          Data
    ______________     ____
    0 sec               300
    0.00056199 sec      300
    0.0033719 sec       300
    0.01 sec            300
    0.02 sec            300
    0.03 sec            300
    0.04 sec            300
    0.05 sec            300
    0.055328 sec        300
    0.055328 sec        300
```

Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```
latestMax = [];

while speedSDIds.hasdata

    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);

end

latestMax
```

```
latestMax = 300
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for

processing, rather than reading and processing the data incrementally with the `read` method. The `readall` method returns a `timetable` with all the signal data.

```
speedTimetable = speedSDIds.readall;
```

```
speedMax = max(speedTimetable.Data)
```

```
speedMax = 300
```

See Also

`matlab.io.datastore.sdidatastore`

Topics

"Tall Arrays" (MATLAB)
"Datastore" (MATLAB)

Introduced in R2017b

matlab.io.datastore.sdidatastore.preview

Class: matlab.io.datastore.sdidatastore
Package: matlab.io.datastore

Return preview of data in sdidatstore

Syntax

```
dataPreview = sdi_ds.preview
```

Description

`dataPreview = sdi_ds.preview` returns the first 10 samples of signal data in the `matlab.io.datastore.sdidatastore`, `sdi_ds`. The `preview` method does not change the read position. Use the `preview` method to verify that the data in your `matlab.io.datastore.sdidatastore` appears as you expect.

Output Arguments

`dataPreview` — Preview of the data
`timetable`

First 10 samples of the signal referenced by the `matlab.io.datastore.sdidatastore` in a `timetable`.

Examples

Process Signal Data Using a `matlab.io.datastore.sdidatastore`

A `matlab.io.datastore.sdidatastore` references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the `matlab.io.datastore.sdidatastore` to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you.

This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

Create a `matlab.io.datastore.sdidatastore` for a Signal

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```
runCount = Simulink.sdi.getRunCount;

latestRunID = Simulink.sdi.getRunIDByIndex(runCount);

latestRun = Simulink.sdi.getRun(latestRunID);

speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the `speed` signal.

```
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

Verify the Contents of the Datastore

Check the `Name` property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```
speedSDIds.Name
```

```
ans = 
'speed'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```
speedSDIds.preview
```

```
ans=10×1 timetable
         Time         Data
    ______________    ____
    0 sec               300
    0.00056199 sec      300
    0.0033719 sec       300
    0.01 sec            300
    0.02 sec            300
    0.03 sec            300
    0.04 sec            300
    0.05 sec            300
    0.055328 sec        300
    0.055328 sec        300
```

Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```
latestMax = [];

while speedSDIds.hasdata

    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);

end

latestMax
```

```
latestMax = 300
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for

processing, rather than reading and processing the data incrementally with the `read` method. The `readall` method returns a `timetable` with all the signal data.

```
speedTimetable = speedSDIds.readall;
```

```
speedMax = max(speedTimetable.Data)
```

```
speedMax = 300
```

See Also

`matlab.io.datastore.sdidatastore`

Topics

“Timetables” (MATLAB)

Introduced in R2017b

matlab.io.datastore.simulationdatastore.sdidatastore.read

Read a chunk of data from an sdidatastore

Syntax

```
data = sdi_ds.read
```

Description

`data = sdi_ds.read` reads a chunk of samples from the `matlab.io.datastore.sdidatastore`, `sdi_ds`, and updates the read position for `sdi_ds` to the point following the endpoint of the returned data. The samples are returned in the `timetable`, `data`. The number of samples read by the `read` method vary, and the returned `timetable` always fits into memory. Use the `read` method to incrementally process signals that are too large to fit into memory.

Output Arguments

data — Chunk of data read from sdidatastore
`timetable`

Chunk of samples read from the `matlab.io.datastore.simulationdatastore`, returned as a `timetable`.

Examples

Process Signal Data Using a `matlab.io.datastore.sdidatastore`

A `matlab.io.datastore.sdidatastore` references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the `matlab.io.datastore.sdidatastore` to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you. This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

Create a matlab.io.datastore.sdidatastore for a Signal

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```
runCount = Simulink.sdi.getRunCount;

latestRunID = Simulink.sdi.getRunIDByIndex(runCount);

latestRun = Simulink.sdi.getRun(latestRunID);

speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the `speed` signal.

```
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

Verify the Contents of the Datastore

Check the `Name` property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```
speedSDIds.Name
```

```
ans = 
'speed'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```
speedSDIds.preview
```

```
ans=10×1 timetable
         Time          Data
    ______________     ____
    0 sec                 300
    0.00056199 sec        300
    0.0033719 sec         300
    0.01 sec              300
    0.02 sec              300
    0.03 sec              300
    0.04 sec              300
    0.05 sec              300
    0.055328 sec          300
    0.055328 sec          300
```

Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```
latestMax = [];

while speedSDIds.hasdata

    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);

end

latestMax
```

```
latestMax = 300
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for

processing, rather than reading and processing the data incrementally with the `read` method. The `readall` method returns a `timetable` with all the signal data.

```
speedTimetable = speedSDIds.readall;
```

```
speedMax = max(speedTimetable.Data)
```

```
speedMax = 300
```

Alternatives

You can use your `matlab.io.datastore.sdidatastore` to create a tall timetable to process signals too large to fit into memory. The tall timetable handles loading and processing the chunks of signal data for you. The `matlab.io.datastore.sdidatastore` reference page includes an example that shows how to process your data using a tall timetable. For more information about working with tall timetables, see “Tall Arrays” (MATLAB).

See Also

`matlab.io.datastore.sdidatastore`

Topics

“Datastore” (MATLAB)

Introduced in R2017b

matlab.io.datastore.sdidatastore.readall

Class: matlab.io.datastore.sdidatastore
Package: matlab.io.datastore

Read all data from an sdidatastore

Syntax

```
data = sdi_ds.readall
```

Description

`data = sdi_ds.readall` reads all the data in the `matlab.io.datastore.sdidatastore`, `sdi_ds`, into memory, returning the `timetable`, `data`. Use `readall` only when the signal referenced by the `matlab.io.datastore.sdidatastore` fits into memory.

Output Arguments

`data` — timetable of data
`timetable`

All the data in the `matlab.io.datastore.sdidatastore`, returned in a `timetable`.

Examples

Process Signal Data Using a `matlab.io.datastore.sdidatastore`

A `matlab.io.datastore.sdidatastore` references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the `matlab.io.datastore.sdidatastore` to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you.

This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

Create a `matlab.io.datastore.sdidatastore` for a Signal

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```
runCount = Simulink.sdi.getRunCount;

latestRunID = Simulink.sdi.getRunIDByIndex(runCount);

latestRun = Simulink.sdi.getRun(latestRunID);

speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the `speed` signal.

```
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

Verify the Contents of the Datastore

Check the `Name` property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```
speedSDIds.Name
```

```
ans = 
'speed'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```
speedSDIds.preview
```

```
ans=10×1 timetable
         Time          Data
    ______________    ____
    0 sec                 300
    0.00056199 sec        300
    0.0033719 sec         300
    0.01 sec              300
    0.02 sec              300
    0.03 sec              300
    0.04 sec              300
    0.05 sec              300
    0.055328 sec          300
    0.055328 sec          300
```

Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```
latestMax = [];

while speedSDIds.hasdata

    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);

end

latestMax
```

```
latestMax = 300
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for

processing, rather than reading and processing the data incrementally with the `read` method. The `readall` method returns a `timetable` with all the signal data.

```
speedTimetable = speedSDIds.readall;

speedMax = max(speedTimetable.Data)

speedMax = 300
```

Alternatives

When your signals fit into memory, you can use other classes and functions of the Simulation Data Inspector programmatic interface, like the `Simulink.sdi.Signal` class, to access and process simulation data.

See Also

`matlab.io.datastore.sdidatastore`

Topics

“Datastore” (MATLAB)

Introduced in R2017b

matlab.io.datastore.sdidatastore.reset

Class: matlab.io.datastore.sdidatastore
Package: matlab.io.datastore

Reset the read position

Syntax

```
sdi_ds.reset
```

Description

`sdi_ds.reset` resets the read position for the `matlab.io.datastore.sdidatastore`, `sdi_ds`, to the beginning.

Examples

Process Signal Data Using a `matlab.io.datastore.sdidatastore`

A `matlab.io.datastore.sdidatastore` references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the `matlab.io.datastore.sdidatastore` to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you. This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

Create a `matlab.io.datastore.sdidatastore` for a Signal

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```
runCount = Simulink.sdi.getRunCount;

latestRunID = Simulink.sdi.getRunIDByIndex(runCount);

latestRun = Simulink.sdi.getRun(latestRunID);

speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the `speed` signal.

```
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

Verify the Contents of the Datastore

Check the `Name` property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```
speedSDIds.Name
```

```
ans = 
'speed'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```
speedSDIds.preview
```

```
ans=10×1 timetable
         Time          Data
    ______________     ____

    0 sec              300 
    0.00056199 sec     300 
    0.0033719 sec      300 
    0.01 sec           300 
    0.02 sec           300 
    0.03 sec           300 
    0.04 sec           300 
    0.05 sec           300 
    0.055328 sec       300 
    0.055328 sec       300 
```

Process Signal Data with the matlab.io.datastore.sdidatastore

When your signal is too large to fit into memory, you can use the readData method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the hasdata method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```
latestMax = [];

while speedSDIds.hasdata

    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);

end

latestMax
```

```
latestMax = 300
```

On each read operation, the read method updates the read position for the start of the next read operation. After reading some or all of the matlab.io.datastore.sdidatastore, you can reset the read position to start again from the beginning of the signal.

```
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your matlab.io.datastore.sdidatastore fits into memory, you can use the readall method to read all the signal data into memory for processing, rather than reading and processing the data incrementally with the read method. The readall method returns a timetable with all the signal data.

```
speedTimetable = speedSDIds.readall;

speedMax = max(speedTimetable.Data)
```

```
speedMax = 300
```

See Also

matlab.io.datastore.sdidatastore

Topics

“Datastore” (MATLAB)

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 `tall` 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

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).

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 MATLAB tall variable as simulation input data.

Examples

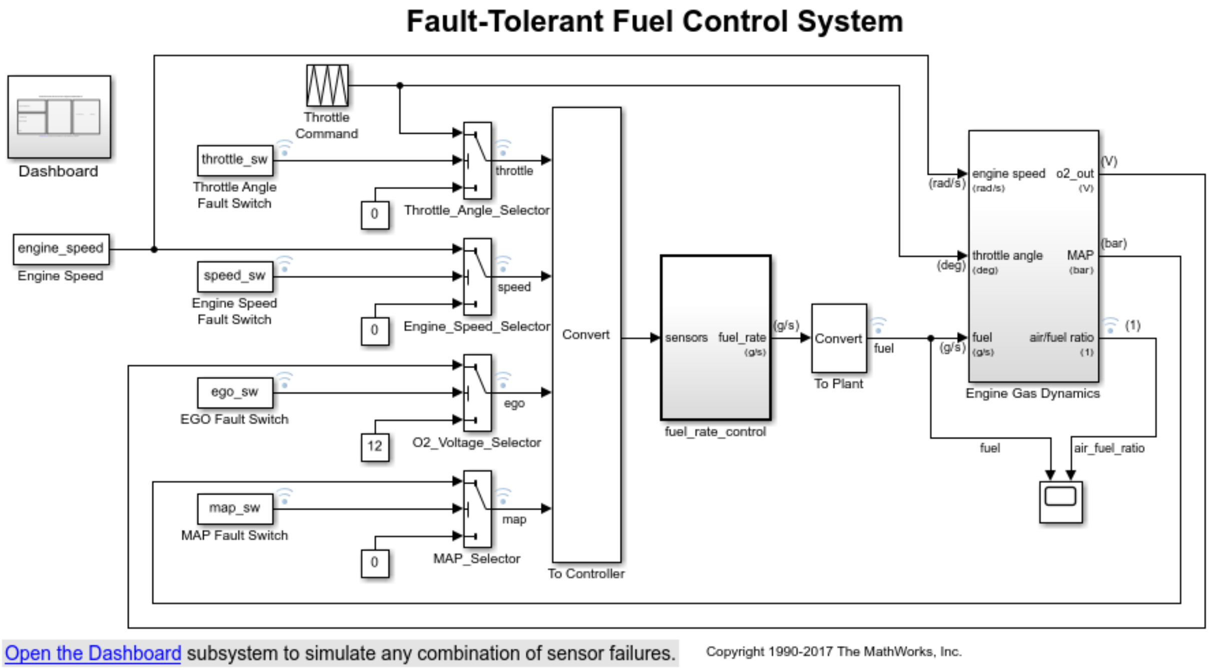
Inspect and Analyze Data in Simulation Datastore

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 1990-2017 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset data 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
```

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

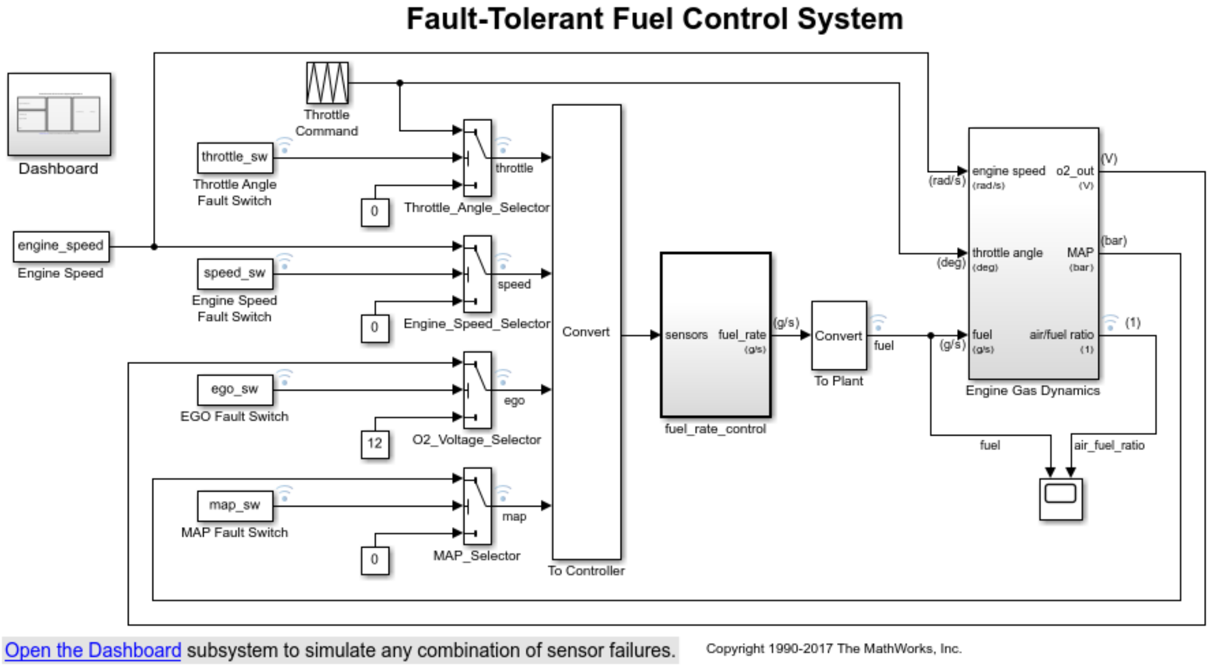
Inspect and Analyze Data in Simulation Datastore

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 1990-2017 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset data 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
```

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

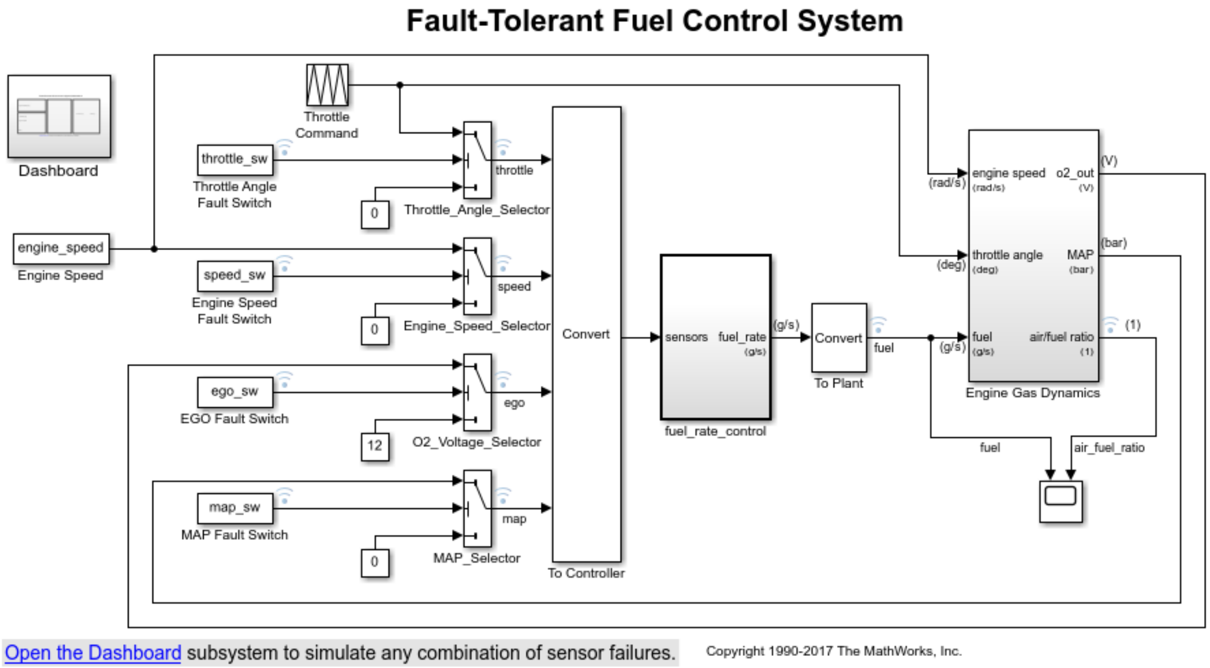
Inspect and Analyze Data in Simulation Datastore

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 1990-2017 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset data 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
```

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

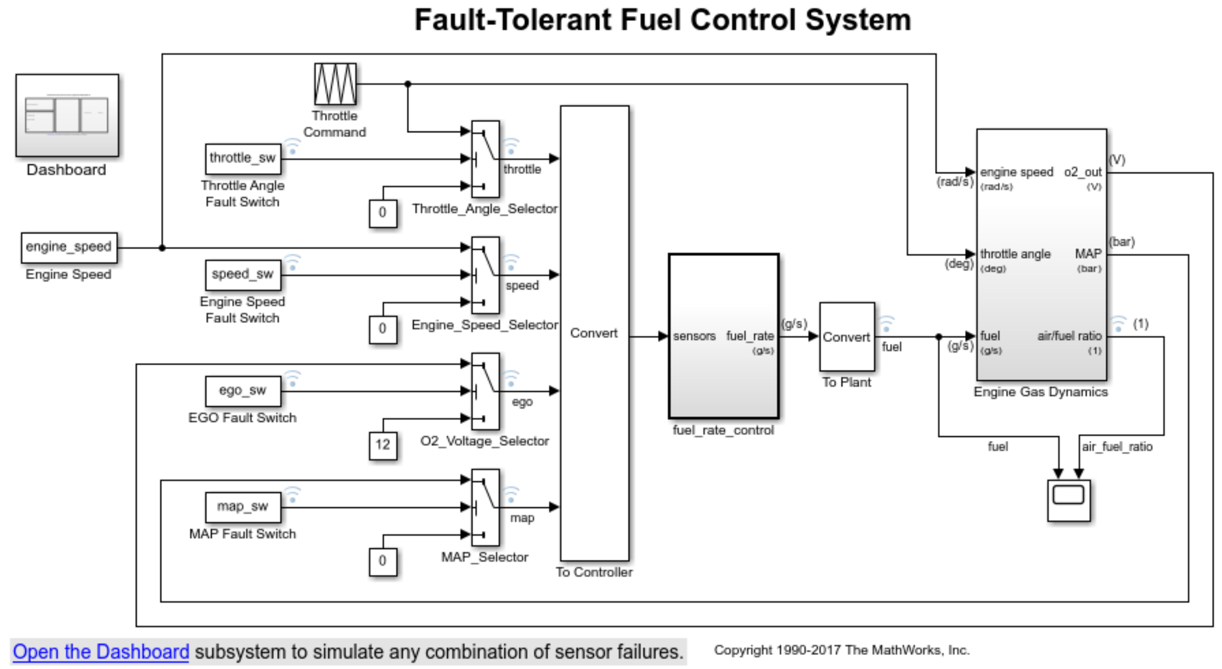
Inspect and Analyze Data in Simulation Datastore

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 1990-2017 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset data 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
```

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

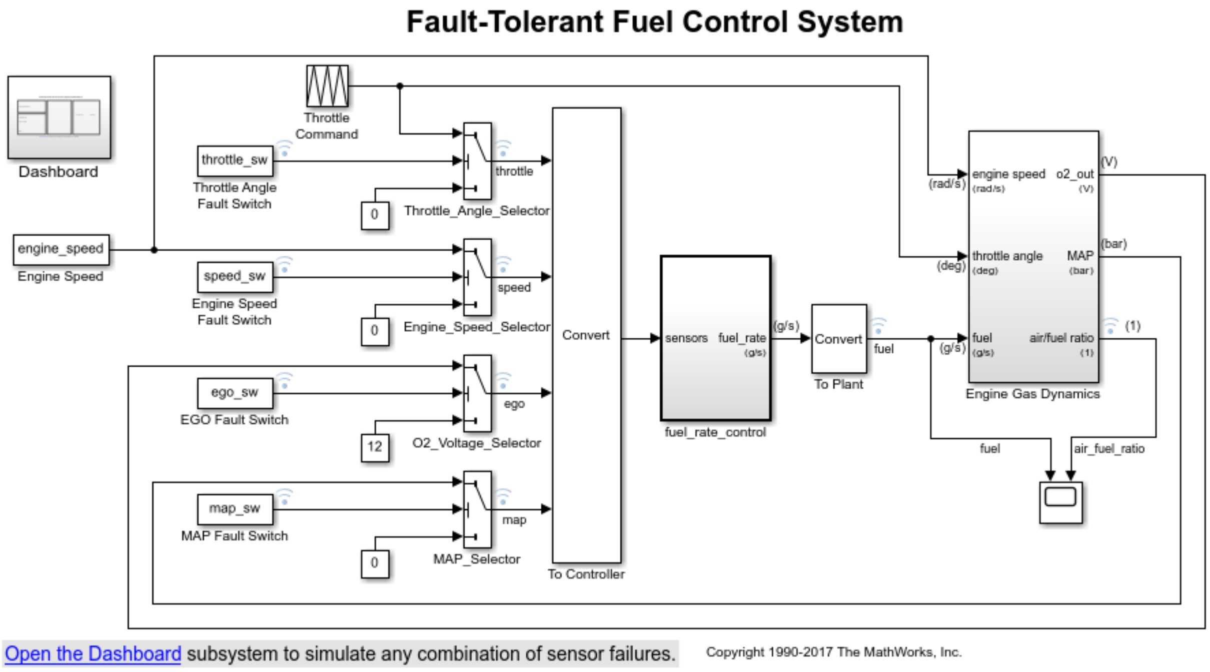
Inspect and Analyze Data in Simulation Datastore

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 1990-2017 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset data 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
```

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 5-75	Type of method execution event that occurred.
“Source” on page 5-75	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. .	`lb = LibraryBrowser.LibraryBrowser2;`
		`refresh(lb)`

`show`	Display the library browser.	`lb = LibraryBrowser.LibraryBrowser2;`
		`show(lb)`

`hide`	Hide the library browser.	`lb = LibraryBrowser.LibraryBrowser2;`
		`hide(lb)`

`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.	`lb = LibraryBrowser.LibraryBrowser2;`
		`getPosition(lb)`
		
		`ans =`
		
		`    50   279   600   600`

`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.	`lb = LibraryBrowser.LibraryBrowser2;`
		`setPosition(lb,[70 250 500 500])`

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	**Default:** `false` — Model reference discrete sample time inheritance is not allowed. If your System object uses discrete sample time in its algorithm, override the default by returning `true` from `allowModelReferenceDiscreteSampleTimeInheritanceImpl`.
One or more inputs	**Default:** `true` — If no other Simulink constraint prevents it, model reference sample time inheritance is allowed. If your System object does not use sample time in its algorithm, override the default by returning `false` from `allowModelReferenceDiscreteSampleTimeInheritanceImpl`.

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
```

See Also

`matlab.System`

Topics

“Set Model Reference Discrete Sample Time Inheritance”
“Model Reference Basics”
“Referenced Model Sample Times”

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
```

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
```

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

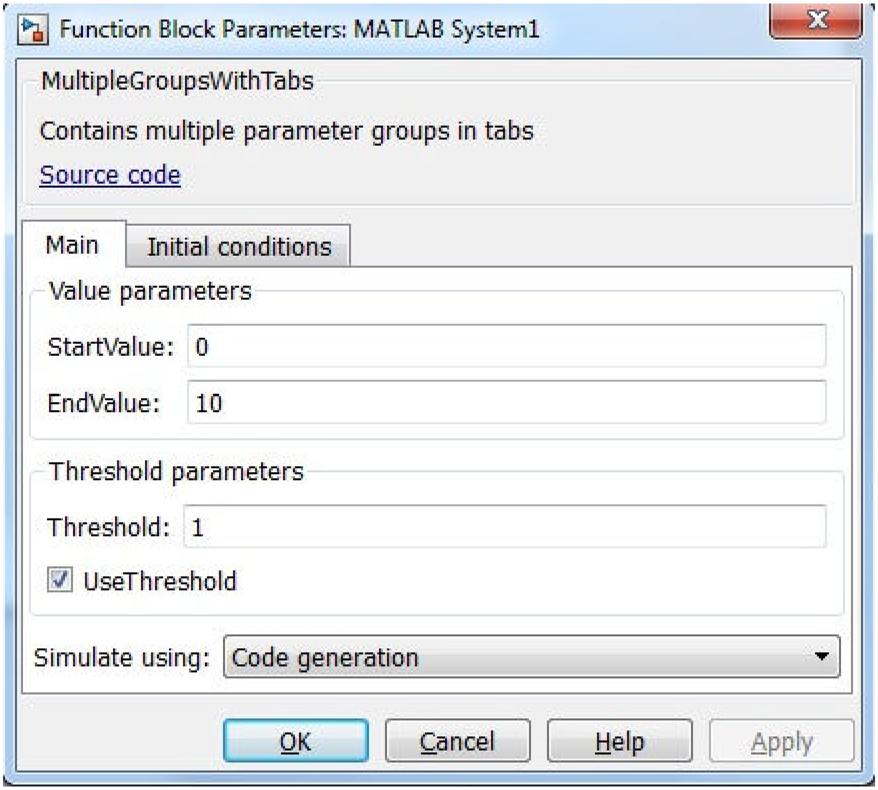
Examples

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.



Function Block Parameters: MATLAB System1

MultipleGroupsWithTabs

Contains multiple parameter groups in tabs

Source code

Value parameters

StartValue: 0

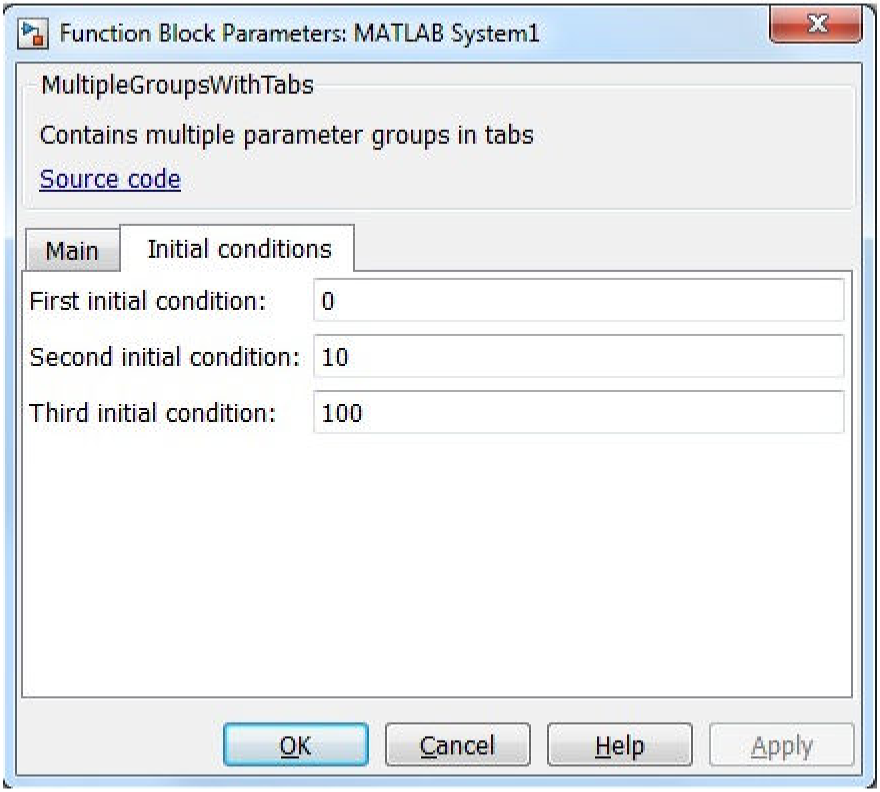
EndValue: 10

Threshold parameters

Threshold: 1

☑ UseThreshold

Simulate using: Code generation



Function Block Parameters: MATLAB System1

MultipleGroupsWithTabs

Contains multiple parameter groups in tabs

Source code

First initial condition: 0

Second initial condition: 10

Third initial condition: 100

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
```

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`

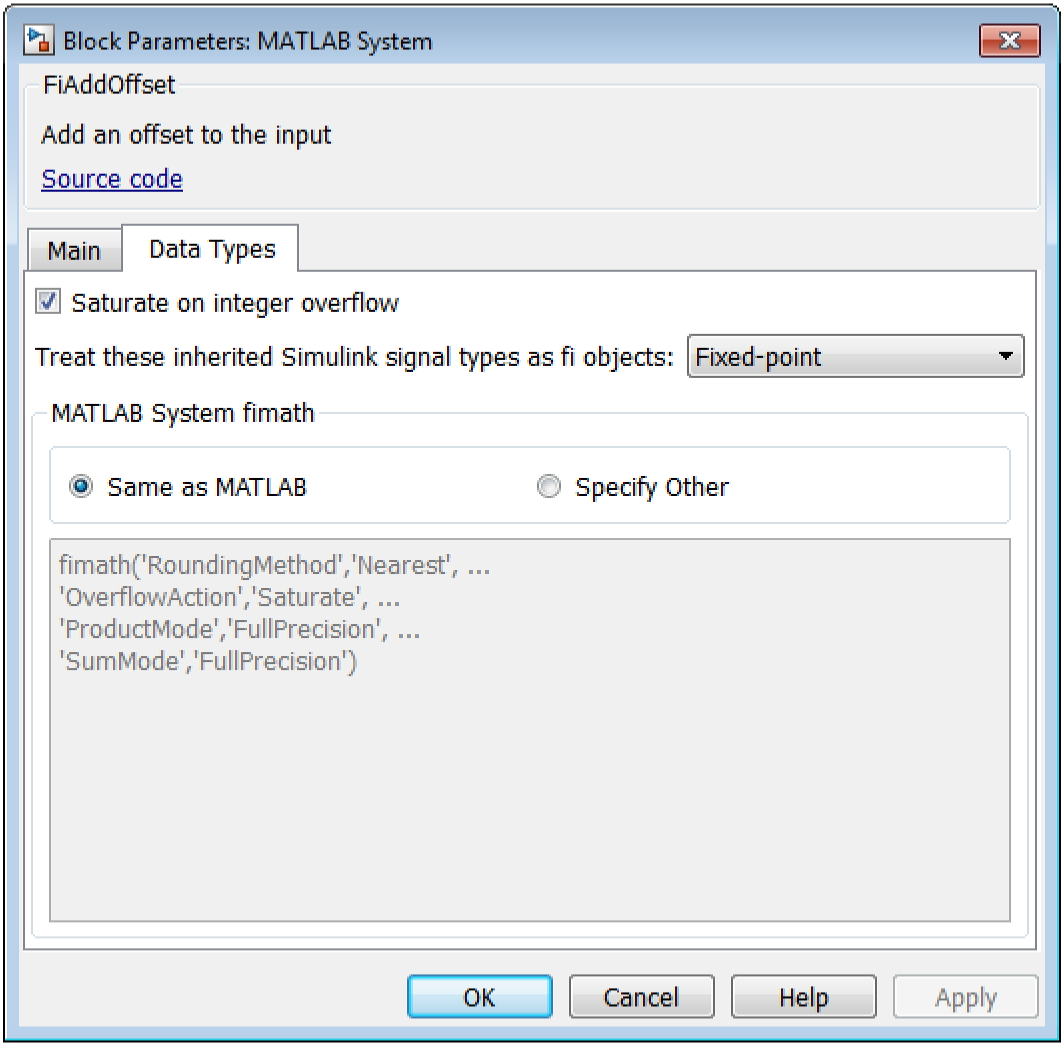
Examples

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.



FiAddOffset

Add an offset to the input

Source code

☑ Saturate on integer overflow

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB System fimath

● Same as MATLAB ○ Specify Other

```
fimath('RoundingMethod','Nearest', ...
'OverflowAction','Saturate', ...
'ProductMode','FullPrecision', ...
'SumMode','FullPrecision')
```

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`

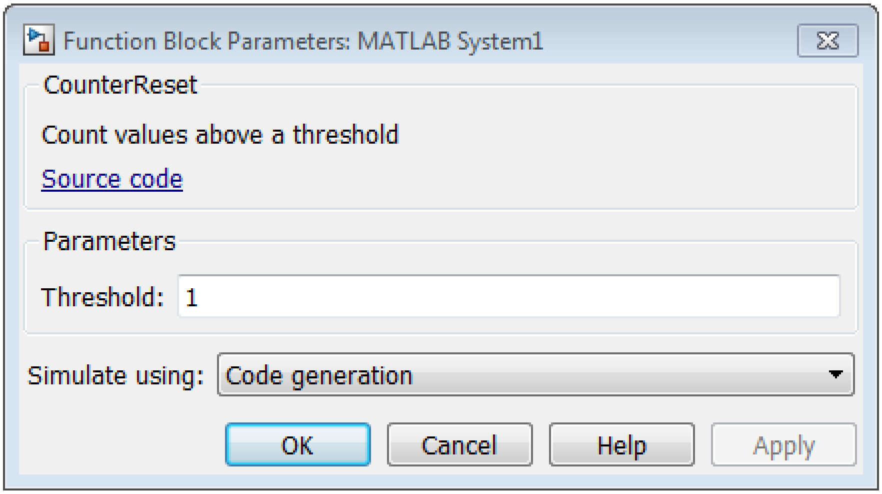
Examples

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.



CounterReset

Count values above a threshold

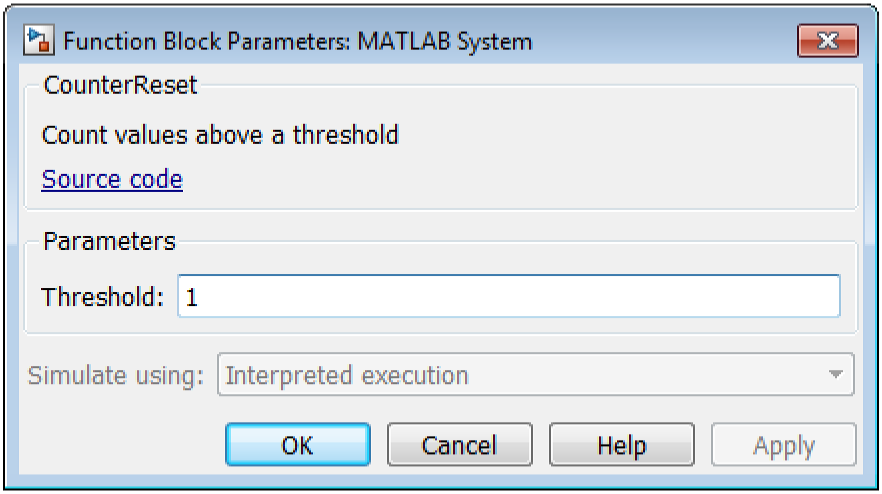
Source code

Parameters

Threshold: 1

Simulate using: Code generation

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.


Function Block Parameters: MATLAB System

CounterReset

Count values above a threshold

Source code

Parameters

Threshold: 1

Simulate using: Interpreted execution

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
```

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
```

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
```

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
```

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 5-106 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

setProperties | validatePropertiesImpl

Topics

“Process Tuned Properties” (MATLAB)
“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

getIconImpl	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.

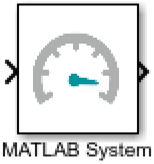
```
methods (Access = protected)
   function icon = getIconImpl(~)
      icon = {'Enhanced','Counter'};
   end
end
```

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.



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.

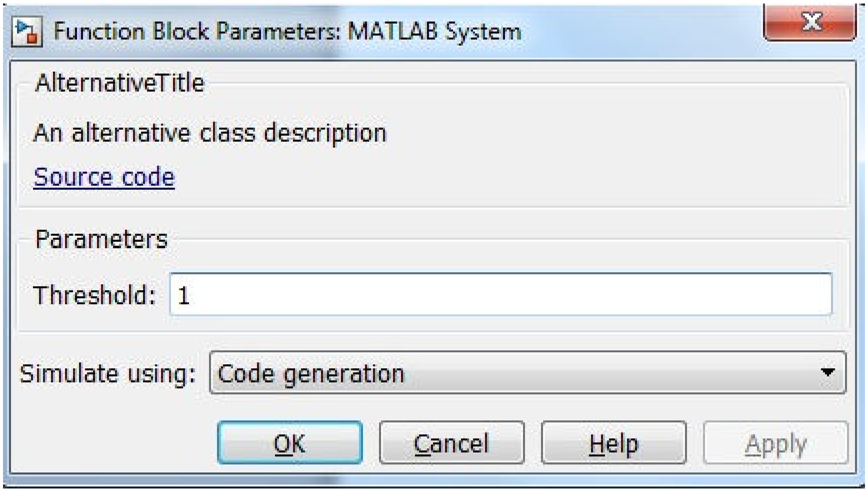
Examples

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`.



Function Block Parameters: MATLAB System

AlternativeTitle

An alternative class description

Source code

Parameters

Threshold: 1

Simulate using: Code generation

See Also

`getHeaderImpl` | `matlab.system.display.Section` | `matlab.system.display.SectionGroup`

Topics

"Creating Classes" (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

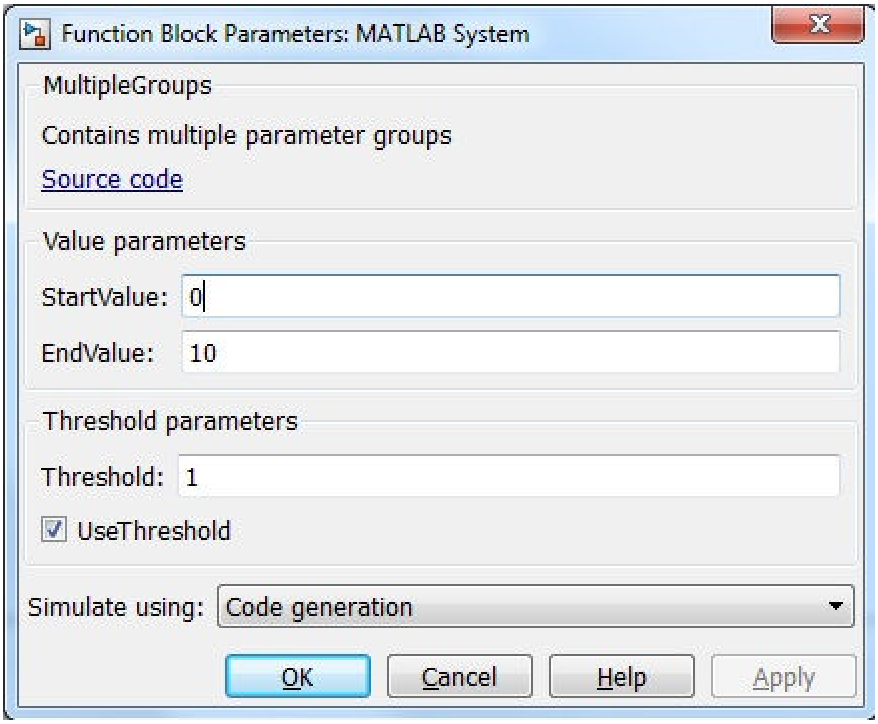
Examples

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.



Function Block Parameters: MATLAB System

MultipleGroups

Contains multiple parameter groups

Source code

Value parameters

StartValue: 0

EndValue: 10

Threshold parameters

Threshold: 1

☑ UseThreshold

Simulate using: Code generation

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 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

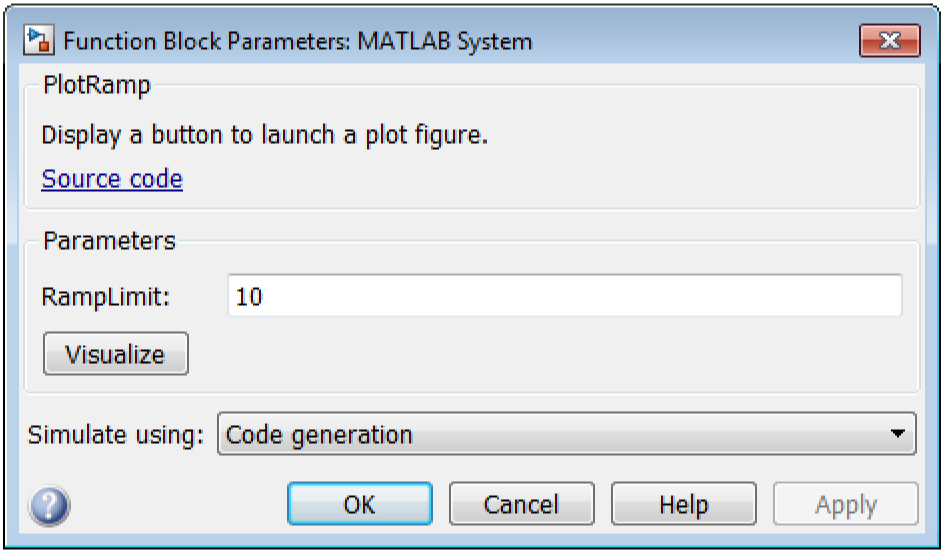
Define Button on MATLAB System Block

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.



Function Block Parameters: MATLAB System

PlotRamp

Display a button to launch a plot figure.

Source code

Parameters

RampLimit: 10

Simulate using: Code generation

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
```

See Also

matlab.System.getPropertyGroupsImpl | matlab.system.display.Section | matlab.system.display.SectionGroup

Topics

"Creating Classes" (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.

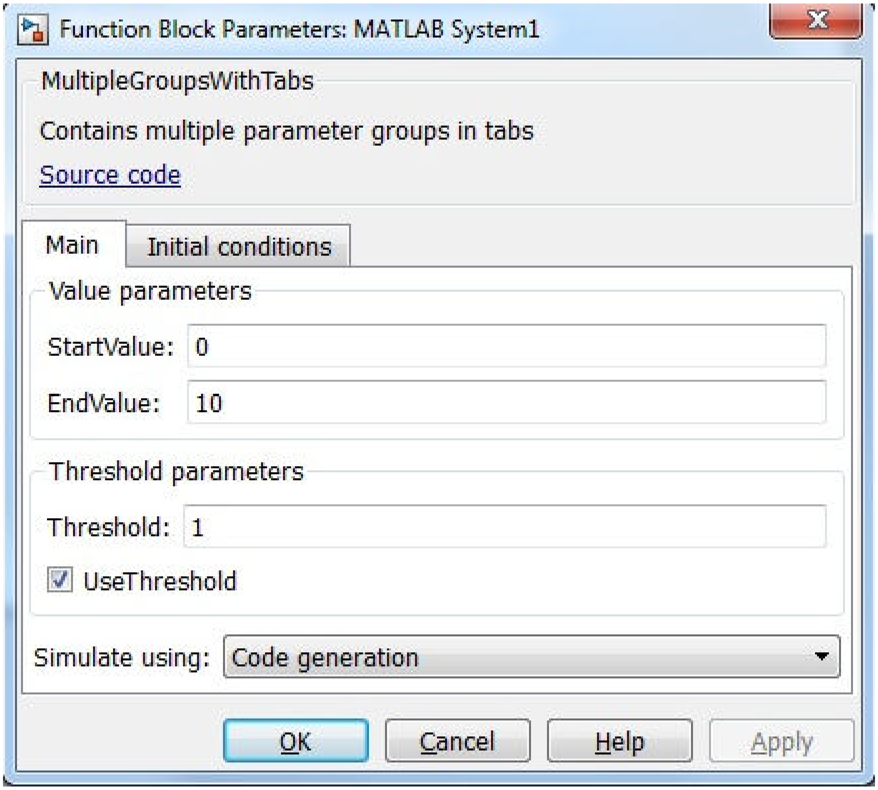
Examples

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.



Function Block Parameters: MATLAB System1

MultipleGroupsWithTabs

Contains multiple parameter groups in tabs

Source code

Main | Initial conditions

Value parameters

StartValue: 0

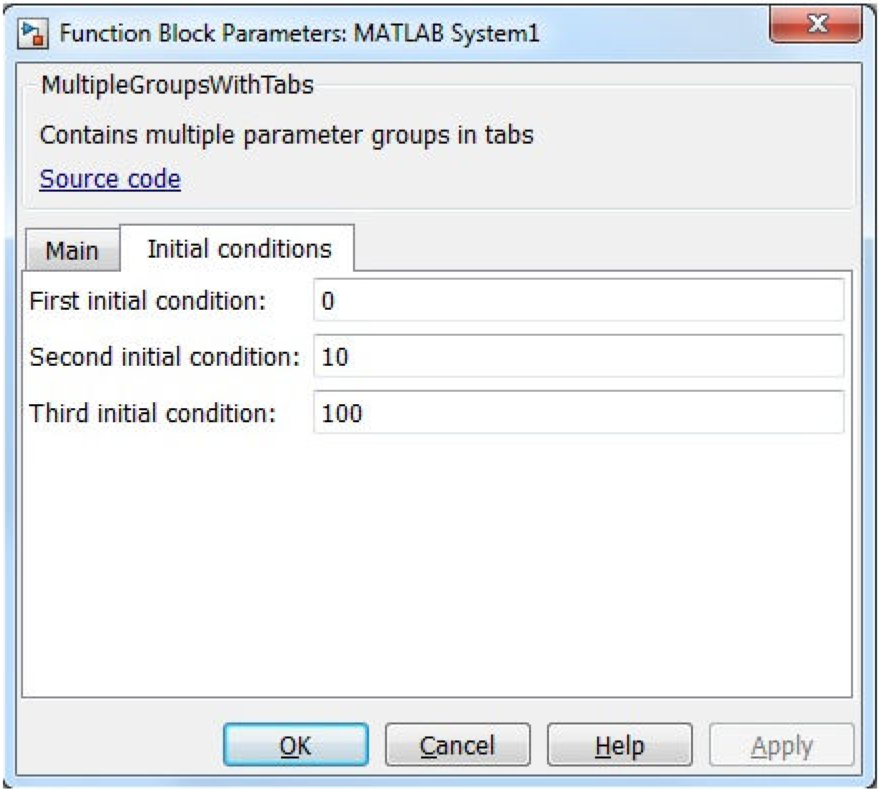
EndValue: 10

Threshold parameters

Threshold: 1

☑ UseThreshold

Simulate using: Code generation



MultipleGroupsWithTabs

Contains multiple parameter groups in tabs

Source code

First initial condition: 0

Second initial condition: 10

Third initial condition: 100

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"`

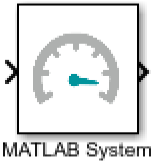
Examples

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.



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

getDiscreteStateSpecificationImpl	Discrete state size, data type, and complexity
getOutputDataTypeImpl	Data types of output ports
getOutputSizeImpl	Sizes of output ports
isOutputComplexImpl	Complexity of output ports
isOutputFixedSizeImpl	Fixed- or variable-size output ports
propagatedInputComplexity	Complexity of input during Simulink propagation
propagatedInputDataType	Data type of input during Simulink propagation
propagatedInputFixedSize	Fixed-size status of input during Simulink propagation
propagatedInputSize	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"
"Set Fixed- or Variable-Size Output"
"Set 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.

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:

- `true` = complex
- `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
```

See Also

`matlab.system.mixin.Propagates`

Topics

“Set 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 numerictype 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
```

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 `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.

In this method, you cannot modify any properties.

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
```

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
```

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
```

See Also

`matlab.system.mixin.Propagates` | `propagatedInputFixedSize`

Topics

"Set Fixed- or Variable-Size Output"

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
```

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 numerictype 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
```

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
```

See Also

`isOutputFixedSizeImpl` | `matlab.system.mixin.Propagates`

Topics

"Set Fixed- or Variable-Size Output"

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
```

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

isInputDirectFeedthroughImpl	Direct feedthrough status of input
outputImpl	Output calculation from input or internal state of System object
updateImpl	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)
[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,
input,input2, ...)
```

Description

`[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj)` specifies whether each input is a direct feedthrough input. If direct feedthrough is `true`, the output depends on the input at each time instant.

`[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj, input,input2, ...)` uses one or more of the System object input specifications to determine whether inputs have direct feedthrough.

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.

Class Information

This method is part of the `matlab.system.mixin.Nondirect` class.

Run-Time Details

`isInputDirectFeedthroughImpl` is called by the MATLAB System block.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify, implement, or access tunable properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object.

`input1,input2,...` — Inputs to the System object
inputs to the System object algorithm

Inputs to the algorithm (`stepImpl`) of the System object. The inputs list must match the order of inputs in the `stepImpl` signature.

Output Arguments

`flag1,...,flagN` — Output flag for each input to the System object
logical

Logical value, either `true` or `false` indicating whether the input is direct feedthrough. The number of output flags must match the number of inputs to the System object (inputs to `stepImpl`, `outputImpl`, or `updateImpl`).

Examples

Specify Input as Nondirect Feedthrough

Use `isInputDirectFeedthroughImpl` in your class definition file for marking all inputs as nondirect feedthrough.

```
methods (Access = protected)
   function flag = isInputDirectFeedthroughImpl(~)
      flag = false;
   end
end
```

Complete Class Definition

```
classdef intDelaySysObj < matlab.System &...
     matlab.system.mixin.Nondirect
   % intDelaySysObj Delay input by specified number of samples.

   properties
      InitialOutput = 0;
   end
   properties (Nontunable)
      NumDelays = 1;
   end
   properties (DiscreteState)
      PreviousInput;
   end

   methods (Access = protected)
      function validatePropertiesImpl(obj)
         if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
            error('Number of delays must be > 0 scalar value.');
         end
         if (numel(obj.InitialOutput)>1)
            error('Initial Output must be scalar value.');
         end
      end

      function setupImpl(obj)
         obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
      end

      function resetImpl(obj)
         obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
      end

      function [y] = outputImpl(obj,~)
         y = obj.PreviousInput(end);
      end
      function updateImpl(obj, u)
         obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
      end
      function flag = isInputDirectFeedthroughImpl(~)
         flag = false;
      end
   end
end
```

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
```

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,input1,input2,...)
```

Description

`updateImpl(obj,input1,input2,...)` specifies the algorithm to update the System object states. You implement this method when your algorithm outputs depend only on the object's internal state and internal properties.

Run-Time Details

`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.

Method Authoring Tips

- Do not use this method to update the outputs from the inputs.
- You must set `Access = protected` for this method.
- If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.

Input Arguments

obj — System object
System object

System object handle used to access properties, states, and methods specific to the object.

input1,input2,... — Inputs to the System object
inputs to the System object

List the inputs to the System object. The order of inputs must match the order of inputs defined in the stepImpl 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
```

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
- Customize the sample time with offsets and tick times
- 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

getSampleTimeImpl	Specify sample time type, offset time, and sample time
getSampleTime	Query sample time
getCurrentTime	Current simulation time in MATLAB System block
createSampleTime	Create sample time specification object
setNumTicksUntilNextHit	Set the number of ticks in Simulink sample time

See Also

Classes
`matlab.System`

`createSampleTime` | `getCurrentTime` | `getSampleTime` | `getSampleTimeImpl`

Blocks
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 Sample Time

Specify a discrete sample time for the MATLAB System block.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'Type','Discrete',...
      'SampleTime',10.2,'OffsetTime',0.5);
end
```

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”.

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.

Note If the MATLAB System block is operating in continuous sample time, `getCurrentTime` may return non-monotonic times due to solver operation.

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”.

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,'Type',Type,Name,Value)
```

Description

`sts = createSampleTime(obj)` creates a sample time specification object for inherited sample time for the System object `obj`. Use this sample time specification 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 specification object with the specified sample time type.

`sts = createSampleTime(obj,'Type',Type,Name,Value)` creates a sample time specification 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Type','Fixed In Minor Step'`

`Type` — Sample time type

`'Inherited'` (default) | `'Controllable'` | `'Discrete'` | `'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”
- `'Controllable'` - “Controllable Sample Time”
- `'Discrete'` - “Discrete Sample Time”
- `'Fixed In Minor Step'` - “Fixed-in-Minor-Step”

Example: `createSampleTime('Type','Fixed In Minor Step')`

`Disallow` — Disallow controllable sample time

`'Controllable'`

Optional for `Inherited` sample time only.

When the sample time type is set to `Inherited`, this option disallows inherited controllable sample time. Use this option if your System object depends on having constant time between each sample-time hit.

If controllable sample time is propagated to the System object, discrete sample time is used instead with the same tick time as the propagated controllable sample time.

Example:
`createSampleTime('Type','Inherited','Disallow','Controllable')`

`SampleTime` — Time between samples

`1` (default) | positive number

For `Discrete` sample time only.

Specify the time between sample hits in Simulink.

Data Types: `single` | `double`

Example: `createSampleTime('Type','Discrete','SampleTime',1)`

`OffsetTime` — Offset from sample time

`0` (default) | nonnegative number less than `SampleTime`

For `Discrete` 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`

Example: `createSampleTime('Type','Discrete','SampleTime',2,'OffsetTime',1)`

`TickTime` — Time between sample time hits

-1 (default) | positive scalar

Required for `Controllable` sample time only.

Specify the time between controllable sample time hits. The tick time must be a positive scalar.

Data Types: `single` | `double`

Example:
`createSampleTime('Type','Controllable','TickTime',obj.TickTime)`

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 Specification Objects

Specify that the MATLAB System block inherits 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 Sample Time Object

Specify a discrete sample time specification for the MATLAB System block, including offset time and the sample time.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'Type','Discrete',...
      'SampleTime',10.2,'OffsetTime',0.5);
end
```

See Also

`getCurrentTime` | `getSampleTime` | `getSampleTimeImpl` | `matlab.system.mixin.SampleTime` | `setNumTicksUntilNextHit`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b

setNumTicksUntilNextHit

Class: matlab.system.mixin.SampleTime

Set the number of ticks in Simulink sample time

Syntax

```
setNumTicksUntilNextHit(obj,ticks)
```

Description

`setNumTicksUntilNextHit(obj,ticks)` sets the number of ticks in Simulink sample time to wait until the next call to `stepImpl`, or `outputImpl`/`updateImpl`. To use this method, set your System object to controllable sample time with `createSampleTime('Type','Controllable')`. Otherwise, your System object gives a compilation error.

You can only call this method from `stepImpl`, `outputImpl`, `updateImpl`, or `resetImpl`.

Input Arguments

`obj` — System object
system object

System object that you want to specify the sample time.

`ticks` — Number of ticks in Simulink sample time
positive integer scalar

Number of ticks in Simulink sample time to wait until the next call to `stepImpl` or `outputImpl`/`updateImpl`. Specify this number as a positive integer scalar less than `intmax('uint64')`.

The number of sample time ticks to wait until the next hit is persistent. If you don’t update this number, Simulink uses the previously set value of number of ticks to wait.

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTime` | `getSampleTimeImpl` | `matlab.system.mixin.SampleTime`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2018a

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) | `true`

Display of the **Code Generation Advisor**, **Upgrade Advisor**, and **Performance Advisor** in the Model Advisor window. 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 form 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

Use a `Simulink.AliasType` to create an alias of a built-in data type such as `int8`.

The name of the object is the alias. The data type to which an alias refers, such as `int8`, is the base type.

You create the object in the base workspace or a data dictionary. To use the alias, you use the name of the object to set data types for signals, states, and parameters in a model.

Using aliases to specify signal and parameter data types can greatly simplify global changes to the data types that a model specifies. In particular, changing the data type of all signals, states, 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, states, 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 code that you generate from a model (Simulink Coder) uses the alias only if you use an ERT-based system target file (Embedded Coder).

Alternatively, to define and name a numeric data type, you can use an object of the class `Simulink.NumericType`.

Creation

You can use either the Model Explorer or MATLAB commands to create a data type alias.

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).

To generate `Simulink.AliasType` objects that correspond to `typedef` statements in your external C code, consider using the `Simulink.importExternalCTypes` function.

To create a data type alias programmatically, use the `Simulink.AliasType` function described below.

Syntax

```
aliasObj = Simulink.AliasType
aliasObj = Simulink.AliasType(baseType)
```

Description

`aliasObj = Simulink.AliasType` returns a `Simulink.AliasType` object with default property values.

`aliasObj = Simulink.AliasType(baseType)` returns a `Simulink.AliasType` object and initializes the value of the `BaseType` property by using `baseType`.

Properties

For information about properties in the property dialog box of a `Simulink.AliasType` object, see “Simulink.AliasType Property Dialog Box”.

BaseType — Name of base data type

`'double'` (default) | character vector

Name of the base data type that this alias renames, specified as a character vector. You can specify the name of a standard data type, such as `int8`, or the name of a custom data type, such as the name of another `Simulink.AliasType` object or the name of an enumeration.

To specify a fixed-point data type, you can use a call to the `fixdt` function, such as `'fixdt(0,16,7)'`.

You can, with one exception, specify a nonstandard data type, e.g., a data type defined by a `Simulink.NumericType` object, by specifying the data type name. The exception is a `Simulink.NumericType` whose `DataTypeMode` 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.

Corresponds to **Base type** in the property dialog box.

Example: `'int8'`

Example: `'myOtherAlias'`

Data Types: `char`

DataScope — Specification to generate or import type definition in the generated code

`'Auto'` (default) | `'Exported'` | `'Imported'`

Specification to generate or import the type definition (`typedef`) in the generated code (Simulink Coder), specified as `'Auto'`, `'Exported`, or `'Imported'`.

The table shows the effect of each option.

Value	Action
`'Auto'` (default)	If no value is specified for `HeaderFile`, export the type definition to *`model`*`_types.h`, where *`model`* 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 `rtwtypes.h`. If a value is specified for `HeaderFile`, 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 `HeaderFile` property. If no value is specified for `HeaderFile`, the header file name defaults to *`type`*`.h`. *`type`* is the data type name.
`'Imported'`	Import the data type definition from a header file, which can be specified in the `HeaderFile` property. If no value is specified for `HeaderFile`, the header file name defaults to *`type`*`.h`. *`type`* is the data type name.

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to **Data scope** in the property dialog box.

`Description` — Custom description of data type alias

`''` (empty character vector) (default) | character vector

Custom description of the data type alias, specified as a character vector.

Corresponds to **Description** in the property dialog box.

Example: `'This type alias corresponds to a floating-point implementation.'`

Data Types: `char`

`HeaderFile` — Name of header file that contains type definition in the generated code

`''` (empty character vector) (default) | character vector

Name of the header file that contains the type definition (`typedef`) in the generated code, specified as a character vector.

If this property is specified, the specified name is used during code generation for importing or exporting. If this property is empty, the value defaults to *type*.h if DataScope equals 'Imported' or 'Exported', or defaults to *model*_types.h if DataScope equals 'Auto'.

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>'.

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to **Header file** in the property dialog box.

Example: 'myHdr.h'

Example: 'myHdr'

Example: 'myHdr.hpp'

Data Types: char

Examples

Create Alias for Enumerated Data Type

To create an alias for an enumerated type called SlDemoSign:

```
myEnumAlias = Simulink.AliasType('Enum: 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. The value of BaseType must be specified as 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)';
```

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
`AnnotationType`	Type of annotation—text (note), area, or image. This property is read only.	`note_annotation` | `area_annotation` | `image_annotation`

Property	Description	Values
`BackgroundColor`	Background color of this annotation.	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. Annotation background color can also be `'black'`, `'white'`, `'red'`, `'green'`, `'blue'`, `'cyan'`, `'magenta'`, `'yellow'`, `'gray'`, `'lightBlue'`, `'orange'`, `'darkGreen'`.
`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.	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. Annotation background color can also be 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.
Handle	Annotation handle.	real
HiliteAncestors	For internal use.	
Horizontal-Alignment	Horizontal alignment of this annotation.	'center' | {'left'}|'right'

Property	Description	Values
`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'`
`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'}`

Property	Description	Values
Text	Text of annotation. Same as Name.	character vector
Type	Annotation type. This is always 'annotation'.	'annotation'
UseDisplayText-AsClickCallback	Specifies whether to use the contents of the Text property as the click function for this annotation. 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. See “Associate a Click Function with an Annotation” for more information.	'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.

Method	Description
`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 5-438 method of a block's run-time object after the model containing the block has been compiled.

Parent

`Simulink.BlockData`

Children

None

Property Summary

Name	Description
“Usage” on page 5-193	Usage type of this DWork vector.
“UsedAsDiscState” on page 5-193	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.BlockCompInputPortData

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 5-439 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 5-194	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.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 5-440 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 5-217	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 5-199	Fundamental base data type.
"AliasedThroughDataTypeID" on page 5-200	Fundamental base data type ID.
"Complexity" on page 5-200	Numeric type (real or complex) of the block data.
"Data" on page 5-200	The block data.
"DataAsDouble" on page 5-200	The block data in `double` form.
"Datatype" on page 5-201	Data type of the block data.
"DatatypeID" on page 5-201	Index of the data type of the block data.

Name	Description
“Dimensions” on page 5-202	Dimensions of the block data.
“Name” on page 5-202	Name of the block data.
“Type” on page 5-202	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 5-201 or “DatatypeID” on page 5-201 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 5-205	Individual component within the block specified by the block path

Method Summary

Name	Description
BlockPath on page 5-205	Create a block path.

Name	Description
`convertToCell` on page 5-208	Convert a block path to a cell array of character vectors.
`getBlock` on page 5-209	Get a single block path in the model reference hierarchy.
`getLength` on page 5-210	Get the length of the block path.
`validate` on page 5-210	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_mdlref_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_mdlref_depgraph/thermostat'
```

Create a block path object called `bp2`, using a cell array of character vectors representing elements of the block path.

```
sldemo_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
```

The resulting block path reflects the model reference hierarchy for the block path

```
bp2 =

  Simulink.BlockPath
  Package: Simulink

  Block Path:
    'sldemo_mdlref_depgraph/thermostat'
      'sldemo_mdlref_heater/Fahrenheit to Celsius'
        'sldemo_mdlref_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_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
cellarray_for_bp2 = bp2.convertToCell()
```

The result is a cell array representing the elements of the block path.

```
cellarray_for_bp2 =

    'sldemo_mdlref_depgraph/thermostat'
    'sldemo_mdlref_heater/Fahrenheit to Celsius'
    'sldemo_mdlref_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_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_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_mdlref_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_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_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.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 5-213	True if this port is connected to a bus.
"IsSampleHit" on page 5-213	True if this port produces output or accepts input at the current simulation time step.
"SampleTime" on page 5-213	Sample time of this port.
"SampleTimeIndex" on page 5-213	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

R0

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 5-439 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 5-216	True if this port has direct feedthrough.
“Overwritable” on page 5-216	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 5-440 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 5-217	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.

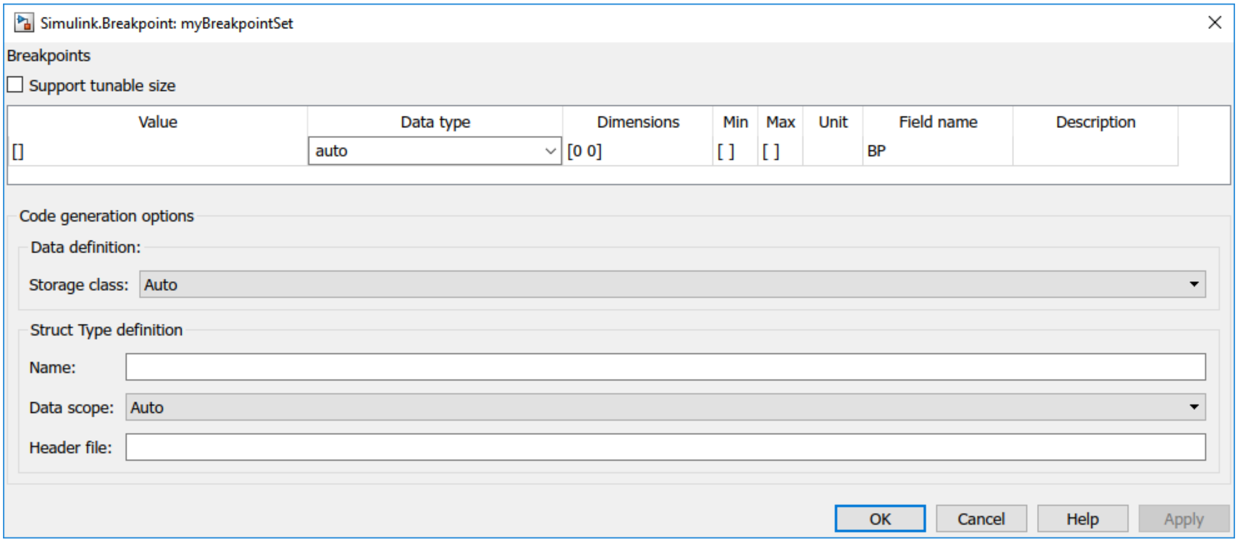
To package lookup table and breakpoint set data into a single structure in the generated code, for example, for STD_AXIS code generation, use a `Simulink.LookupTable` object to store all of the data. See “Package Shared Breakpoint and Table Data for Lookup Tables”.

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:

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. Enumerated data types are also supported.

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

If you have Embedded Coder, you can choose a custom storage class. For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (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 the **Support Tunable Size** parameter.

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: [0×0 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 default header file.

Value	Action
`'Imported'`	Import the data type definition from the specified header file or 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
`Simulink.Bus.cellToObject`	Convert cell array containing bus information to bus objects
`Simulink.Bus.createMATLABStruct`	Create MATLAB structures using same hierarchy and attributes as bus signals
`Simulink.Bus.createObject`	Create bus objects from blocks or MATLAB structures
`Simulink.Bus.objectToCell`	Use bus objects to create cell array containing bus information

Method	Purpose
Simulink.Bus.save	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.

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”
“Organize Data into Structures in Generated Code” (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 the `Simulink.BusElement` 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` — Specify how to handle size of bus element

`'Fixed'` (default) | `'Variable'`

Specify 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”.

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 (see “Create Bus Objects from Blocks”)
- MATLAB data (see “Create Bus Objects from MATLAB Data”)

Compatibility Considerations

`Simulink.BusElement` objects will no longer support the `SamplingMode` property

Not recommended starting in R2016b

In R2016b, the `SamplingMode` property of `Simulink.BusElement` objects was removed. Scripts that use the `SamplingMode` property of `Simulink.BusElement` objects continue to work. `Simulink.Bus.cellToObject` continues to require the `SamplingMode` field and `Simulink.Bus.objectToCell` continues to include the sampling mode in the output cell arrays.

In a future release, support for `SamplingMode` will be removed.

To specify whether a signal is sample-based or frame-based, define the sampling mode of input signals at the block level instead of at the signal level.

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, state, or parameter data

Description

Use a `Simulink.CoderInfo` object to specify code generation settings for signal, state, and parameter data in a model.

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`
- `Simulink.DualScaledParameter`

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';
```

Creation

When you create a data object, Simulink sets the value of the `CoderInfo` property by creating a `Simulink.CoderInfo` object. You do not need to create a `Simulink.CoderInfo` object explicitly.

Properties

`Alias` — Alternative name for code generation

`''` (empty character vector) (default) | character vector

Alternative name for the data in the generated code, specified as a character vector.

Example: `'myOtherName'`

Data Types: `char`

`Alignment` — Data alignment boundary

`-1` (default) | positive integer

Data alignment boundary for this data, specified as a positive integer that is a power of 2, not exceeding 128. Specify an integer number of `double` data type. See "Data Alignment for Code Replacement" (Embedded Coder) for more information.

Example: 8

Data Types: `double`
Complex Number Support: Yes

`CustomAttributes` — Custom storage class attributes of this data

`SimulinkCSC.AttribClass_Simulink_Default` object (default) | custom attributes object

Custom storage class attributes of this data, returned as a custom attributes object. You must set the property `StorageClass` to `'Custom'` to enable this property.

Depending on the custom storage class that you apply by using the `CustomStorageClass` property of the `Simulink.CoderInfo` object, Simulink sets the value of this property by creating a custom attributes object. Then, you can set the values of the properties of the custom attributes object. See "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder) for more information.

`CustomStorageClass` — Custom storage class of this data

`'Default'` (default) | character vector

Custom storage class of this data, specified as a character vector. You must set the property `StorageClass` to `'Custom'` to enable this property.

For a list of valid custom storage classes (Embedded Coder) when you create the data object from the `Simulink` package, see “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

Example: `'ExportToFile'`

Data Types: `char`

`StorageClass` — Storage class of this data
`'Auto'` (default) | character vector

Storage class of this data, specified as a character vector. For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Example: `'ExportedGlobal'`

Data Types: `char`

Examples

Configure Code Generation Settings Programmatically

For examples that show how to configure code generation settings for a data item programmatically, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder).

See Also

Topics

“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)
“Data Objects”
“Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (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

attachComponent	Attach a component to a configuration set
copy	Copy a configuration set
getComponent	Get a configuration set component
getFullName	
getModel	
get_param	Get parameter names and values
isActive	
isValidParam	
saveAs	
setPropEnabled	
set_param	Set system and block parameter values

Examples

Get the Active Configuration Set for a Model

This example gets the active configuration set of the currently selected model.

```
hCs = getActiveConfigSet(gcs);
```

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 5-243	Description of the configuration reference.
“Name” on page 5-243	Name of the configuration reference.
“SourceName” on page 5-244	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 5-244	Create a copy of a configuration reference.
“getFullName” on page 5-244	Get the full pathname of a configuration reference.
“getModel” on page 5-245	Get the handle of the model that owns a configuration reference.

Name	Description
"get_param" on page 5-245	Get the value of a configuration set parameter indirectly through a configuration reference.
"getRefConfigSet" on page 5-246	Get the configuration set specified by a configuration reference.
"isActive" on page 5-246	Determine whether a configuration reference is the active configuration object of the model.
"refresh" on page 5-246	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.FindOptions class

Package: Simulink

Options for finding blocks in models and subsystems

Description

Create an options object to use with `Simulink.findBlocks` and `Simulink.findBlocksOfType` to constrain the search.

Construction

`f = Simulink.FindOptions` creates a `FindOptions` object that uses the default search options.

`f = Simulink.FindOptions(Option1,Value1,...OptionN,ValueN)` creates the object using the specified search options.

Input Arguments

`'CaseSensitive'` — Option to specify whether to match case when searching
`true` (default) | `false`

Option to specify whether to match case when searching, specified as `true` for case-sensitive search or `false` for case-insensitive search.

`'FollowLinks'` — Option to follow library links
`false` (default) | `true`

Option to follow library links, specified as `true` or `false`. If `true`, search follows links into library blocks.

`'IncludeCommented'` — Option for the search to include commented blocks
`true` (default) | `false`

Option for the search to include commented blocks, specified as `true` or `false`.

`'LookUnderMasks'` — Options to search masked blocks

`'all'` (default) | `'none'` | `'functional'` | `'graphical'`

Options to search masked blocks, specified as:

- `'all'` — Search in all masked blocks.
- `'none'` — Prevent searching in masked systems.
- `'functional'` — Include masked subsystems that do not have dialogs.
- `'graphical'` — Include masked subsystems that do not have workspaces or dialogs.

`'Variants'` — Options to search Variant subsystems

`'AllVariants'` (default) | `'ActiveVariants'` | `'ActivePlusCodeVariants'`

Options to search Variant subsystems, specified as:

- `'AllVariants'` — Search all variant choices.
- `'ActiveVariants'` — Search only active variant choices.
- `'ActivePlusCodeVariants'` — Search all variant choices with `'Generate preprocessor conditionals'` active. Otherwise, search only the active variant choices.

The `'Variants'` search constraint applies only to variant subsystems and model variants.

`'RegExp'` — Option to treat the search text as a regular expression

`false` (default) | `true`

Option to treat the search text as a regular expression, specified as `true` or `false`. To learn more about MATLAB regular expressions, see "Regular Expressions" (MATLAB).

`'SearchDepth'` — Levels in the model to search

positive integer

Levels in the model to search, specified as a positive integer. The default is to search all levels. Specify:

- `1` — Search in the top-level system.
- `2` — Search the top-level system and its children, `3` to search an additional level, and so on.

Examples

Specify Search Options to Use with Simulink.findBlocks

Search for all blocks in the Unlocked subsystem but not in any of its children.

```
f = Simulink.FindOptions('SearchDepth',1);
load_system('sldemo_clutch');
bl = Simulink.findBlocks('sldemo_clutch/Unlocked',f)
```

```
bl =

   1.0e+03 *

    1.1140
    1.1150
    1.1160
    1.1170
    1.1180
    1.1190
    1.1200
    1.1210
    1.1220
    1.1230
    1.1240
    1.1250
    1.1260
    1.1270
    1.1280
    1.1290
    1.1300
    1.1310
    1.1320
    1.1330
```

To get the block name and path instead of the handle, use `getfullname`.

```
bl = getfullname (Simulink.findBlocks('sldemo_clutch/Unlocked',f))
```

```
bl =

  20×1 cell array

    {'sldemo_clutch/Unlocked/Tfmaxk'                     }
    {'sldemo_clutch/Unlocked/Tin'                        }
    {'sldemo_clutch/Unlocked/Enable'                     }
    {'sldemo_clutch/Unlocked/E_Sum'                      }
    {'sldemo_clutch/Unlocked/Engine↵Damping'             }
    {'sldemo_clutch/Unlocked/Engine↵Inertia'             }
    {'sldemo_clutch/Unlocked/Engine↵Integrator'          }
    {'sldemo_clutch/Unlocked/Goto'                       }
    {'sldemo_clutch/Unlocked/Goto1'                      }
    {'sldemo_clutch/Unlocked/Max↵Dynamic↵Friction↵Torque'}
    {'sldemo_clutch/Unlocked/V_Sum'                      }
    {'sldemo_clutch/Unlocked/Vehicle↵Damping'            }
    {'sldemo_clutch/Unlocked/Vehicle↵Inertia'            }
    {'sldemo_clutch/Unlocked/Vehicle↵Integrator'         }
    {'sldemo_clutch/Unlocked/W_Slip'                     }
    {'sldemo_clutch/Unlocked/slip direction'             }
    {'sldemo_clutch/Unlocked/w0'                         }
    {'sldemo_clutch/Unlocked/w0 '                        }
    {'sldemo_clutch/Unlocked/we'                         }
    {'sldemo_clutch/Unlocked/wv'                         }
```

See Also

Simulink.allBlockDiagrams | Simulink.findBlocks | Simulink.findBlocksOfType

Introduced in R2018a

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

blockPath1 = 
  Simulink.BlockPath
  Package: Simulink

  Block Path:
    sldemo_absbrake/Bus Creator

  Use the getBlock method to access block path strings from this object.

  Methods

blockPath2 = sig2.BlockPath

blockPath2 = 
  Simulink.BlockPath
  Package: Simulink

  Block Path:
    sldemo_absbrake/Relative Slip

  Use the getBlock method to access block path strings from this object.

  Methods
```

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

"View Data with the Simulation Data Inspector"
"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
```

```
blockPath1 = 
  Simulink.BlockPath
  Package: Simulink

  Block Path:
    sldemo_absbrake/Bus Creator

  Use the getBlock method to access block path strings from this object.

  Methods

blockPath2 = sig2.BlockPath

blockPath2 = 
  Simulink.BlockPath
  Package: Simulink

  Block Path:
    sldemo_absbrake/Relative Slip

  Use the getBlock method to access block path strings from this object.

  Methods
```

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

"View Data with the Simulation Data Inspector"
"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.

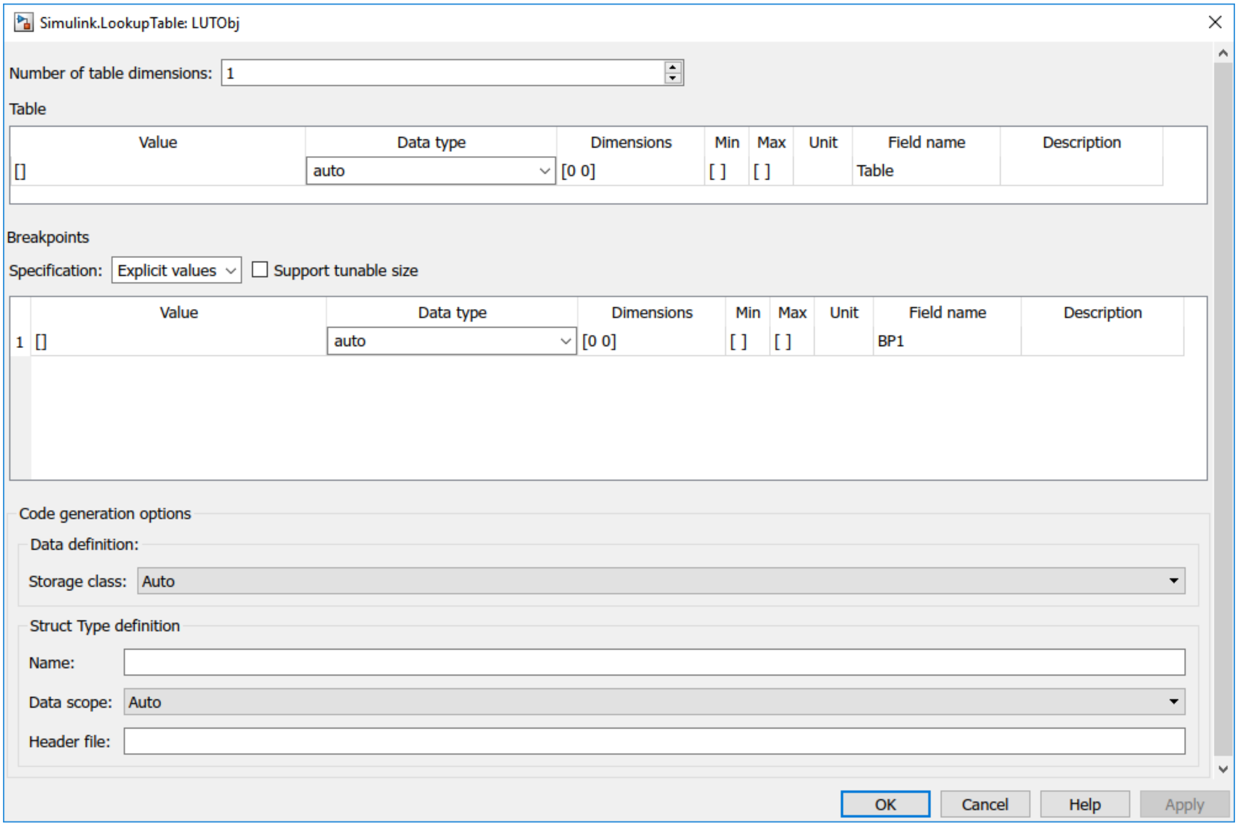
To share a breakpoint set between multiple lookup tables, for example for COM_AXIS code generation, use a `Simulink.Breakpoint` object in one or more Prelookup blocks. Use `Simulink.LookupTable` objects in Interpolation Using Prelookup blocks. Then, configure the lookup table objects to refer to the 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:

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. Enumerated data types are also supported.

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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

If you have Embedded Coder, you can choose a custom storage class. For information about custom storage classes, see “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (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.lookuptable.Breakpoint` objects | cell array of character vectors

Breakpoint set information, specified as a vector of `Simulink.lookuptable.Breakpoint` objects, a cell array of character vectors, or a vector of `Simulink.lookuptable.Evenspacing` objects.

If you use a vector of `Simulink.lookuptable.Breakpoint` objects, each object represents a breakpoint set. Using a vector of `Simulink.lookuptable.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.lookuptable.Evenspacing` objects, each object represents a breakpoint set. Using a vector of `Simulink.lookuptable.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 the **Breakpoints** > **Specification** parameter.

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.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 `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 the **Support Tunable Size** parameter.

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: [1×1 Simulink.lookuptable.Table]
    BreakpointsSpecification: 'Explicit values'
                 Breakpoints: [1×1 Simulink.lookuptable.Breakpoint]
          SupportTunableSize: 0
```

```
                   CoderInfo: [1×1 Simulink.CoderInfo]
              StructTypeInfo: [1×1 Simulink.lookuptable.StructTypeInfo]
```

2 Set up the breakpoint property to even spacing.

```
LUTObj.BreakpointsSpecification='Even spacing'
```

```
LUTObj = 

  LookupTable with properties:

                       Table: [1×1 Simulink.lookuptable.Table]
    BreakpointsSpecification: 'Even spacing'
                 Breakpoints: [1×1 Simulink.lookuptable.Evenspacing]
          SupportTunableSize: 0
                   CoderInfo: [1×1 Simulink.CoderInfo]
              StructTypeInfo: [1×1 Simulink.lookuptable.StructTypeInfo]
```

3 Get the properties of the breakpoint.

```
LUTObj.Breakpoints(1)
```

```
ans = 

  Evenspacing with properties:

         FirstPoint: 0
            Spacing: 1
           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`.

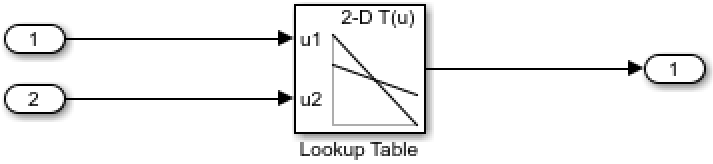
```
#include "ex_myHdr_LUT.h"

#if bp1Len == 2 && bp1Len == 2
LUTObj_Type LUTObj = {
  { 1.0, 2.0 },

  { 3.0, 4.0 },

  { 3.0, 2.0, 4.0, 1.0 }
} ;
#endif

#if bp1Len == 3 && bp1Len == 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.lookuptable.Breakpoint object with default property values, and sets the property that you specified.

The value of the Breakpoints property is an array of Simulink.lookuptable.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.lookuptable.Breakpoint objects and sets the Value property of each object. LUTObj now stores information for three breakpoint sets.

Properties

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`

`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.lookuptable.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.lookuptable.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.lookuptable.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.lookuptable.Evenspacing class

Package: Simulink.lookuptable

Configure even spacing set data for lookup table object

Description

An object of the `Simulink.lookuptable.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.lookuptable.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.lookuptable.Evenspacing` object appears as the value of the `Breakpoints` property.

To create more `Simulink.lookuptable.Evenspacing` objects for a `Simulink.LookupTable` object, use this technique:

Access the `Breakpoints` property by specifying a vector index.

To create a `Simulink.lookuptable.Evenspacing` object, you can set the value of any of the object properties. The `Simulink.LookupTable` object creates the `Simulink.lookuptable.Evenspacing` object with default property values, and sets the property that you specified.

The value of the Breakpoints property is an array of Simulink.lookuptable.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.lookuptable.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.lookuptable.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.lookuptable.StructTypeInfo class

Package: Simulink.lookuptable

Configure settings for structure type that lookup table object uses in the generated code

Description

An object of the `Simulink.lookuptable.StructTypeInfo` class controls the structure type that the generated code creates to store data for lookup table objects. The `Simulink.lookuptable.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.lookuptable.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

`DataType` — 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`

The Description parameter of the model. For details, see “Version Information Properties”.

`FileFormatMinorVersion`

Contains a value only for a Service Pack release of Simulink. Indicates that the file was saved by a Service Pack release, when the Service Pack included changes to the model file format.

`FileName`

The fully-qualified Name of the 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. Logical.

`LastModifiedBy`

Name of the user who last saved the model.

`LastSavedArchitecture`

Platform architecture when saved, for example, `'glnxa64'`.

`Metadata`

Names and values 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 file version number.

`ReleaseName`

Name of release of Simulink that was used to save the model file, 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	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	**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	**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 5-314	Close Model Advisor report.
“deselectCheck” on page 5-314	Clear checks.
“deselectCheckAll” on page 5-315	Clear all checks.
“deselectCheckForGroup” on page 5-315	Clear a group of checks.
“deselectCheckForTask” on page 5-316	Clear checks that belong to a specified task or set of tasks.
“deselectTask” on page 5-316	Clear tasks.
“deselectTaskAll” on page 5-317	Clear all tasks.
“displayReport” on page 5-317	Display Model Advisor report.
“exportReport” on page 5-318	Copy report to a specified location.
“filterResultWithExclusion” on page 5-318	Filter objects that have been excluded by user-defined exclusions.
“getBaselineMode” on page 5-319	Get baseline mode setting for the Model Advisor.
“getCheckAll” on page 5-320	Get the IDs of the checks performed by the Model Advisor.
“getCheckForGroup” on page 5-320	Get checks belonging to a check group.
“getCheckForTask” on page 5-321	Get checks belonging to a task.
“getCheckResult” on page 5-321	Get check results.
“getCheckResultData” on page 5-322	Get check result data.
“getCheckResultStatus” on page 5-322	Get pass/fail status of a check or set of checks.
“getGroupAll” on page 5-323	Get the IDs of the groups of tasks performed by the Model Advisor.
“getInputParameters” on page 5-323	Get input parameters of a check.
“getListViewParameters” on page 5-324	Get list view parameters of a check.

Name	Description
“getModelAdvisor” on page 5-325	Get the Model Advisor for a model or subsystem.
“getSelectedCheck” on page 5-325	Get selected checks.
“getSelectedSystem” on page 5-326	Get path of system currently targeted by the Model Advisor.
“getSelectedTask” on page 5-326	Get selected tasks.
“getTaskAll” on page 5-327	Get the IDs of the tasks performed by the Model Advisor.
“Simulink.ModelAdvisor.openConfigUI” on page 5-327	Start the Model Advisor Configuration editor.
“Simulink.ModelAdvisor.reportExists” on page 5-328	Determine whether a report exists for a system or subsystem.
“runCheck” on page 5-328	Run selected checks.
“runTask” on page 5-329	Run checks for selected tasks.
“selectCheck” on page 5-329	Select checks.
“selectCheckAll” on page 5-330	Select all checks.
“selectCheckForGroup” on page 5-330	Select a group of checks.
“selectCheckForTask” on page 5-331	Select checks that belong to a specified task.
“selectTask” on page 5-331	Select tasks.
“selectTaskAll” on page 5-332	Select all tasks.
“setActionEnable” on page 5-332	Set enable/disable status for a check action.
“setBaselineMode” on page 5-333	Set baseline mode for the Model Advisor.
“setCheckErrorSeverity” on page 5-334	Set severity of a check failure.
“setCheckResult” on page 5-334	Set result for the currently running check.
“setCheckResultData” on page 5-335	Set result data for the currently running check.

Name	Description
"setCheckResultStatus" on page 5-336	Set pass/fail status for the currently running check.
"setListViewParameters" on page 5-336	Set list view parameters for a check.
"verifyCheckRan" on page 5-337	Verify that checks have run.
"verifyCheckResult" on page 5-338	Generate a baseline set of check results or compare the current set of results to the baseline results.
"verifyCheckResultStatus" on page 5-339	Verify that a model has passed or failed a set of checks.
"verifyHTML" on page 5-340	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 5-317

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 5-320, “deselectCheckForGroup” on page 5-315, “selectCheck” on page 5-329

deselectCheckAll

Clear all checks

```
success = deselectCheckAll
```

`success`

True (1) if all checks are cleared.

Clears all checks that are not disabled.

“selectCheckAll” on page 5-330

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 5-330

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 5-327, “selectCheckForTask” on page 5-331

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 5-331, “getTaskAll” on page 5-327

deselectTaskAll

Clears all tasks

```
success = deselectTaskAll
```

`success`

True (1) if this method succeeds in clearing all tasks.

Clears all tasks.

“selectTaskAll” on page 5-332

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 5-328

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 5-328

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 5-325

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 5-340, to generate baseline data.

“setBaselineMode” on page 5-333, “verifyHTML” on page 5-340, “verifyCheckResult” on page 5-338, “verifyCheckResultStatus” on page 5-339

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 5-327, “getGroupAll” on page 5-323

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 5-321

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 5-320

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).

“getCheckResultData” on page 5-322, “getCheckResultStatus” on page 5-322

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).

“getCheckResult” on page 5-321, “getCheckResultStatus” on page 5-322

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 5-321, “getCheckResultData” on page 5-322

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 5-320, “getTaskAll” on page 5-327

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).

`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).

“setListViewParameters” on page 5-336, `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 5-326

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.ModelAdisor.getModelAdvisor`.

“getModelAdvisor” on page 5-325

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 5-325

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 5-320, “getGroupAll” on page 5-323

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 5-318

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 5-329

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 5-331

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 5-330, “selectCheckForGroup” on page 5-330, “deselectCheck” on page 5-314

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 5-329, “selectCheckForGroup” on page 5-330, “deselectCheck” on page 5-314

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 5-315

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 5-316

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 5-316

selectTaskAll

Select all tasks

```
success = selectTaskAll
```

`success`

True (1) if this method succeeds in selecting all tasks.

Selects all tasks.

“deselectTaskAll” on page 5-317

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).

`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 5-319, “verifyCheckResult” on page 5-338, “verifyHTML” on page 5-340

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).

“setCheckResultStatus” on page 5-336

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).

“getCheckResult” on page 5-321, “setCheckResultData” on page 5-335, “setCheckResultStatus” on page 5-336

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).

“getCheckResultData” on page 5-322, “setCheckResult” on page 5-334, “setCheckResultStatus” on page 5-336

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).

“getCheckResultStatus” on page 5-322, “setCheckResult” on page 5-334, “setCheckResultData” on page 5-335, “setCheckErrorSeverity” on page 5-334

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).

“getListViewParameters” on page 5-324, `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 5-339

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 5-333), 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 5-328) 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 5-333. 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 5-319). Then invoke this method with the path of the baseline report as the value of the `baseline` input argument.

“setBaselineMode” on page 5-333, “getBaselineMode” on page 5-319, “runCheck” on page 5-328, “verifyCheckResultStatus” on page 5-339

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 passor 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 5-328) before invoking `verifyCheckResultStatus`.

“runCheck” on page 5-328

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 5-333), 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 5-319). The invoke this method with the path of the baseline report as the value of the *baseline* input argument.

“setBaselineMode” on page 5-333, “getBaselineMode” on page 5-319, “verifyCheckResult” on page 5-338

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 logsout.

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

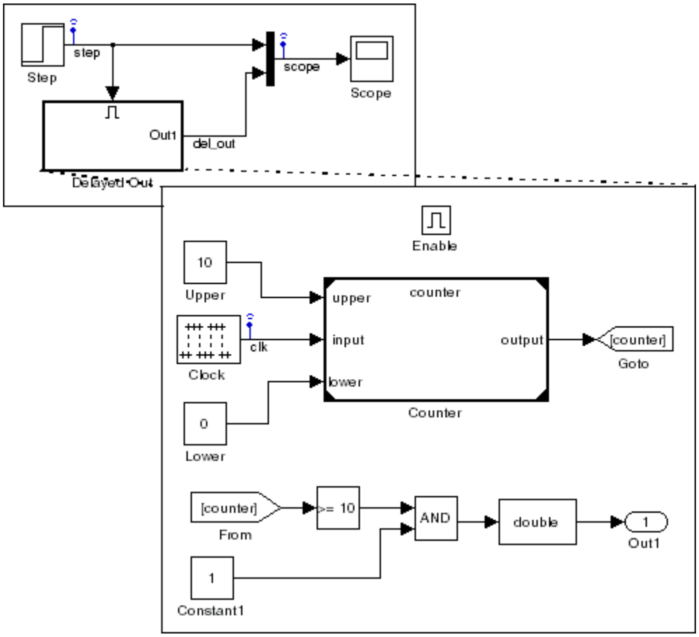
- `Simulink.TsArray`

  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	`Simulink.ModelDataLogs`
Subsystem in a model	`Simulink.SubsysDataLogs`
Signal other than a bus or Mux signal	`Simulink.Timeseries`
Bus signal or Mux signal	`Simulink.TsArray`

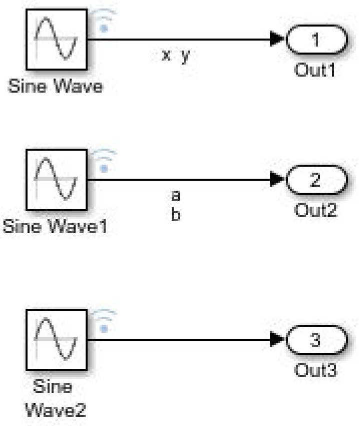
Handling Spaces and Newlines in Logged Names

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.

```
logsout
```

```
logsout =

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:

```
logsout.x y

??? logsout.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:

```
logsout.('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')
logsout.(['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:

```
logsout.(['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

getBlockSimState	Access SimState of individual Stateflow Chart, MATLAB Function, or S-function block
setBlockSimState	Set SimState of individual Stateflow Chart, MATLAB Function, or S-function block

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

Simulink.ModelWorkspace

Interact with the model workspace of a model programmatically

Description

Use a `Simulink.ModelWorkspace` object to interact with a model workspace. For example, you can add and remove variables, set the data source of the workspace, and save changes to the workspace.

For more information, see “Model Workspaces”.

Creation

To create a `Simulink.ModelWorkspace`, use the `get_param` function to query the value of the model parameter `ModelWorkspace`. For example, to create an object named `mdlWks` that represents the model workspace of a model named `myModel.slx`:

```
mdlWks = get_param('myModel','ModelWorkspace')
```

Properties

`DataSource` — Source for initializing variables in model workspace

`'Model File'` (default) | `'MAT-File'` | `'MATLAB Code'` | `'MATLAB File'`

Source for initializing the variables in the model workspace, specified as one of these character vectors:

- `'Model File'` — The variables are stored in the model file. When you save the model, you also save the variables.
- `'MATLAB Code'` — The variables are created by MATLAB code that you write and store in the model file.
- `'MAT-File'` — The variables are stored in a MAT-file, which you can manage and manipulate separately from the model file.

- `'MATLAB File'` — The variables are created by MATLAB code in a script file, which you can manage and manipulate separately from the model file.

Data Types: `char`

`FileName` — Name of external file that stores or creates variables

`''` (empty character vector) (default) | character vector

Name of the external file that stores or creates variables, specified as a character vector. To enable this property, set `DataSource` to `'MAT-File'` or `'MATLAB File'`.

Example: `'myFile.mat'`

Example: `'myFile.m'`

Data Types: `char`

`MATLABCode` — MATLAB code for initializing variables

`''` (empty character vector) (default) | character vector

MATLAB code for initializing variables, specified as a character vector. To enable this property, set `DataSource` to `'MATLAB Code'`.

Example: `sprintf('%% Create variables that this model uses.\n\nK = 0.00983;\n\nP = Simulink.Parameter(5);')`

Data Types: `char`

Object Functions

getVariable	Return value of variable in the model workspace of a model
getVariablePart	Get value of variable property in model workspace
setVariablePart	Set property of variable in model workspace
hasVariable	Determine whether variable exists in the model workspace of a model
whos	Return list of variables in the model workspace of a model
saveToSource	Save model workspace changes to the external data source of the model workspace
save	Save contents of model workspace to a MAT-file
reload	Reinitialize variables from the data source of a model workspace
evalin	Evaluate expression in the model workspace of a model
clear	Clear variables from the model workspace of a model
assignin	Assign value to variable in the model workspace of a model

Examples

Interact With Model Workspace Programmatically

Create a variable in the model workspace of a model. Then, modify the variable and query the variable value to confirm the modification.

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object `mdlWks` that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named `myVar` with value `5.12` in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
```

Apply a new value, `7.22`. To do so, first create a temporary copy of the variable by using the `getVariable` function. Then, modify the copy and use it to overwrite the original variable in the model workspace.

```
temp = getVariable(mdlWks,'myVar');
temp = 7.22;
assignin(mdlWks,'myVar',temp)
```

Confirm the new value by querying the value of the variable.

```
getVariable(mdlWks,'myVar')
```

```
ans =

    7.2200
```

See Also

Topics

“Model Workspaces”

“Variables”

Introduced before R2006a

assignin

Package: Simulink

Assign value to variable in the model workspace of a model

Syntax

```
assignin(mdlWks,varName,varValue)
```

Description

`assignin(mdlWks,varName,varValue)` assigns the value `varValue` to the MATLAB variable `varName` in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`. If the variable does not exist, `assignin` creates it.

Examples

Assign Value to Variable in Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named `myVar` with value `5.12` in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
```

Input Arguments

`mdlWks` — Target model workspace

`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

`varName` — Name of target variable

character vector

Name of the target variable, specified as a character vector.

Example: `'myVar'`

Data Types: `char`

`varValue` — Value to assign to target variable

valid value

Value to assign to the target value, specified as a valid value. For example, you can specify a literal number, a structure, or an expression that evaluates to a valid value.

If you specify the name of a handle object, such as a `Simulink.Parameter` object, use the `copy` function to create a separate copy of the object.

Example: `5.12`

Example: `struct('a',5.12,'b',7.22)`

Example: `Simulink.Parameter(5.12)`

Example: `copy(myExistingParameterObject)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

See Also

`Simulink.ModelWorkspace`

Introduced before R2006a

clear

Package: Simulink

Clear variables from the model workspace of a model

Syntax

```
clear(mdlWks)
```

Description

`clear(mdlWks)` removes all variables from the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

Examples

Clear Variables From Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named `myVar` with value `5.12` in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
```

Clear all variables from the model workspace, including `myVar`.

```
clear(mdlWks)
```

Input Arguments

`mdlWks` — Target model workspace
`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

See Also

`Simulink.ModelWorkspace`

Introduced before R2006a

evalin

Package: Simulink

Evaluate expression in the model workspace of a model

Syntax

```
result = evalin(mdlWks,expression)
```

Description

`result = evalin(mdlWks,expression)` evaluates the expression `expression` in the model workspace represented by the `Simulink.ModelWorkspace` object represented by `mdlWks`. The function returns the result of the expression in `result`.

Note For setting and getting variable properties in the model workspace, consider using `setVariablePart` and `getVariablePart` instead of `evalin` because:

- The functions do not create new variables or cause unintended results.
- `getVariablePart` does not dirty the model.

For information on these alternatives, see `Simulink.ModelWorkspace`.

Examples

Evaluate Expression in Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create some variables in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)
```

Evaluate the expression `myLastVar = myVar + myOtherVar` in the model workspace. The expression creates another variable, `myLastVar`, whose value is the sum of the first two variables.

```
evalin(mdlWks,'myLastVar = myVar + myOtherVar');
```

Input Arguments

`mdlWks` — Target model workspace

`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

`expression` — Expression to evaluate

character vector

Expression to evaluate, specified as a character vector.

Example: `'myLastVar = myVar + myOtherVar'`

Data Types: `char`

Output Arguments

`result` — Result of expression

number, structure, or other MATLAB value

Result of the evaluated expression, returned as a number, structure, or other MATLAB value.

See Also

`Simulink.ModelWorkspace`

Introduced before R2006a

reload

Package: Simulink

Reinitialize variables from the data source of a model workspace

Syntax

```
reload(mdlWks)
```

Description

`reload(mdlWks)` reinitializes the variables in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`. When you set the `DataSource` property of the model workspace to `'MAT-File'`, `'MATLAB File'`, or `'MATLAB Code'`, `reload` overwrites variables that exist in the model workspace by loading the associated MAT-file or by running the associated MATLAB code.

Examples

Reinitialize Variables in a Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Configure the model workspace to use some MATLAB code as a data source.

```
mdlWks.DataSource = 'MATLAB Code';
mdlWks.MATLABCode = sprintf('myVar = 5.12;\nmyOtherVar = 7.22;');
```

Create variables in the model workspace by executing the MATLAB code.

```
reload(mdlWks)
```

Assign new values to the variables in the model workspace.

```
assignin(mdlWks,'myVar',5.22)
assignin(mdlWks,'myOtherVar',7.33)
```

Overwrite the new values with the values specified by the MATLAB code.

```
reload(mdlWks)
```

Confirm that the variables have the values specified by the MATLAB code.

```
myVarValue = getVariable(mdlWks,'myVar')
myOtherVarValue = getVariable(mdlWks,'myOtherVar')
```

```
myVarValue =

    5.1200


myOtherVarValue =

    7.2200
```

Input Arguments

`mdlWks` — Target model workspace
`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

See Also

`Simulink.ModelWorkspace`

Introduced before R2006a

save

Package: Simulink

Save contents of model workspace to a MAT-file

Syntax

```
save(mdlWks,fileName)
```

Description

`save(mdlWks,fileName)` saves the variables in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks` to the MAT-file specified by `fileName`.

When you set the `DataSource` property of the model workspace to `'MAT-File'` or `'MATLAB File'`, to save to the file that acts as the external data source of the model, use `saveToSource` instead of `save`.

Examples

Save Contents of Model Workspace to MAT-File

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create some variables in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)
```

Save the variables to a new MAT-file named `myVars.mat`.

```
save(mdlWks,'myVars.mat')
```

The MAT-file appears in your current folder.

Input Arguments

`mdlWks` — Target model workspace

`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

`fileName` — Name of target MAT-file

character vector

Name of the target MAT-file, specified as a character vector.

Example: `'myFile.mat'`

Data Types: `char`

See Also

`Simulink.ModelWorkspace`

Introduced before R2006a

saveToSource

Package: Simulink

Save model workspace changes to the external data source of the model workspace

Syntax

```
saveToSource(mdlWks)
```

Description

`saveToSource(mdlWks)` saves the variables in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks` to the MAT-file or script file specified by the `FileName` property of the model workspace.

When you set the `DataSource` property of the model workspace to `'MAT-File'` or `'MATLAB File'`, the `FileName` property specifies the name of the file that acts as the external data source of the workspace. As you make changes to the variables in the model workspace, use `saveToSource` to permanently save the changes in the external data source.

Examples

Save Variables to External Data Source of Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create some variables in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)
```

Configure the model workspace to use a MAT-file named `myVars.mat` as the data source.

```
mdlWks.DataSource = 'MAT-File';
mdlWks.FileName = 'myVars.mat';
```

Save the variables to the external data source (the MAT-file).

```
saveToSource(mdlWks)
```

The file appears in your current folder.

Input Arguments

`mdlWks` — Target model workspace

`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

See Also

`Simulink.ModelWorkspace`

Introduced before R2006a

whos

Package: Simulink

Return list of variables in the model workspace of a model

Syntax

```
varList = whos(mdlWks)
```

Description

`varList = whos(mdlWks)` returns a list of the variables in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

Examples

Return List of Variables in Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create some variables in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)
```

Display a list of the variables in the model workspace.

```
whos(mdlWks)
```

```
  Name            Size            Bytes  Class     Attributes

  myOtherVar      1x1                 8  double
  myVar           1x1                 8  double
```

Input Arguments

`mdlWks` — Target model workspace
`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

Output Arguments

`varList` — List of variables
nested structure array

List of variables, returned as a nested structure array. For details about the information in the list, see `whos`.

See Also

`Simulink.ModelWorkspace`

Introduced before R2006a

getVariable

Package: Simulink

Return value of variable in the model workspace of a model

Syntax

```
varValue = getVariable(mdlWks,varName)
```

Description

`varValue = getVariable(mdlWks,varName)` returns the value of the variable whose name is `varName` that exists in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

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 5-371).

Examples

Return Value of Variable in Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a Simulink.ModelWorkspace object that represents the model workspace of vdp.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named myVar in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
```

Return the value of the new variable. Store the value in another variable named varValue.

```
varValue = getVariable(mdlWks,'myVar');
```

Modify Property Value of Handle Object

Modify a property of the Simulink.Parameter object K, which is defined in model mdl.slx. 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);
```

Input Arguments

mdlWks — Target model workspace
Simulink.ModelWorkspace object

Target model workspace, specified as a Simulink.ModelWorkspace object.

varName — Name of target variable
character vector

Name of the target variable, specified as a character vector.

Example: 'myVariable'

Data Types: char

Output Arguments

`varValue` — Value of target variable
number, structure, or other MATLAB value

Value of the target variable, returned as a number, structure, or other MATLAB value.

See Also

`Simulink.ModelWorkspace` | `get_param`

Introduced in R2012a

getVariablePart

Package: Simulink

Get value of variable property in model workspace

Syntax

```
varValue = getVariablePart(mdlWks,varName.Property)
```

Description

`varValue = getVariablePart(mdlWks,varName.Property)` returns the value of the variable property named `varName.Property` that exists in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

If the value of the variable property is a handle to a handle object (such as `Simulink.Parameter`), `getVariablePart` returns a copy of the handle.

Using `getVariablePart` is preferable to using `evalin` for getting variable properties in the model workspace because:

- The function does not dirty the model.
- Use of the function does not result in the creation of a new variable or other unintended results.

Examples

Return Value of Variable Properties in Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a structure named `myStruct` with fields `a`, `b`, and `c`.

```
aStruct.a = 10;
aStruct.b = {1,2,3,4,5};
aStruct.c = Simulink.Parameter(7);
mdlWks.assignin('myStruct',aStruct);
```

Return the values of the structure fields. Store the values in `varValuea`, `varValueb`, and `varValuec`.

```
varValuea = getVariablePart(mdlWks,'myStruct.a');
varValueb = getVariablePart(mdlWks,'myStruct.b{1}');
varValueC = getVariablePart(mdlWks,'myStruct.c.Value');
```

Input Arguments

`mdlWks` — Target model workspace
`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

`varName.Property` — Name of target variable property
character vector

Name of the target variable property, specified as a character vector.

Example: `'myVariable.Property'`

Data Types: `char`

Output Arguments

`varValue` — Value of variable property
number, structure, or other MATLAB value

Value of the variable property, returned as a number, structure, or other MATLAB value.

If the value of the variable property is a handle to a handle object (such as `Simulink.Parameter`), `getVariablePart` returns a copy of the handle.

See Also

`Simulink.ModelWorkspace`

Introduced in R2018b

setVariablePart

Package: Simulink

Set property of variable in model workspace

Syntax

```
varValue = setVariablePart(mdlWks,varName.Property, varValue)
```

Description

`varValue = setVariablePart(mdlWks,varName.Property, varValue)` assigns `varValue` to the MATLAB variable property `varName.Property` in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

Using `setVariablePart` is preferable to using `evalin` for assigning variable properties in the model workspace because the `setVariablePart` function does not create a new variable or cause unintended results.

Examples

Assign Value to Variable Properties in Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a structure named `myStruct` with fields `a`, `b`, and `c`.

```
aStruct.a = 10;
aStruct.b = {1,2,3,4,5};
aStruct.c = Simulink.Parameter(7);
mdlWks.assignin('myStruct',aStruct);
```

Assign new values to the structure fields.

```
setVariablePart(mdlWks,'myStruct.a', 2);
setVariablePart(mdlWks,'myStruct.b{1}', 2);
setVariablePart(mdlWks,'myStruct.c', Simulink.Parameter(2));
```

Return the new values of the structure fields. Store the values in `varValuea`, `varValueb`, and `varValuec`.

```
varValuea = getVariablePart(mdlWks,'myStruct.a')
varValueb = getVariablePart(mdlWks,'myStruct.b{1}')
varValueC = getVariablePart(mdlWks,'myStruct.c.Value')
```

Input Arguments

`mdlWks` — Target model workspace

`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

`varName.Property` — Name of target variable property

character vector

Name of the target variable, property specified as a character vector.

Example: `'myVariable.Property'`

Data Types: `char` | `string`

`varValue` — Value to assign to variable property

valid value

Value to assign to the value property, specified as a valid value. For example, you can specify a literal number, a structure, or an expression that evaluates to a valid value.

If you specify the name of a handle object, such as a `Simulink.Parameter` object, use the `copy` function to create a separate copy of the object.

Example: `5.12`

Example: `struct('a',5.12,'b',7.22)`

Example: `Simulink.Parameter(5.12)`

Example: `copy(myExistingParameterObject)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

See Also

`Simulink.ModelWorkspace`

Introduced in R2018b

hasVariable

Package: Simulink

Determine whether variable exists in the model workspace of a model

Syntax

```
varExists = hasVariable(mdlWks,varName)
```

Description

`varExists = hasVariable(mdlWks,varName)` returns `1` if a variable whose name is `varName` exists in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

Examples

Determine Existence of Variable in Model Workspace

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named `myVar` in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
```

Determine whether a variable named `myVar` exists in the model workspace.

```
exists = hasVariable(mdlWks,'myVar')
exists =
   1
```

Input Arguments

`mdlWks` — Target model workspace
`Simulink.ModelWorkspace` object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

`varName` — Name of target variable
character vector

Name of the target variable, specified as a character vector.

Example: `'myVariable'`

Data Types: `char`

Output Arguments

`varExists` — Indication of existence
`1` | `0`

Indication of variable existence, returned as `1` (true) or `0`.

See Also

`Simulink.ModelWorkspace` | `get_param`

Introduced in R2012a

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 5-384	enable Level-2 MATLAB S-function to use multidimensional signals.
“DialogPrmsTunable” on page 5-384	Specifies which of the S-function's dialog parameters are tunable.
“NextTimeHit” on page 5-384	Time of the next sample hit for variable sample time S-functions.

Method Summary

Name	Description
“AutoRegRuntimePrms” on page 5-385	Register this block's dialog parameters as run-time parameters.
“AutoUpdateRuntimePrms” on page 5-385	Update this block's run-time parameters.
“IsDoingConstantOutput” on page 5-385	Determine whether the current simulation stage is the constant sample time stage.
“IsMajorTimeStep” on page 5-386	Determine whether the current simulation time step is a major time step.
“IsSampleHit” on page 5-386	Determine whether the current simulation time is one at which a task handled by this block is active.
“IsSpecialSampleHit” on page 5-387	Determine whether the current simulation time is one at which multiple tasks handled by this block are active.
“RegBlockMethod” on page 5-388	Register a callback method for this block.

Name	Description
“RegisterDataTypeFxpBinaryPoint” on page 5-388	Register fixed-point data type with binary point-only scaling.
“RegisterDataTypeFxpFSlopeFixexpBias” on page 5-389	Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias.
“RegisterDataTypeFxpSlopeBias” on page 5-390	Register data type with [Slope Bias] scaling.
“SetAccelRunOnTLC” on page 5-392	Specify whether to use this block's TLC file to generate the simulation target for the model that uses it.
“SetPreCompInpPortInfoToDynamic” on page 5-392	Set precompiled attributes of this block's input ports to be inherited.
“SetPreCompOutPortInfoToDynamic” on page 5-393	Set precompiled attributes of this block's output ports to be inherited.
“SetPreCompPortInfoToDefaults” on page 5-393	Set precompiled attributes of this block's ports to the default values.
“SetSimViewingDevice” on page 5-393	Specify whether block is a viewer.
“SupportsMultipleExecInstances” on page 5-394	
“WriteRTWParam” on page 5-394	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 5-385 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 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 5-437.

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 5-389 to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use “RegisterDataTypeFxpSlopeBias” on page 5-390 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 5-437.

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 5-388 to register a data type with binary point-only scaling.
- Use “RegisterDataTypeFxpSlopeBias” on page 5-390 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 5-437.

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 5-388 to register a data type with binary point-only scaling.
- Use “RegisterDataTypeFxpFSlopeFixexpBias” on page 5-389 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 `msfcn_times_two.m` in the Simulink model `msfcndemo_timestwo` for an example.

Note The default JIT Accelerator mode does not support inlining of user-written TLC S-Functions. Please see “How Acceleration Modes Work” and “Control S-Function Execution” for more information.

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

Use a `Simulink.NumericType` object to set and share numeric data types for signal, state, and parameter data in a model.

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 clear the **Is alias** property.
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 (by generating a `typedef` statement), you can use an object of the class `Simulink.AliasType`.

Creation

To create a `Simulink.NumericType` object interactively, use the Model Explorer:

1. On the Model Explorer **Model Hierarchy** pane, select a workspace, such as the base workspace, or a data dictionary.
2. From the Model Explorer **Add** menu, select **Simulink.NumericType**.

   The Model Explorer creates an instance of a `Simulink.NumericType` object and assigns it to a variable named `Numeric` in the target 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, use the **Data type mode** property to select a data type that the object represents.

To create a `Simulink.NumericType` object programmatically, use the `Simulink.NumericType` function described below.

Syntax

`typeObj = Simulink.NumericType`

Description

`typeObj = Simulink.NumericType` returns a `Simulink.NumericType` object with default property values.

Properties

`Bias` — Bias for slope and bias scaling

`0` (default) | real number

Bias for slope and bias scaling of a fixed-point data type (Fixed-Point Designer), specified as a real number.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

Corresponds to **Bias** in the property dialog box.

Example: 3

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

DataScope — Specification to generate or import type definition in the generated code

'Auto' (default) | 'Exported' | 'Imported'

Specification to generate or import the type definition (typedef) in the generated code (Simulink Coder), specified as 'Auto', 'Exported, or 'Imported'.

The table shows the effect of each option.

Value	Action
'Auto' (default)	If no value is specified for HeaderFile, export the type definition to *model*_types.h. *model* is the model name. If a value is specified for HeaderFile, 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 HeaderFile property. If no value is specified for HeaderFile, the header file name defaults to *type*.h. *type* is the data type name.
'Imported'	Import the data type definition from a header file, which can be specified in the HeaderFile property. If no value is specified for HeaderFile, the header file name defaults to *type*.h. *type* is the data type name.

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to **Data scope** in the property dialog box.

DataTypeMode — Mode of numeric data type

'Double' (default) | 'Single' | 'Boolean' | 'Fixed-point: unspecified scaling' | 'Fixed-point: binary point scaling' | 'Fixed-point: slope and bias scaling'

Mode of the numeric data type, specified as one of these character vectors:

- '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.
- `'Fixed-point: slope and bias scaling'` — A fixed-point data type with slope and bias scaling.

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.

Corresponds to **Data type mode** in the property dialog box.

Data Types: `char`

`DataTypeOverride` — Data type override mode

`'Inherit'` (default) | `'Off'`

Data type override mode, specified as `'Inherit'` or `'Off'`.

- If you specify `'Inherit'`, the 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.
- If you specify `'Off'`, data type override does not apply to this numeric type.

For more information about data type override, see “Control Data Type Override”.

Corresponds to **Data type override** in the property dialog box.

Data Types: `char`

`Description` — Custom description of data type

`''` (empty character vector) (default) | character vector

Custom description of the data type, specified as a character vector.

Corresponds to **Description** in the property dialog box.

Example: `'This is a floating-point data type.'`

Data Types: `char`

`FixedExponent` — Exponent for binary point scaling

`0` (default) | real number

Exponent for binary point scaling, specified as a real number. 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`.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

This property does not appear in the property dialog box.

Example: `-8`

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

`FractionLength` — Bit length of the fractional portion of a fixed-point number

`0` (default) | real integer

Bit length of the fractional portion of a fixed-point number (Fixed-Point Designer), specified as a real integer. This property equals `-FixedExponent`. Setting this property causes Simulink software to set the `FixedExponent` property accordingly, and vice versa.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

Corresponds to **Fraction length** in the property dialog box.

Example: 8

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

`HeaderFile` — Name of header file that contains type definition in the generated code

`''` (empty character vector) (default) | character vector

Name of the header file that contains the type definition (`typedef`) in the generated code, specified as a character vector.

If this property is specified, the specified name is used during code generation for importing or exporting. If this property is empty, the value defaults to *`type`*`.h` if `DataScope` equals `'Imported'` or `'Exported'`, or defaults to *`model`*`_types.h` if `DataScope` equals `'Auto'`.

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>'`.

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to **Header file** in the property dialog box.

Example: `'myHdr.h'`

Example: `'myHdr'`

Example: `'myHdr.hpp'`

Data Types: `char`

`IsAlias` — Specification to create data type alias using object name

`false` (default) | `true`

Specification to create a data type alias by using the name of the object, specified as `true` (yes) or `false` (no).

If you specify `true`, the object acts as a data type alias in a similar manner to a `Simulink.AliasType` object. For more information, see “Control Data Type Names in Generated Code” (Embedded Coder).

Corresponds to **Is alias** in the property dialog box.

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

`Signedness` — Signedness of fixed-point data type

`'Signed` (default) | `'Unsigned'` | `'Auto'`

Signedness of a fixed-point data type (Fixed-Point Designer), specified as `'Signed'` (signed), `'Unsigned'` (unsigned), or `'Auto'` (inherit signedness).

Corresponds to **Signedness** in the property dialog box.

Data Types: `char`

`Slope` — Slope for slope and bias scaling of fixed-point data type

`2^0` (default) | real number

Slope for slope and bias scaling of a fixed-point data type (Fixed-Point Designer), specified as a real number.

This property equals `SlopeAdjustmentFactor * 2^FixedExponent`. If `SlopeAdjustmentFactor` is 1.0, Simulink software displays the value of this field as `2^SlopeAdjustmentFactor`. Otherwise, it displays it as a numeric value. Setting this property causes Simulink software to set the `FixedExponent` and `SlopeAdjustmentFactor` properties accordingly, and vice versa.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

This property appears only if `DataTypeMode` is `Fixed-point: slope and bias scaling`.

Corresponds to **Slope** in the property dialog box.

Example: `5.2`

Example: `2^9`

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

`SlopeAdjustmentFactor` — Slope for slope and bias scaling of fixed-point data type

`1` (default) | real number

Slope for slope and bias scaling of a fixed-point data type (Fixed-Point Designer), specified as a real number in the range [1, 2).

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`.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

This property does not appear in the property dialog box.

Example: `1.7`

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

`WordLength` — Word size of fixed-point or integer data type

16 (default) | integer

Word size of a fixed-point (Fixed-Point Designer) or integer data type, specified as an integer number of bits.

This property appears only if `DataTypeMode` is `Fixed-point`.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

Corresponds to **Word length** in the property dialog box.

Example: 8

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

Object Functions

isboolean	Determine whether numeric type represents the Boolean data type boolean
isdouble	Determine whether numeric type represents the double-precision, floating-point data type double
isfixed	Determine whether numeric type represents a fixed-point data type
isfloat	Determine whether numeric type represents a floating-point data type
isscalingbinarypoint	Determine whether fixed-point numeric type has binary-point scaling
isscalingslopebias	Determine whether numeric type represents a fixed-point data type with slope-and-bias scaling
isscalingunspecified	Determine whether numeric type represents a data type with unspecified scaling
issingle	Determine whether numeric type represents the single-precision, floating-point data type single

Examples

Share a Data Type Between Separate Algorithms, Data Paths, Models, and Bus Elements

See “Share a Data Type Between Separate Algorithms, Data Paths, Models, and Bus Elements”.

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

isboolean

Package: Simulink

Determine whether numeric type represents the Boolean data type `boolean`

Syntax

```
indication = isboolean(numericType)
```

Description

`indication = isboolean(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents the Boolean data type `boolean` and `0` (`false`) otherwise.

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

`numericType` — Target numeric type

`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

`indication` — Indication of whether target object represents boolean

`1` | `0`

Indication of whether the target object represents `boolean`, returned as `1` (`true`) or `0` (false).

See Also

`Simulink.NumericType`

Topics

"Control Signal Data Types"
"Control Block Parameter Data Types"
"Control Data Type Names in Generated Code" (Embedded Coder)

Introduced in R2010b

isdouble

Package: Simulink

Determine whether numeric type represents the double-precision, floating-point data type `double`

Syntax

```
indication = isdouble(numericType)
```

Description

`indication = isdouble(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents the double-precision, floating-point data type `double` and `0` (`false`) otherwise.

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

`numericType` — Target numeric type

`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

`indication` — Indication of whether target object represents double

`1` | `0`

Indication of whether the target object represents `double`, returned as `1` (`true`) or `0` (`false`).

See Also

`Simulink.NumericType` | `isfloat` | `issingle`

Topics

“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b

isfixed

Package: Simulink

Determine whether numeric type represents a fixed-point data type

Syntax

```
indication = isfixed(numericType)
```

Description

`indication = isfixed(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents a fixed-point data type and `0` (`false`) otherwise.

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

numericType — Target numeric type

`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

indication — Indication of whether target object represents a fixed-point type

`1` | `0`

Indication of whether the target object represents a fixed-point type, returned as `1` (`true`) or `0` (`false`).

See Also

`Simulink.NumericType` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified`

Topics

"Control Signal Data Types"
"Control Block Parameter Data Types"
"Control Data Type Names in Generated Code" (Embedded Coder)

Introduced in R2010b

isfloat

Package: Simulink

Determine whether numeric type represents a floating-point data type

Syntax

```
indication = isfloat(numericType)
```

Description

`indication = isfloat(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents a floating-point data type such as `double` or `single`, and `0` (`false`) otherwise.

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

`numericType` — Target numeric type

`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

`indication` — Indication of whether target object represents a floating-point type

`1` | `0`

Indication of whether the target object represents a floating-point type, returned as `1` (`true`) or `0` (false).

See Also

`Simulink.NumericType` | `isdouble` | `issingle`

Topics

"Control Signal Data Types"
"Control Block Parameter Data Types"
"Control Data Type Names in Generated Code" (Embedded Coder)

Introduced in R2010b

isscalingbinarypoint

Package: Simulink

Determine whether fixed-point numeric type has binary-point scaling

Syntax

```
indication = isscalingbinarypoint(numericType)
```

Description

`indication = isscalingbinarypoint(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents a fixed-point data type with binary-point scaling and `0` (`false`) otherwise. A numeric type object can use binary-point scaling if you explicitly specify it or if you specify trivial slope-and-bias scaling (the slope is an integer power of two and the bias is zero).

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

`numericType` — Target numeric type
`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

`indication` — Indication of whether target object represents fixed-point type with binary-point scaling
`1` | `0`

Indication of whether the target object represents a fixed-point type with binary-point scaling, returned as `1` (`true`) or `0` (`false`).

See Also

`Simulink.NumericType` | `isfixed` | `isscalingslopebias` | `isscalingunspecified`

Topics

“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b

isscalingslopebias

Package: Simulink

Determine whether numeric type represents a fixed-point data type with slope-and-bias scaling

Syntax

```
indication = isscalingslopebias(numericType)
```

Description

`indication = isscalingslopebias(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents a fixed-point data type with nontrivial slope-and-bias scaling and `0` (`false`) otherwise. A slope-and-bias fixed-point type has trivial scaling if the slope is an integer power of two and the bias is zero.

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

`numericType` — Target numeric type
`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

`indication` — Indication of whether target object represents fixed-point type with slope-and-bias scaling
`1` | `0`

Indication of whether the target object represents a fixed-point type with nontrivial slope-and-bias scaling, returned as `1` (`true`) or `0` (`false`).

See Also

`Simulink.NumericType` | `isfixed` | `isscalingbinarypoint` | `isscalingunspecified`

Topics

"Control Signal Data Types"
"Control Block Parameter Data Types"
"Control Data Type Names in Generated Code" (Embedded Coder)

Introduced in R2010b

isscalingunspecified

Package: Simulink

Determine whether numeric type represents a data type with unspecified scaling

Syntax

```
indication = isscalingunspecified(numericType)
```

Description

`indication = isscalingunspecified(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents a fixed-point or scaled double data type with unspecified scaling and `0` (`false`) otherwise.

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

`numericType` — Target numeric type
`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

`indication` — Indication of whether target object represents a type with unspecified scaling
`1` | `0`

Indication of whether the target object represents a type with unspecified scaling, returned as `1` (`true`) or `0` (`false`).

See Also

`Simulink.NumericType` | `isfixed` | `isscalingbinarypoint` | `isscalingslopebias`

Topics

"Control Signal Data Types"
"Control Block Parameter Data Types"
"Control Data Type Names in Generated Code" (Embedded Coder)

Introduced in R2010b

issingle

Package: Simulink

Determine whether numeric type represents the single-precision, floating-point data type `single`

Syntax

```
indication = issingle(numericType)
```

Description

`indication = issingle(numericType)` returns `1` (`true`) if the `Simulink.NumericType` object `numericType` represents the single-precision, floating-point data type `single` and `0` (`false`) otherwise.

In Simulink, a `Simulink.NumericType` object represents a data type that you can share between different data items in a model. For more information, see `Simulink.NumericType`.

Input Arguments

`numericType` — Target numeric type

`Simulink.NumericType` object

Target numeric type, specified as a `Simulink.NumericType` object.

Output Arguments

`indication` — Indication of whether target object represents `single`

`1` | `0`

Indication of whether the target object represents `single`, returned as `1` (`true`) or `0` (`false`).

See Also

`Simulink.NumericType` | `isdouble` | `isfloat`

Topics

“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b

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 “How Generated Code Stores Internal Signal, State, and Parameter Data” (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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Choose Storage Class for Controlling Data Representation in Generated Code” (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`

DataType — 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 `DataType` 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 `DataType` 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
- Mathematical expression (see “Set Variable Value by Using a Mathematical Expression”)

You can use MATLAB syntax to specify the value.

Example Expression	Description
`15.23`	Specifies a scalar value
`[3 4; 9 8]`	Specifies a matrix
`3+2i`	Specifies a complex value
`struct('A',20,'B',5)`	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").
`slexpr('myVar + myOtherVar')`	Specifies the expression `myVar + myOtherVar` where `myVar` and `myOtherVar` are other MATLAB variables or parameter objects. Simulink preserves this mathematical relationship between the object and the variables.

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)`

Example: `slexpr('myVar + myOtherVar')`

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.

Set Parameter Value to a Mathematical Expression

This example shows how to set the value of a parameter object, myParam, to the sum of two other variables, myVar and myOtherVar. With this technique, when you change the values of the independent variables, Simulink immediately calculates the new value of the parameter object.

1. Create the two independent variables.

   ```
   myVar = 5.2;
   myOtherVar = 9.8;
   ```
2. Create the parameter object.

   ```
   myParam = Simulink.Parameter;
   ```

3 Set the value of the parameter object to the expression `myVar + myOtherVar`.

```
myParam.Value = slexpr('myVar + myOtherVar')
```

See Also

`AUTOSAR.Parameter` | `Simulink.CoderInfo` | `Simulink.LookupTable` | `Simulink.Signal`

Topics

“Data Objects”
“Set Block Parameter Values”
“How Generated Code Stores Internal Signal, State, and Parameter Data” (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 run-time 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 5-433	Block's handle.
“CurrentTime” on page 5-433	Current simulation time.
“NumDworks” on page 5-433	Number of discrete work vectors used by the block.
“NumOutputPorts” on page 5-434	Number of block output ports.
“NumContStates” on page 5-434	Number of block's continuous states.
“NumDworkDiscStates” on page 5-434	Number of block's discrete states
“NumDialogPrms” on page 5-434	Number of parameters that can be entered on S-function block's dialog box.
“NumInputPorts” on page 5-435	Number of block's input ports.
“NumRuntimePrms” on page 5-435	Number of run-time parameters used by block.
“SampleTimes” on page 5-435	Sample times at which block produces outputs.

Method Summary

Name	Description
“ContStates” on page 5-436	Get a block's continuous states.
“DataTypeIsFixedPoint” on page 5-436	Determine whether a data type is fixed point.
“DatatypeName” on page 5-436	Get name of a data type supported by this block.
“DatatypeSize” on page 5-437	Get size of a data type supported by this block.
“Derivatives” on page 5-437	Get a block's continuous state derivatives.
“DialogPrm” on page 5-438	Get a parameter entered on an S-function block's dialog box.

Name	Description
“Dwork” on page 5-438	Get one of a block's DWork vectors.
“FixedPointNumericType” on page 5-439	Determine the properties of a fixed-point data type.
“InputPort” on page 5-439	Get one of a block's input ports.
“OutputPort” on page 5-440	Get one of a block's output ports.
“RuntimePrm” on page 5-440	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 “Create and Update S-Function 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
```

DataTypeIsFixedPoint

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 5-437

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 5-436

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 5-440

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 5-440

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 5-198 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 1x*n* character array 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 1x*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.CustomSnapshot class

Package: Simulink.sdi

Specify settings for a snapshot without opening or affecting the Simulation Data Inspector

Description

Use a `Simulink.sdi.CustomSnapshot` object to specify settings for a snapshot you want to create without opening the Simulation Data Inspector or affecting the open session. Creating a snapshot using a `Simulink.sdi.CustomSnapshot` object is the best option for fully scripted workflows. You can specify the snapshot dimensions in pixels, the subplot layout, and limits for the *x*- and *y*- axes. You can use the `clearSignals` and `plotOnSubplot` methods to plot signals you want to include in the snapshot. To capture the snapshot, you can pass the `Simulink.sdi.CustomSnapshot` object as the value for the `settings` name-value pair for the `Simulink.sdi.snapshot` function or use the `snapshot` method.

Construction

`snap = Simulink.sdi.CustomSnapshot` creates a `Simulink.sdi.CustomSnapshot` object.

Properties

`Width` — Image width
`600` (default) | scalar

Image width, in pixels.

Example: `750`

`Height` — Image height
`400` (default) | scalar

Image height, in pixels.

Example: `500`

Rows — Number of subplot rows

`1` (default) | scalar

Number of subplot rows, specified as a scalar between `1` and `8`, inclusive. Use `Rows` and `Columns` to set your desired subplot layout.

Example: `2`

`Columns` — Number of subplot columns

`1` (default) | scalar

Number of subplot columns, specified as a scalar between `1` and `8`, inclusive. Use `Rows` and `Columns` to set your desired subplot layout.

Example: `3`

`TimeSpan` — X-axis limits

`2x1` matrix

Limits for the time axis in the snapshot. The time axis limits are the same for all subplots. By default, the time axis adjusts to accommodate the largest time range of the plotted signals.

Example: `[0 20]`

YRange — Y-axis limits

cell array

Cell array of 1-by-2 matrices specifying the *y*-axis limits for all subplots in the custom snapshot. By default, `YRange` is `[-3 3]` for all subplots.

Example: `{[-10 10],[0 100]}`

Methods

clearSignals	Clear signals plotted on subplots of a `Simulink.sdi.CustomSnapshot` object
plotOnSubPlot	Plot signals on `Simulink.sdi.CustomSnapshot` object subplots
snapshot	Create a custom snapshot

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Copy View Settings to a Run

This example shows how to copy the view settings for aligned signals from one run to another.

Simulate Your Model and get Run Object

Simulate the `vdp` model to create a run of data to visualize.

```
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
sim('vdp')

runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify View Settings for Signals

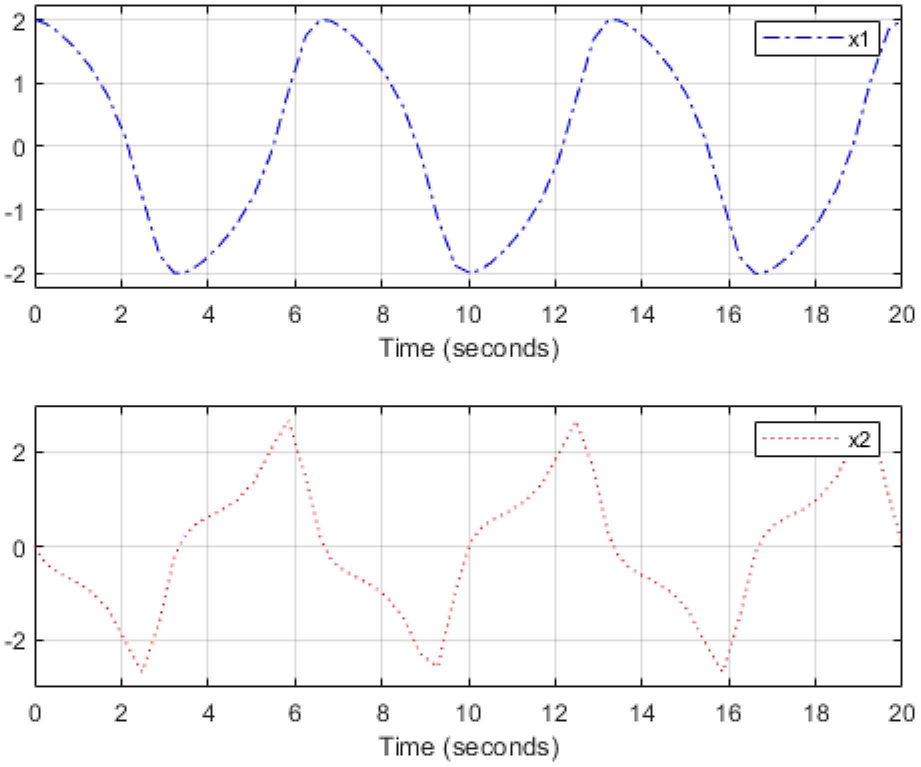
Use the `Simulink.sdi.Run` object to access the signals in the run. Then, modify the signals' view settings, and plot them in the Simulation Data Inspector. Open the Simulation Data Inspector and use `Simulink.sdi.snapshot` to view the results.

```
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

Capture a Snapshot from the Simulation Data Inspector

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```
snap = Simulink.sdi.CustomSnapshot;

snap.Rows = 2;
snap.YRange = {[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```



Copy the View Settings to a New Simulation Run

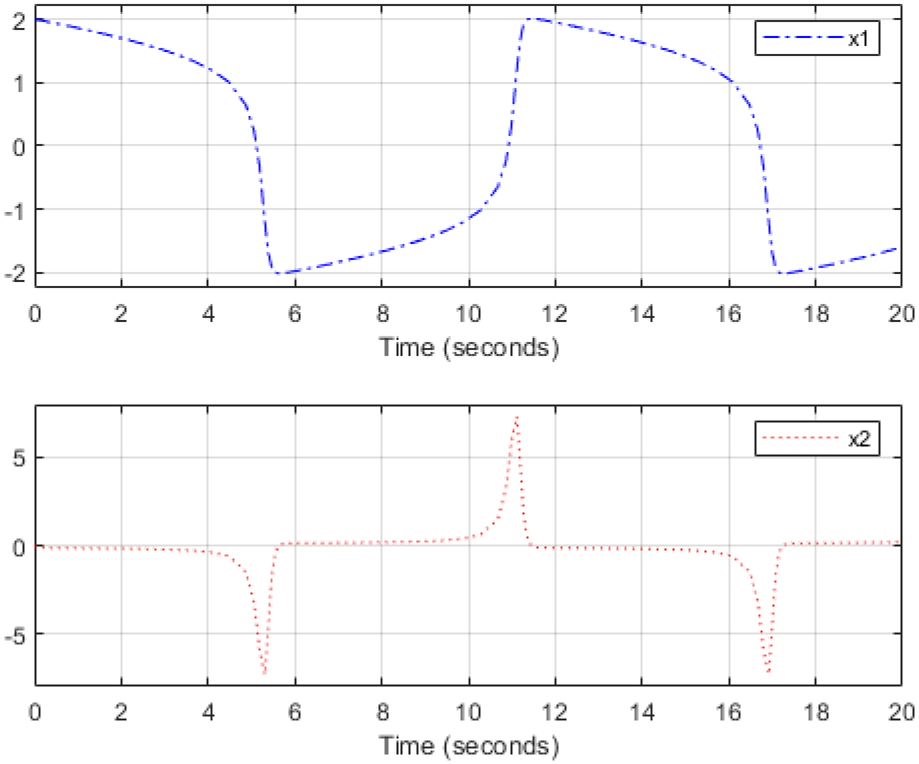
Simulate the model again, with a different Mu value. Then, visualize the new run by copying the view settings from the first run. Specify the plot input as true to plot the signals from the new run.

```
set_param('vdp/Mu','Gain','5')
sim('vdp')

runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);

sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,true);
```

Capture a Snapshot of the New Simulation Run

Use the Simulink.sdi.CustomSnapshot object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```
snap.clearSignals
snap.YRange = {[-2.25 2.25],[-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```



See Also

`Simulink.sdi.Signal` | `Simulink.sdi.snapshot`

Topics

"Inspect and Compare Data Programmatically"
"Create Plots Using the Simulation Data Inspector"

Introduced in R2018a

clearSignals

Class: Simulink.sdi.CustomSnapshot
Package: Simulink.sdi

Clear signals plotted on subplots of a `Simulink.sdi.CustomSnapshot` object

Syntax

```
snap.clearSignals
```

Description

`snap.clearSignals` clears plotted signals from all subplots in the `Simulink.sdi.CustomSnapshot` object, `snap`. Using the `clearSignals` method does not affect any subplots or signals in your open Simulation Data Inspector session.

Examples

Copy View Settings to a Run

This example shows how to copy the view settings for aligned signals from one run to another.

Simulate Your Model and get Run Object

Simulate the `vdp` model to create a run of data to visualize.

```
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
sim('vdp')

runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify View Settings for Signals

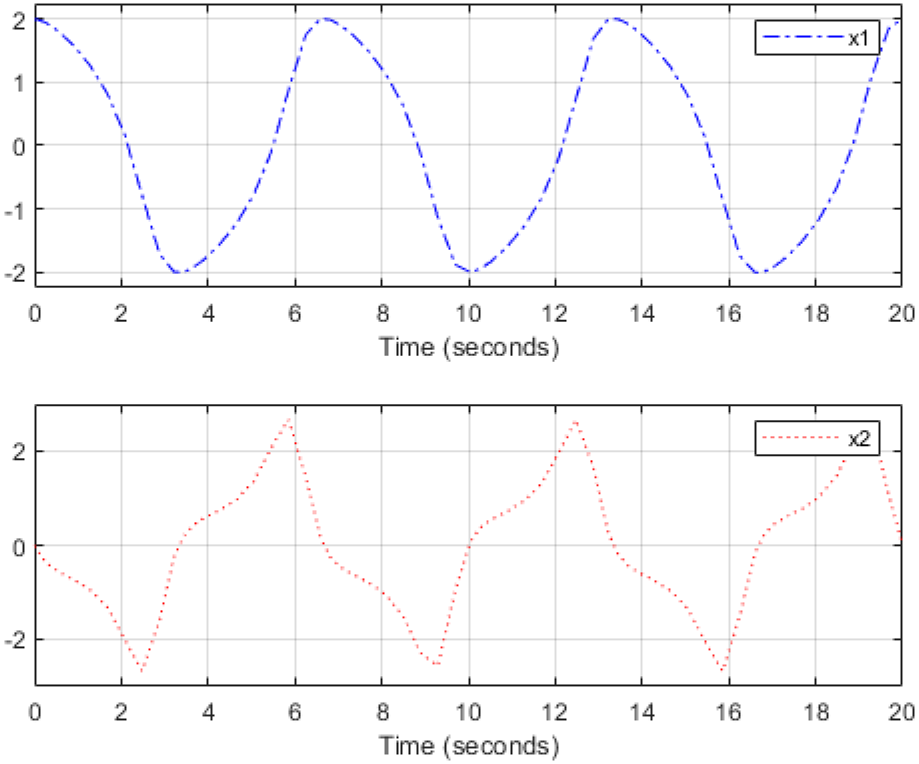
Use the Simulink.sdi.Run object to access the signals in the run. Then, modify the signals' view settings, and plot them in the Simulation Data Inspector. Open the Simulation Data Inspector and use Simulink.sdi.snapshot to view the results.

```
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

Capture a Snapshot from the Simulation Data Inspector

Create a Simulink.sdi.CustomSnapshot object and use the Simulink.sdi.snapshot function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```
snap = Simulink.sdi.CustomSnapshot;

snap.Rows = 2;
snap.YRange = {[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```



Copy the View Settings to a New Simulation Run

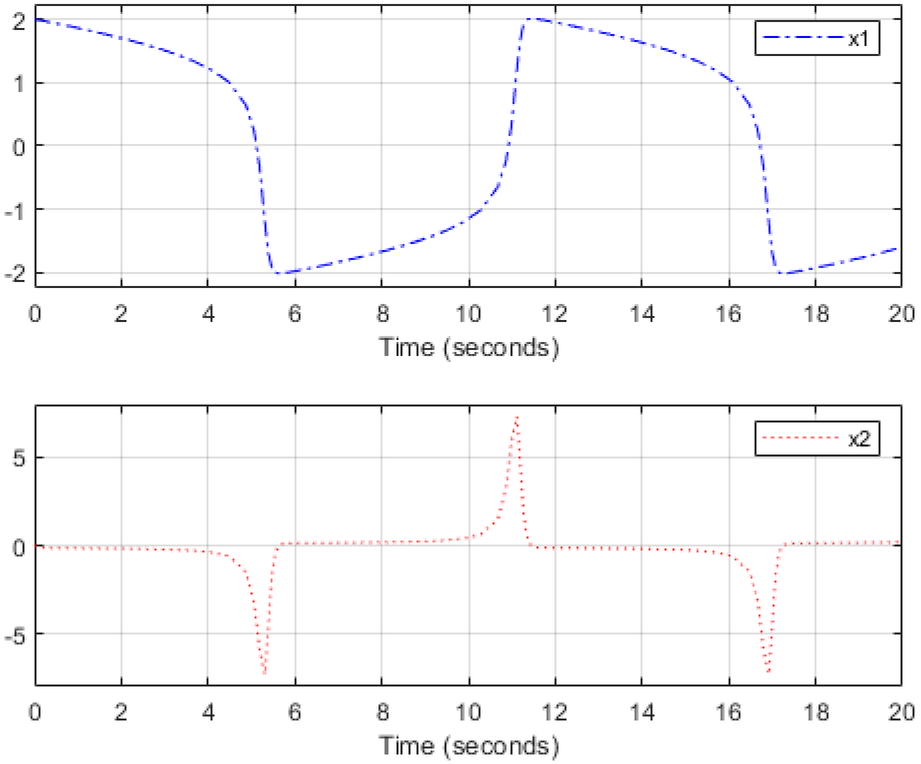
Simulate the model again, with a different `Mu` value. Then, visualize the new run by copying the view settings from the first run. Specify the `plot` input as `true` to plot the signals from the new run.

```
set_param('vdp/Mu','Gain','5')
sim('vdp')

runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);

sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,true);
```

Capture a Snapshot of the New Simulation Run

Use the `Simulink.sdi.CustomSnapshot` object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```
snap.clearSignals
snap.YRange = {[-2.25 2.25],[-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```



See Also

`Simulink.sdi.CustomSnapshot` | `Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.snapshot` | `plotOnSubPlot` | `snapshot`

Topics

"Inspect and Compare Data Programmatically"
"Create Plots Using the Simulation Data Inspector"

Introduced in R2018a

plotOnSubPlot

Class: Simulink.sdi.CustomSnapshot
Package: Simulink.sdi

Plot signals on `Simulink.sdi.CustomSnapshot` object subplots

Syntax

```
snap.plotOnSubPlot(row,column,signal,plot)
```

Description

`snap.plotOnSubPlot(row,column,signal,plot)` plots the `signal` on the subplot in the `Simulink.sdi.CustomSnapshot` object, `snap`, specified by `row` and `column` when `plot` is `true`. When `plot` is `false`, `plotOnSubPlot` clears the `signal` from the subplot.

Input Arguments

`row` — Subplot row

scalar

Row for subplot on which you want to plot a signal. Specify `row` as a value from 1 through 8, inclusive.

Example: 2

`column` — Subplot column

scalar

Column for subplot on which you want to plot a signal. Specify `column` as a value from 1 through 8, inclusive.

Example: 3

signal — Signal to plot
Simulink.sdi.Signal | signal ID

Signal ID or Simulink.sdi.Signal object corresponding to the signal you want to plot.

Example: sigID

plot — Plot indicator
logical

Logical indicator of whether to plot or clear the signal from the subplot.

- true – Plot the signal.
- false – Clear the signal.

Example: true

Data Types: logical

Examples

Copy View Settings to a Run

This example shows how to copy the view settings for aligned signals from one run to another.

Simulate Your Model and get Run Object

Simulate the vdp model to create a run of data to visualize.

```
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
sim('vdp')

runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify View Settings for Signals

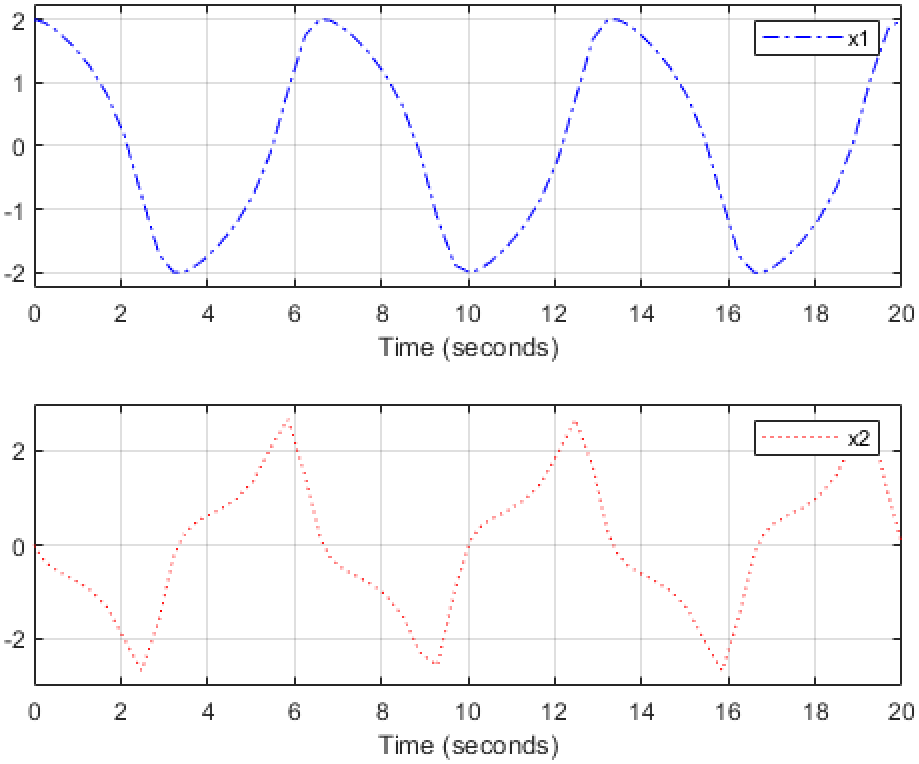
Use the Simulink.sdi.Run object to access the signals in the run. Then, modify the signals' view settings, and plot them in the Simulation Data Inspector. Open the Simulation Data Inspector and use Simulink.sdi.snapshot to view the results.

```
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

Capture a Snapshot from the Simulation Data Inspector

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```
snap = Simulink.sdi.CustomSnapshot;

snap.Rows = 2;
snap.YRange = {[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```



Copy the View Settings to a New Simulation Run

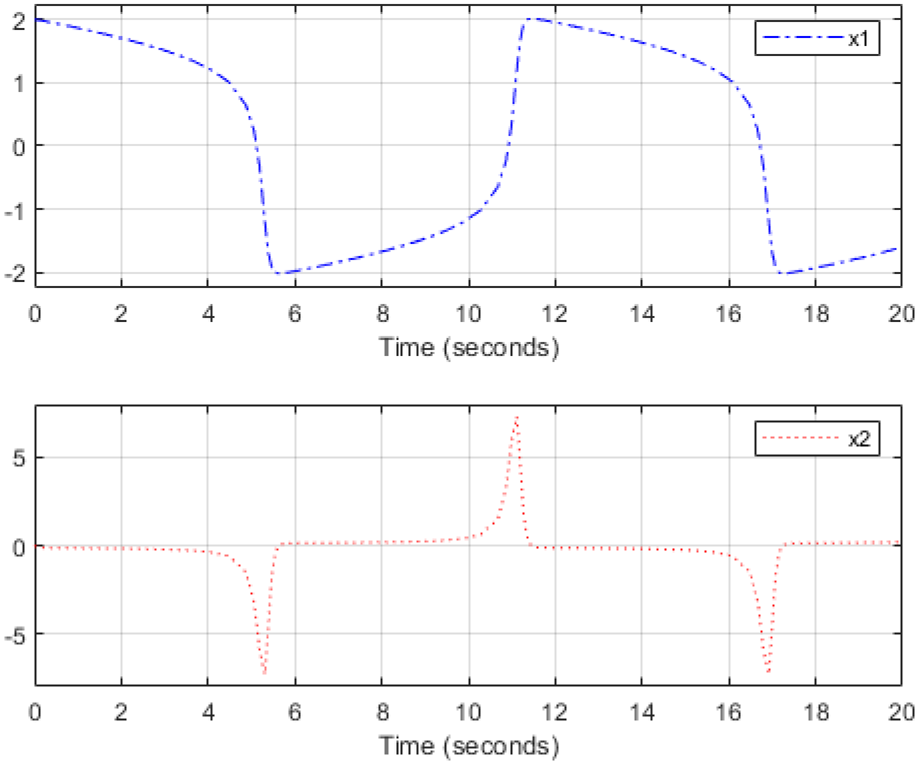
Simulate the model again, with a different `Mu` value. Then, visualize the new run by copying the view settings from the first run. Specify the `plot` input as `true` to plot the signals from the new run.

```
set_param('vdp/Mu','Gain','5')
sim('vdp')

runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);

sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,true);
```

Capture a Snapshot of the New Simulation Run

Use the `Simulink.sdi.CustomSnapshot` object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```
snap.clearSignals
snap.YRange = {[-2.25 2.25],[-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```



See Also

`Simulink.sdi.CustomSnapshot` | `Simulink.sdi.Signal` | `Simulink.sdi.snapshot` | `clearSignals` | `snapshot`

Topics

"Inspect and Compare Data Programmatically"
"Create Plots Using the Simulation Data Inspector"

Introduced in R2018a

snapshot

Class: Simulink.sdi.CustomSnapshot
Package: Simulink.sdi

Create a custom snapshot

Syntax

```
fig = snap.snapshot
[fig,image] = snap.snapshot
fig = snap.snapshot(Name,Value)
[fig,image] = snap.snapshot
```

Description

`fig = snap.snapshot` creates a figure according to the properties of the `Simulink.sdi.CustomSnapshot` object, `snap`, and returns the handle for the figure, `fig`.

`[fig,image] = snap.snapshot` creates a figure according to the properties of the `Simulink.sdi.CustomSnapshot` object, `snap`, and returns the handle for the figure, `fig`, and an array of image data, `image`.

`fig = snap.snapshot(Name,Value)` creates a figure according to the properties of the `Simulink.sdi.CustomSnapshot` object, `snap`, with additional options specified by one or more `Name,Value` pair arguments. This syntax returns the figure handle, `fig`.

`[fig,image] = snap.snapshot` creates a figure according to the properties of the `Simulink.sdi.CustomSnapshot` object, `snap`, with additional options specified by one or more `Name,Value` pair arguments. This syntax returns the figure handle, `fig`, and an array of image data, `image`.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'to','figure','props',{'Name','My Data'}`

`to` — Type of snapshot to create

`'image'` (default) | `'figure'` | `'file'` | `'clipboard'`

Type of snapshot to create.

- `'image'` — Create a figure and return the figure handle and an array of image data. When you specify `'to','image'`, the `fig` and `image` outputs both have value.
- `'figure'` — Create a figure and return the figure handle. When you specify `'to','figure'` the `fig` output has value, and the `image` output is empty.
- `'file'` — Save to a PNG file with the name specified by the `filename` name-value pair. If you do not specify a `filename` name-value pair, the file is named `plots.png`. When you specify `'to','file'`, the `fig` and `image` outputs are both empty.
- `'clipboard'` — Copy the plots to your system clipboard. From the clipboard, you can paste the image into another program such as Microsoft Word. When you specify `'to','clipboard'`, the `fig` and `image` outputs are both empty.

Example: `'to','file'`

Data Types: `char` | `string`

`filename` — Name for image file

`'plots.png'` (default) | character array | string

Name of the image file to store the snapshot when you specify `'to','file'`.

Example: `'filename','MyImage.png'`

Data Types: `char` | `string`

`props` — Properties to customize the figure

cell array

Figure properties, specified as a cell array. You can include settings for the figure properties described in Figure Properties.

Example: `'props',{'Name','MyData','NumberTitle','off'}`

Data Types: `char` | `string`

Output Arguments

`fig` — Figure handle

figure handle

Handle for the figure. When a figure is not created with your specified options, the `fig` output is empty.

`image` — Image data

array

Array of image data. The `image` output has value when you use `Simulink.sdi.snapshot` without any input arguments or without a `to` name-value pair and when you specify `'to','image'`.

Examples

Copy View Settings to a Run

This example shows how to copy the view settings for aligned signals from one run to another.

Simulate Your Model and get Run Object

Simulate the `vdp` model to create a run of data to visualize.

```
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
sim('vdp')

runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify View Settings for Signals

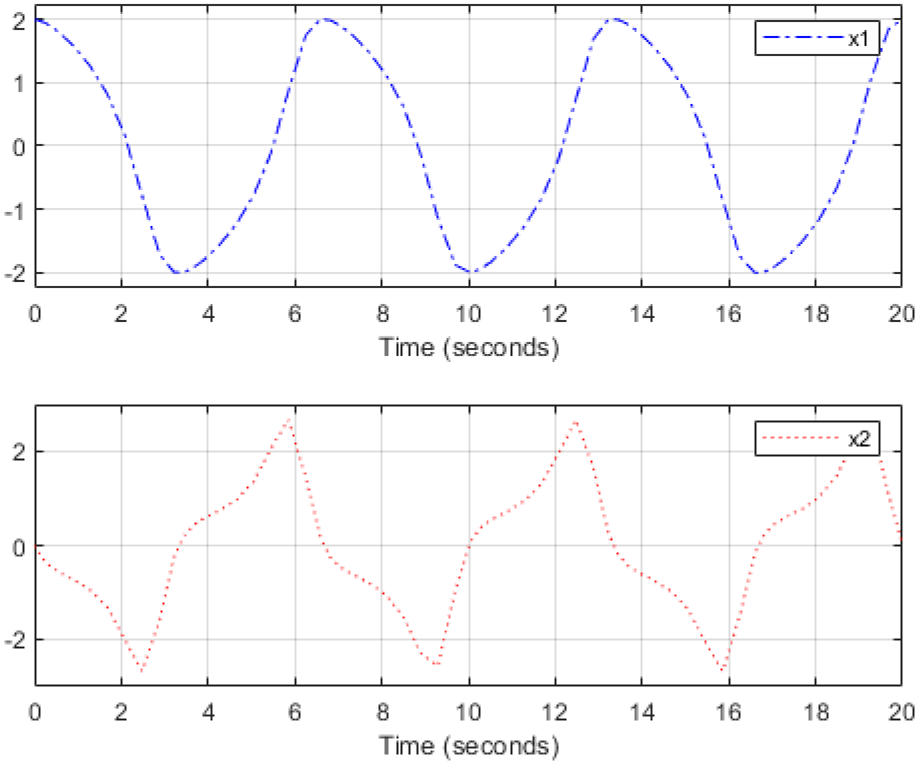
Use the `Simulink.sdi.Run` object to access the signals in the run. Then, modify the signals' view settings, and plot them in the Simulation Data Inspector. Open the Simulation Data Inspector and use `Simulink.sdi.snapshot` to view the results.

```
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

Capture a Snapshot from the Simulation Data Inspector

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```
snap = Simulink.sdi.CustomSnapshot;

snap.Rows = 2;
snap.YRange = {[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```



Copy the View Settings to a New Simulation Run

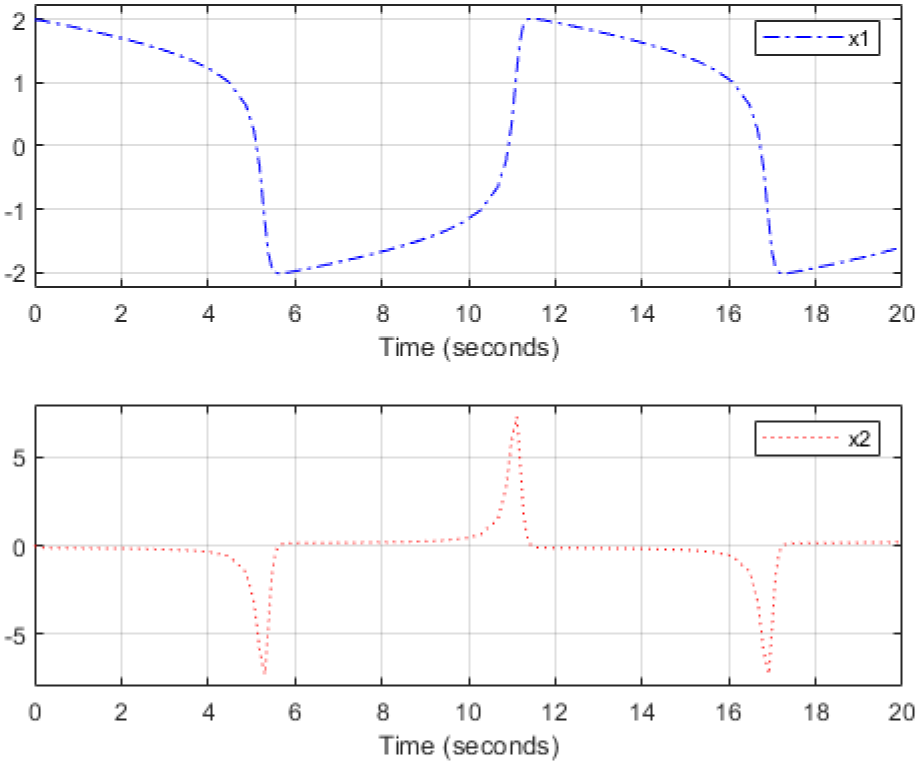
Simulate the model again, with a different `Mu` value. Then, visualize the new run by copying the view settings from the first run. Specify the `plot` input as `true` to plot the signals from the new run.

```
set_param('vdp/Mu','Gain','5')
sim('vdp')

runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);

sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,true);
```

Capture a Snapshot of the New Simulation Run

Use the `Simulink.sdi.CustomSnapshot` object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```
snap.clearSignals
snap.YRange = {[-2.25 2.25],[-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```



See Also

Figure Properties | `Simulink.sdi.CustomSnapshot` | `Simulink.sdi.snapshot` | `clearSignals` | `plotOnSubPlot`

Topics

“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2018a

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.DatsetRef` 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

compare	Compare runs with DatasetRef objects
getAsDatastore	Retrieve element as sdidatastore object
getElement	Retrieve DatasetRef element by index
getElementNames	Get character vectors of element names
getSignal	Return Signal object
plot	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_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
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.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

outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]



% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% 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
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.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
```

```
outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]
```

```
% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% 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 sdidatastore object

Syntax

```
[elementDatastore, name, index] = SDIDatasetRef.getAsDatastore(arg)
```

Description

`[elementDatastore, name, index] = SDIDatasetRef.getAsDatastore(arg)` returns the requested element as a `matlab.io.datastore.sdidatastore`object, 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.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`

Output Arguments

`elementDatastore` — Element as sdidatastore
sdidatastore object

Element as `matlab.io.datastore.sdidatastore` 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 sdidatastore Object for a Signal

This example shows how to create a `sdidatastore` 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
fuelRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = fuelRun.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames
```

```
names =

  15x1 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'                         }
    {'ego_sw'                       }
    {'engine_speed'                 }
    {'speed_sw'                     }
    {'map_sw'                       }
    {'throttle_sw'                  }

% Get sdidatastore object for fuel signal
fuel_ds = run_DSRef.getAsDatastore(10);
```

Alternatives

You can construct a `sdidatastore` object for a specified signal using `matlab.io.datastore.sdidatastore`.

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.DatasetRef.getElement` | `matlab.io.datastore.SimulationDatastore` | `matlab.io.datastore.sdidatastore`

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
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.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
```

```
outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]


% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% 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_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
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.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
```

```
outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]
```

```
% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% 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_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
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.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
```

```
outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]
```

```
% Get slp signal
```

```
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% 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

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_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
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.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

outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]



% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% 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 “View Data with the Simulation Data Inspector”.

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.

Setup

Load the model, and mark the `q, rad/sec` and `alpha, rad` signals for logging. Then, simulate the model to create the baseline run.

```
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Simulate system
sim('slexAircraftExample')
```

Modify Model Parameter

Modify the model parameter `Ts` in the model workspace to change the time constant of the input low-pass filter.

```
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',1)

% Simulate again
sim('slexAircraftExample')
```

Compare Runs and Inspect Results

Use the `Simulink.sdi.compareRuns` function to compare the data from the simulations. Then, inspect the `match` property of the signal result to see whether the signals fell within the default tolerance of 0.

```
% 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
```

```
sig1Result1 = diffRun1.getResultByIndex(1);
sig2Result1 = diffRun1.getResultByIndex(2);

% Check whether signals matched
sig1Result1.Match

ans = logical
   0


sig2Result1.Match

ans = logical
   0
```

Compare Runs with Signal Tolerances

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

```
% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sigID2 = run1.getSignalIDByIndex(2);

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

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = diffRun2.getResultByIndex(1);
sig2Result2 = diffRun2.getResultByIndex(2);
```

```
% Check the result
sig1Result2.Match

ans = logical
   1


sig2Result2.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.

Setup

Load the model, and mark the `q, rad/sec` and `alpha, rad` signals for logging. Then, simulate the model to create the baseline run.

```
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on

% Simulate system
sim('slexAircraftExample')
```

Modify Model Parameter

Modify the model parameter `Ts` in the model workspace to change the time constant of the input low-pass filter.

```
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',1)

% Simulate again
sim('slexAircraftExample')
```

Compare Runs and Inspect Results

Use the `Simulink.sdi.compareRuns` function to compare the data from the simulations. Then, inspect the `match` property of the signal result to see whether the signals fell within the default tolerance of 0.

```
% 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
sig1Result1 = diffRun1.getResultByIndex(1);
sig2Result1 = diffRun1.getResultByIndex(2);

% Check whether signals matched
sig1Result1.Match

ans = logical
   0


sig2Result1.Match

ans = logical
   0
```

Compare Runs with Signal Tolerances

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

```
% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sigID2 = run1.getSignalIDByIndex(2);

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

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;
```

```
% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = diffRun2.getResultByIndex(1);
sig2Result2 = diffRun2.getResultByIndex(2);

% Check the result
sig1Result2.Match

ans = logical
   1


sig2Result2.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".

Tol — 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 the input and output signals for a control system. 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
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = aircraftRun.getSignalIDByIndex(1);
signalID2 = aircraftRun.getSignalIDByIndex(2);

if (aircraftRun.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (aircraftRun.isValidSignalID(signalID1) && aircraftRun.isValidSignalID(signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signals match within tolerance
    match = sigDiff.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

add	Add signals to run
create	Create a Simulink.sdi.Run object
export	Export run to Simulink.SimulationData.Dataset object
getDatasetRef	Create a Simulink.sdi.DatasetRef object for a run
getSignalByIndex	Get `Simulink.sdi.Signal` object by index
getSignalIDByIndex	Return signal ID for signal at index
isValidSignalID	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 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
fuelRun = Simulink.sdi.getRun(runID);

% Check signal count of the run
fuelRun.signalCount
```

```
ans = int32
    15

% Get signal objects for the signals in the run
signal1 = fuelRun.getSignalByIndex(4);
signal2 = fuelRun.getSignalByIndex(9);
signal3 = fuelRun.getSignalByIndex(10);

% Create subplot 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
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.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 = sinusoidsRun.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 = sinusoidsRun.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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an `spmd` code block, load the `slexAircraftExample` model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use `parfor` to run the seven simulations in parallel. Select the value for `Ts` for each simulation, and modify the value of `Ts` in the model workspace. Then, run the simulation and build an array of `Simulink.sdi.WorkerRun` objects to access the data with the Simulation Data Inspector. After the `parfor` loop, use another `spmd` segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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 “View Data with the Simulation Data Inspector”.

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 that the data comes from a file.
- `'vars'` indicates that 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
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.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 = sinusoidsRun.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 = sinusoidsRun.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

"View Data with the Simulation Data Inspector"

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
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.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 = sinusoidsRun.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 = sinusoidsRun.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 run in the Simulation Data Inspector to a `Simulink.SimulationData.Dataset` object in the base workspace you can use to further process your data. The method you choose to export your run depends on the processing you do in your script. If you have a run object for the run, you can use the `export` method to create a `Simulink.SimulationData.Dataset` object with the run's data in the base workspace. If you do not have a run object, use the `Simulink.sdi.exportRun` function to export the run to the workspace.

Export Run Using Simulink.sdi.exportRun

Use the `Simulink.sdi.export` function when your workflow does not include creating a run object.

```
% Load vdp model
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 vdp model
sim('vdp')

% Get run ID for simulation run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Export run
simDataset = Simulink.sdi.exportRun(runID);
```

Export Run Using export Method

When you already have a `Simulink.sdi.Run` object for your run, you can use the `export` method to create a `Simulink.SimulationData.Dataset` object in the base workspace for further processing of the data.

```
% Load vdp model
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);
vdpRun = Simulink.sdi.getRun(runID);

% Get signal ids for signals
sigID1 = vdpRun.getSignalIDByIndex(1);
sigID2 = vdpRun.getSignalIDByIndex(2);

% Compare signals
diffResult = Simulink.sdi.compareSignals(sigID1,sigID2);
diffResult.match
```

```
ans = logical
   0
```

```
% Export run
simDataset = vdpRun.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_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
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.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

outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]



% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% 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 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
fuelRun = Simulink.sdi.getRun(runID);

% Check signal count of the run
fuelRun.signalCount

ans = int32
    15

% Get signal objects for the signals in the run
signal1 = fuelRun.getSignalByIndex(4);
signal2 = fuelRun.getSignalByIndex(9);
signal3 = fuelRun.getSignalByIndex(10);

% Create subplot 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 the input and output signals for a control system. 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
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = aircraftRun.getSignalIDByIndex(1);
signalID2 = aircraftRun.getSignalIDByIndex(2);

if (aircraftRun.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (aircraftRun.isValidSignalID(signalID1) && aircraftRun.isValidSignalID(signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signals match within tolerance
    match = sigDiff.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 the input and output signals for a control system. 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,'o

% Simulate model slexAircraftExample
sim('slexAircraftExample')

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

% Get Run object
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = aircraftRun.getSignalIDByIndex(1);
signalID2 = aircraftRun.getSignalIDByIndex(2);

if (aircraftRun.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (aircraftRun.isValidSignalID(signalID1) && aircraftRun.isValidSignalID(signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signals match within tolerance
    match = sigDiff.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 generated by the Simulation Data Inspector. You can get the signal ID for a signal using methods of the `Simulink.sdi.Run` object or using the `Simulink.sdi.getSignal` function.

`index` — Signal index within run
integer

Index of the signal within the run.

Example: `1`

Properties

Signal Properties (read only)

`ID` — Signal identifier
integer

Unique number identifying the signal.

Example: `1330`

`RunID` — Run identifier
integer

Run identifier for the run that contains the signal.

Example: `1402`

`Name` — Signal name
character vector

Name of the signal.

Example: `'fuel'`

`Units` — Signal measurement units
character vector

Signal units of measure.

Example: `'g/s'`

`Data Type` — Data type for signal data
character vector

Data type of signal data.

Example: `'double'`

`Complexity` — Complexity of signal data
character vector

Specifies whether signal data is real or complex.

Example: `'real'`

`SampleTime` — Signal sample time

character vector

Signal sample time. A value of `'Continuous'` indicates a variable-step simulation.

Example: `'Continuous'`

Example: `0.1`

`Model` — Name of model that produced signal

character vector

Name of the model that produced the signal.

Example: `'sldemo_fuelsys'`

`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`.

Example: `'sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion/MinMax'`

`PortIndex` — Block port index

integer

Index of the output port that defines the signal.

Example: 1

`Dimensions` — Dimensions of matrix containing signal

integer array

Dimensions of the matrix that contains the signal.

Example: `[1]`

`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 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

`ComplexFormat` — Display format for complex signals

`"real-imaginary"` | `"magnitude"` | `"magnitude-phase"` | `"phase"`

Complex format specifying how to display complex signal data in the Simulation Data Inspector.

- `"real-imaginary"` — The real and imaginary components of the signal display together when you plot the signal. The imaginary component of the signal is plotted with a different shade of the **Line Color**.

- `"magnitude"` — The magnitude of the signal displays when you plot the signal.
- `"magnitude-phase"` — The magnitude and phase of the signal display together when you plot the signal.
- `"phase"` — The phase of the signal displays when you plot the signal. The phase is plotted with a different shade of the **Line Color**.

Data Types: `char` | `string`

`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-by-3` vector

Color of signal in plots, specified as a 1-by-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 comparisons. The Simulation Data Inspector uses tolerances specified in the signal properties of the baseline signal when **Override Global Tol** is set to `yes`. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: `0.1`

Data Types: `double`

`RelTol` — Relative tolerance

0 (default) | double

Positive-valued relative tolerance for the signal used for signal comparisons. The Simulation Data Inspector uses tolerances specified in the signal properties of the baseline signal when **Override Global Tol** is set to `yes`. The relative tolerance is expressed as a fractional multiplier. For example, `0.1` specifies a 10 percent tolerance. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: `0.05`

Data Types: `double`

`TimeTol` — Time tolerance

0 (default) | double

Positive-valued time tolerance for the signal used in signal comparisons. The Simulation Data Inspector uses tolerances specified in the signal properties of the baseline signal when **Override Global Tol** is set to `yes`. Specify the time tolerance in seconds. For more information about tolerances in the Simulation Data Inspector, 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

convertUnits	Convert signal units
export	Export signal object to MATLAB timeseries
getAsTall	Return tall timetable with time and data values
plotOnSubPlot	Plot signal on specified sub-plot

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.

Setup

Load the model, and mark the `q, rad/sec` and `alpha, rad` signals for logging. Then, simulate the model to create the baseline run.

```
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on
% Simulate system
sim('slexAircraftExample')
```

Modify Model Parameter

Modify the model parameter `Ts` in the model workspace to change the time constant of the input low-pass filter.

```
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',1)

% Simulate again
sim('slexAircraftExample')
```

Compare Runs and Inspect Results

Use the `Simulink.sdi.compareRuns` function to compare the data from the simulations. Then, inspect the `match` property of the signal result to see whether the signals fell within the default tolerance of 0.

```
% 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
sig1Result1 = diffRun1.getResultByIndex(1);
sig2Result1 = diffRun1.getResultByIndex(2);

% Check whether signals matched
sig1Result1.Match
```

```
ans = logical
   0
```

```
sig2Result1.Match
```

```
ans = logical
   0
```

Compare Runs with Signal Tolerances

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

```
% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sigID2 = run1.getSignalIDByIndex(2);

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

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = diffRun2.getResultByIndex(1);
sig2Result2 = diffRun2.getResultByIndex(2);

% Check the result
sig1Result2.Match
```

```
ans = logical
   1
```

```
sig2Result2.Match
```

```
ans = logical
   1
```

Define Comparison and Visualization Properties for a Signal

This example shows how to obtain a `Simulink.sdi.Signal` object and modify its properties using the Simulation Data Inspector programmatic interface.

Acquire a `Simulink.sdi.Signal` Object

First, run a simulation to create a run. This example uses example model `slexAircraftExample`. Then, use the Simulation Data Inspector programmatic interface to get the `Simulink.sdi.Signal` object for your signal of interest.

```
% 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 sign
% 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));
```

Modify the Signal Properties

The `Simulink.sdi.Signal` object has several comparison and visualization properties that you can modify.

```
% 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 Properties

You can view the signal properties in the command window and in the Simulation Data Inspector to verify that the signal has its properties defined how you want them.

```
signalObj
```

```
signalObj = 
  Signal with properties:

               ID: 45696
            RunID: 45685
             Name: 'Integrate:CSTATE'
            Units: ''
         DataType: 'double'
       Complexity: "real"
    ComplexFormat: "real-imaginary"
```

```
         SampleTime: ''
              Model: 'slexAircraftExample'
          BlockPath: 'slexAircraftExample/Aircraft↵Dynamics↵Model/Vertical Channel/Integ
      FullBlockPath: 'slexAircraftExample/Aircraft Dynamics Model/Vertical Channel/Integ
          PortIndex: 0
         Dimensions: 1
            Channel: [1×0 int32]
            Checked: 1
          LineColor: [1 0.4000 0.6000]
         LineDashed: '-'
       InterpMethod: 'linear'
             AbsTol: 0
             RelTol: 0
            TimeTol: 0
         SyncMethod: 'intersection'
             Values: [1×1 timeseries]
         RootSource: 'simOut.get('xout')'
         TimeSource: 'simOut.get('xout').time'
         DataSource: 'simOut.get('xout').signals(1).values'
```

```
Simulink.sdi.view
```

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of the input and output signals for a control system. 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
aircraftRun = Simulink.sdi.getRun(runID);
```

```
% Get signal IDs
signalID1 = aircraftRun.getSignalIDByIndex(1);
signalID2 = aircraftRun.getSignalIDByIndex(2);

if (aircraftRun.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (aircraftRun.isValidSignalID(signalID1) && aircraftRun.isValidSignalID(signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signals match within tolerance
    match = sigDiff.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 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
fuelRun = Simulink.sdi.getRun(runID);

% Check signal count of the run
fuelRun.signalCount
```

```
ans = int32
    15
```

```
% Get signal objects for the signals in the run
signal1 = fuelRun.getSignalByIndex(4);
signal2 = fuelRun.getSignalByIndex(9);
signal3 = fuelRun.getSignalByIndex(10);

% Create subplot 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

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

convertUnits

Class: Simulink.sdi.Signal
Package: Simulink.sdi

Convert signal units

Syntax

```
signal.convertUnits(units)
```

Description

`signal.convertUnits(units)` converts the units of `signal` to the units specified by `units`. For a list of acceptable units, see Allowed Units. You can use `Simulink.sdi.Signal.convertUnits` to convert the units on `Simulink.sdi.Signal` objects with data of all built-in and fixed-point types.

Input Arguments

`units` — Desired signal units

string | character vector

Desired units for the signal.

Example: `'m'`

Example: `"ft/s"`

Data Types: `char` | `string`

Examples

Programmatically Convert Signal Units

This example shows how to use the `convertUnits` method to convert the units of a `Simulink.sdi.Signal` object, using the model `sldemo_autotrans`.

Generate Simulation Data

Simulate the model to create a run of data. Then, use the Simulation Data Inspector programmatic interface to get the run data.

```
% Simulate the model
sim('sldemo_autotrans')

% Get a Simulink.sdi.Run object for the most recently created run
ids = Simulink.sdi.getAllRunIDs;
id = ids(end);
transRun = Simulink.sdi.getRun(id);
```

Inspect the Signal Properties

Get a `Simulink.sdi.Signal` object for the `EngineRPM` signal and inspect its properties to determine the units.

```
% Get Simulink.sdi.Signal object
signal = transRun.getSignalByIndex(1)
```

```
signal = 
  Signal with properties:

               ID: 22254
            RunID: 22291
             Name: 'EngineRPM'
            Units: 'rpm'
         DataType: 'double'
       Complexity: "real"
    ComplexFormat: "real-imaginary"
       SampleTime: '0.04'
            Model: 'sldemo_autotrans'
        BlockPath: 'sldemo_autotrans/Engine'
    FullBlockPath: 'sldemo_autotrans/Engine'
        PortIndex: 1
       Dimensions: 1
```

```
          Channel: [1x0 int32]
          Checked: 0
        LineColor: [0.9290 0.6940 0.1250]
       LineDashed: '-'
      InterpMethod: 'linear'
           AbsTol: 0
           RelTol: 0
          TimeTol: 0
       SyncMethod: 'union'
           Values: [1x1 timeseries]
       RootSource: ''
       TimeSource: ''
       DataSource: ''
```

Convert Signal Units

Use the `convertUnits` method to convert the `EngineRPM` signal units to rad/s. Then, change the signal name to reflect the new units.

```
% Convert units
signal.convertUnits('rad/s')

signal.Name = 'Engine,rad/s';
```

See Also

`Simulink.sdi.Signal`

Topics

"Inspect and Compare Data Programmatically"
"Unit Specification in Simulink Models"
"Units in Simulink"

Introduced in R2018a

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
fuelRun = Simulink.sdi.getRun(runID);

% Get a signal
signal = fuelRun.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
fuelRun = Simulink.sdi.getRun(runID);

% Get a signal
signal = fuelRun.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 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
fuelRun = Simulink.sdi.getRun(runID);

% Check signal count of the run
fuelRun.signalCount

ans = int32
    15

% Get signal objects for the signals in the run
signal1 = fuelRun.getSignalByIndex(4);
signal2 = fuelRun.getSignalByIndex(9);
signal3 = fuelRun.getSignalByIndex(10);

% Create subplot 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

getDataset	Create Dataset of worker run data
getDatasetRef	Create DatasetRef for worker run
getLatest	Create worker run for latest run
getLocalRun	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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an `spmd` code block, load the `slexAircraftExample` model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use `parfor` to run the seven simulations in parallel. Select the value for `Ts` for each simulation, and modify the value of `Ts` in the model workspace. Then, run the simulation and build an array of `Simulink.sdi.WorkerRun` objects to access the data with the Simulation Data Inspector. After the `parfor` loop, use another `spmd` segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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 outport 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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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 outport 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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an `spmd` code block, load the `slexAircraftExample` model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an `spmd` code block, load the `slexAircraftExample` model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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 programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an `spmd` code block, load the `slexAircraftExample` model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```
p = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Create temporary directory on each worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end
```

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % 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

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)

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 3: slexAircraftExample'
            Run: [1×1 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 “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Choose Storage Class for Controlling Data Representation in Generated Code” (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`

`DataType` — 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 `MxN` matrix signal.

To use symbolic dimensions, specify a character vector.

Example: `[1 3]`

Example: `'[1 myDimParam]'`

Data Types: `double` | `char`

`DimensionsMode` — Dimension mode of signal

`'auto'` (default) | `'Fixed'` | `'Variable'`

Dimensions mode of the signal. Valid values are:

- `'auto'`— Allows variable-size and fixed-size signals.
- `'Fixed'`—Allows only fixed-size signals. Does not allow variable-size signals.
- `'Variable'`—Allows only variable-size signals.

For information about variable-size signals, see “Variable-Size Signal Basics”.

`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.

- Outport 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”
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (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 5-205	Individual component within the block specified by the block path

Method Summary

Name	Description
`BlockPath` on page 5-205	Create a block path.
`convertToCell` on page 5-208	Convert a block path to a cell array of character vectors.
`getBlock` on page 5-209	Get a single block path in the model reference hierarchy.
`getLength` on page 5-210	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_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
```

The resulting block path reflects the model reference hierarchy for the block path.

```
bp1 =

  Simulink.BlockPath
  Package: Simulink

  Block Path:
    'sldemo_mdlref_depgraph/thermostat'
      'sldemo_mdlref_heater/Fahrenheit to Celsius'
        'sldemo_mdlref_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_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
cellarray_for_bp1 = bp1.convertToCell()
```

The result is a cell array representing the elements of the block path.

```
cellarray_for_bp1 =

    'sldemo_mdlref_depgraph/thermostat'
    'sldemo_mdlref_heater/Fahrenheit to Celsius'
    'sldemo_mdlref_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_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_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_mdlref_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_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
 'sldemo_mdlref_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 “Loading Data Arrays to Root-Level Inputs”.
  - An `nx1` 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 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, `logsout` 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

addElement	Add element to end of data set
concat	Concatenate dataset to another dataset
get	Get element or collection of elements from dataset
getElementNames	Return names of all elements in dataset
find	Get element or collection of elements from dataset
numElements	Get number of elements in data set
plot	Plot dataset elements in Signal Preview window or Simulation Data Inspector
setElement	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 `logsout` signal logging dataset that has three elements.

Get the second element of the `logsout` dataset.

```
logsout{2}
```

Change the name of the third element.

```
logsout{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);
logsout{4} = element4
```

See Also

| `Simulink.ModelDataLogs` | `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 (logsout) 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','logsout');
```

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, `'logsout'` 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
`numElements`	Get number of elements from dataset
`getElementNames`	Return names of all elements in dataset
`get` The `get` method is an alias for the `getElement` method. **Note** You can use curly braces to streamline indexing syntax to access elements in a dataset reference, instead of using `get` or `getElement` 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 `get` and `getElement` methods support specifying an element by name or block path, as well as by index.	Get element from dataset
`getAsDatastore`	Get `matlab.io.datastore.SimulationDatastore` representation of element from a `DatasetRef` object
getDatasetVariableNames on page 2-889	List names of `Dataset` variables in MAT-file

Method	Purpose
`plot`	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

  'logsout'   '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 `logsout`.

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat.','logsout')
```

```
sigLogRef =

  Simulink.SimulationData.DatasetRef
  Characteristics:
          Location: out.mat (/my_files/out.mat)
        Identifier: logsout
  Resolved Dataset: 'logsout' 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.

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 5-618	Location of Data Store Memory block for the logged data store
`DSMWriterBlockPaths` on page 5-619	Location of Data Store Write blocks that write to the data store
`DSMWriters` on page 5-619	Data Store Write blocks for each signal value
`Name` on page 5-619	Name of the data store dataset
`Scope` on page 5-620	Scope of the data store: `'local'` or `'global'`
`Values` on page 5-620	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 `n`th element of `DSMWriters` contains the index of the element in `DSMWriterBLockPaths` that contains the block path of the writer that performed the `n`th 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.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
`DataLogging` on page 5-622	Signal logging mode.
`NameMode` on page 5-622	Source of signal logging name.
`LoggingName` on page 5-622	Custom signal logging name.
`DecimateData` on page 5-623	Use subset of sample points.
`Decimation` on page 5-623	Decimation value (`n`): Simulink logs every `n`th data point.
`LimitDataPoints` on page 5-623	Limit number of data points to log.
`MaxPoints` on page 5-624	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
`LoggingInfo` on page 5-624	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_mdlref_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.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 marked for logging within a model.

You can control whether a top-level model and referenced models 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 `LoggingMode` to `LogAllAsSpecifiedInModel`.
For top-level model and all referenced models, use override signal logging settings.	Set `LoggingMode` to `OverrideSignals`.
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 `LoggingMode` to `OverrideSignals`. Include models you want to ignore override signal logging settings in the `LogAsSpecifiedByModels` cell array.

For more information and examples, see “Override Signal Logging Settings from MATLAB”.

Property Summary

Name	Description
`LoggingMode` on page 5-628	Signal logging override status
`LogAsSpecifiedByModels` on page 5-629	Source of signal logging settings for the top-level model or a top-level Model block
`Signals` on page 5-630	All signals that have signal override settings

Method Summary

Name	Description
`findSignal` on page 5-634	Find signals within the `Signals` vector, using block path and output port index.
`verifySignalAndModelPaths` on page 5-638	Verify signal and model paths for the model signal logging override object.
`getLogAsSpecifiedInModel` on page 5-636	Determine whether the model logs signals as specified in the model or uses override settings.
`setLogAsSpecifiedInModel` on page 5-637	Set the logging mode for the top-level model or a top-level Model block.
`createFromModel` on page 5-630	Create and populate a model signal logging override object with all logged signals in the model reference hierarchy.
`ModelLoggingInfo` on page 5-633	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'`, the `LogAsSpecifiedByModels` cell array specifies the top-level models and top-level Model blocks that ignore the `'OverrideSignals'` setting and log signals as specified in the models or Model blocks.

- 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_mdlref_bus` model and automatically adds each logged signal in the model to that object:

```
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
'sldemo_mdlref_bus')
```

```
mi = 

  ModelLoggingInfo with properties:

                   Model: 'sldemo_mdlref_bus'
             LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
                 Signals: [1x3 Simulink.SimulationData.SignalLoggingInfo]
```

To apply the model override object settings, use:

```
set_param(sldemo_mdlref_bus,'DataLoggingOverride',mi);
```

You can use the options for the `createFromModel` method to specify how the method should handle model components like variants and model references. For example, use the `Variants` option to create a `model_logging_override` object that includes logged signals in all variants of the `sldemo_variant_subsystems` model.

By default, the `sldemo_variant_subsystems` model does not log any signals. Start by configuring the output signals from the `Linear Controller` and `Nonlinear Controller` subsystems for logging.

```
% Open the sldemo_variant_subsystems model
sldemo_variant_subsystems;
```

```
% Mark the output of the Linear Controller subsystem for logging
ph = get_param('sldemo_variant_subsystems/Controller/Linear Controller',...
    'PortHandles');
set_param(ph.Outport(1),'DataLogging','on');

% Mark the output of the Nonlinear Controller subsystem for logging
ph1 = get_param('sldemo_variant_subsystems/Controller/Nonlinear Controller',...
    'PortHandles');
set_param(ph1.Outport(1),'DataLogging','on');
```

Then, use the `createFromModel` method to create a `model_logging_override` object that includes signals logged in all variant subsystems of the `sldemo_variant_subsystems` model.

```
% Create a model_logging_override object for the model including all variants
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
'sldemo_variant_subsystems', 'Variants', 'AllVariants')
```

```
mi = 

  ModelLoggingInfo with properties:

                     Model: 'sldemo_variant_subsystems'
               LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
                   Signals: [1×2 Simulink.SimulationData.SignalLoggingInfo]
```

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_mdlref_bus');
mi.LoggingMode = 'LogAllAsSpecifiedInModel'
```

```
mi =

  ModelLoggingInfo with properties:

                   Model: 'sldemo_mdlref_bus'
             LoggingMode: 'LogAllAsSpecifiedInModel'
    LogAsSpecifiedByModels: {}
                 Signals: []
```

To apply the model override object settings, use:

```
set_param(sldemo_mdlref_bus, 'DataLoggingOverride', mi);
```

The following example shows how to log only signals in the top-level model:

```
mi = ...
Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
mi.LoggingMode = 'OverrideSignals';
mi = mi.setLogAsSpecifiedInModel('sldemo_mdlref_bus', true);
set_param('sldemo_mdlref_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_mdlref_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_mdlref_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).

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_mdlref_bus;
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
mi.LoggingMode = 'OverrideSignals';
mi = setLogAsSpecifiedInModel(mi, 'sldemo_mdlref_bus', true);
set_param('sldemo_mdlref_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: [1x11 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.Parameter class

Stores logged parameter data and metadata

Description

The `Simulink.SimulationData.Parameter` object stores data and metadata for logged block parameters. Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector during simulation. To access logged parameter data, you can export the simulation run from the Simulation Data Inspector using the UI or the `Simulink.sdi.exportRun` function. For more information about exporting simulation runs with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”.

Construction

`dataset = Simulink.sdi.exportRun(runID)` returns a `Simulink.SimulationData.Parameter` object as an element in `dataset` when the run corresponding to `runID` contains logged parameter data.

Input Arguments

`runID` — Run ID for a run with logged parameter data
integer

Run ID for the run containing logged parameter data. Run IDs are assigned by the Simulation Data Inspector. You can get the run ID for a simulation run using the `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex` function.

Output Arguments

`dataset` — Dataset containing the run data
`Simulink.SimulationData.Dataset`

`Simulink.SimulationData.Dataset` object containing the run data and metadata. When the run contains logged parameter data, the dataset contains a

Simulink.SimulationData.Parameter object as an element for each logged parameter. The Simulink.SimulationData.Parameter element takes the name of the logged parameter. You can access a Simulink.SimulationData.Parameter object using get.

Properties

Name — Parameter name in Dashboard label
character vector

Parameter name as it appears in the label for the Dashboard block.

Example: 'Mu:Gain'

BlockPath — Path of the block associated with the parameter
Simulink.SimulationData.BlockPath

Path to the block the parameter or variable corresponds to, returned as a Simulink.SimulationData.BlockPath object.

Example: vdp/Mu

ParameterName — Parameter name
character vector

Name of the logged parameter as it appears in the block dialog box. For variables, the ParameterName property is empty.

Example: 'Gain'

VariableName — Variable name
character vector

Name of the logged variable. For parameters, the VariableName property is empty.

Example: 'Zw'

Values — Timeseries of parameter values
timeseries

timeseries of parameter values. For logged variables, the timeseries name is the variable name. For logged parameters, the timeseries name is empty.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Access Logged Parameter Data

This example shows how to access logged parameter data. Parameter data automatically logs to the Simulation Data Inspector when you connect a Dashboard block to a block parameter. Parameter data does not export to the workspace with other simulation data at the end of simulation. You can access the logged parameter data by exporting the run containing it from the Simulation Data Inspector.

Log Parameter Data

Run a simulation of the model `ex_vdp_param`, a modified version of the `vdp` model with an Edit block connected to the gain parameter of the Mu block. The parameter data logs with the signal data for signals marked for logging.

```
sim('ex_vdp_param')
```

Export Run

Use the Simulation Data Inspector programmatic interface to get the run ID for the `ex_vdp_param` simulation, and export the run.

```
index = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(index);

dataset = Simulink.sdi.exportRun(runID);
```

Access Parameter Data

Use the `get` method to access the `Simulink.SimulationData.Parameter` object for the logged parameter data. The `Values` property contains the timeseries data for the parameter.

```
muGain = dataset.get('Mu:Gain')
```

```
muGain = 
  Simulink.SimulationData.Parameter
  Package: Simulink.SimulationData

  Properties:
         Name: 'Mu:Gain'
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
       Values: [1x1 timeseries]


  Methods, Superclasses
```

See Also

Simulink.SimulationData.BlockPath | Simulink.SimulationData.Dataset | get

Topics

"Tune and Visualize Your Model with Dashboard Blocks"
"View Data with the Simulation Data Inspector"

Introduced in R2018a

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 5-645	`Simulink.BlockPath` of source block of a signal to log.
`OutputPortIndex` on page 5-646	Index of an output port to log.
`LoggingInfo` on page 5-646	`Simulink.SimulationData.LoggingInfo` object containing all logging override settings for a signal.

Method Summary

Name	Description
`SignalLoggingInfo` on page 5-647	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_mdlref_bus model.

```
sldemo_mdlref_bus;
mi = Simulink.SimulationData.ModelLoggingInfo(...
'sldemo_mdlref_bus');
mi.LoggingMode = 'OverrideSignals';
mi.Signals = ...
   Simulink.SimulationData.SignalLoggingInfo(...
   {'sldemo_mdlref_bus/CounterA', ...
'sldemo_mdlref_counter_bus/Bus Creator'}, 1)
```

The output is:

```
mi =

  Data.ModelLoggingInfo with properties:

                     Model: 'sldemo_mdlref_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 5-649	Block path for the source block for the signal
`Name` on page 5-650	Name of signal element to use for name-based access
`PropagatedName` on page 5-650	Propagated signal name, if any
`PortIndex` on page 5-650	Numeric index of port that was logged
`PortType` on page 5-651	Type of port that was logged: for signal logging, the port type is `'outport'`
`Values` on page 5-651	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 Outport 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 Outport 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 `'outport'`

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_mdlref_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)
- MATLAB `timetable` object
- Structure of MATLAB `timetable` objects (for bus signals)
- Array of structures of MATLAB `timetable` objects (for array of buses signals)
- Cell array of MATLAB `timetable` 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”
“Loading MATLAB Timeseries Data to 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 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.

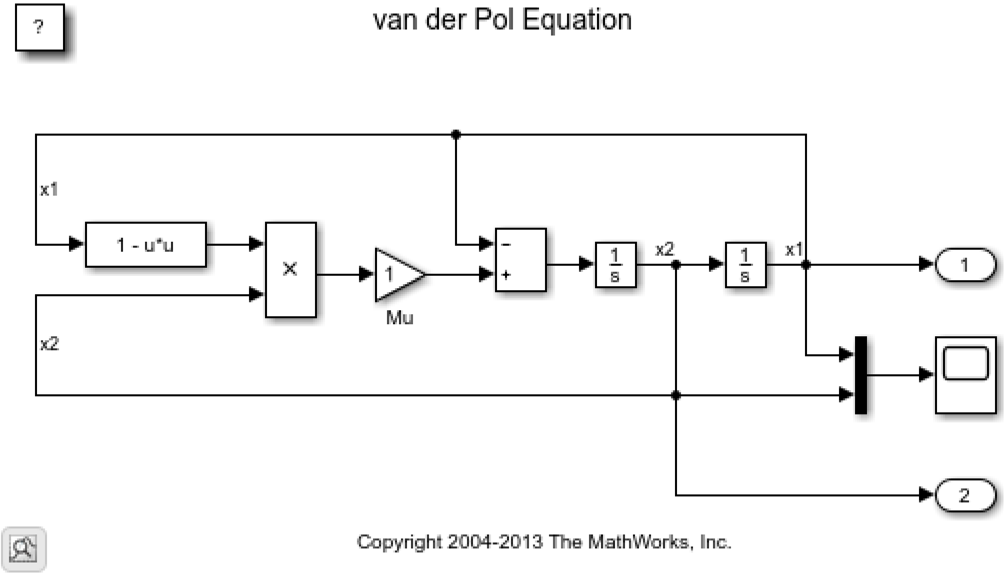
Examples

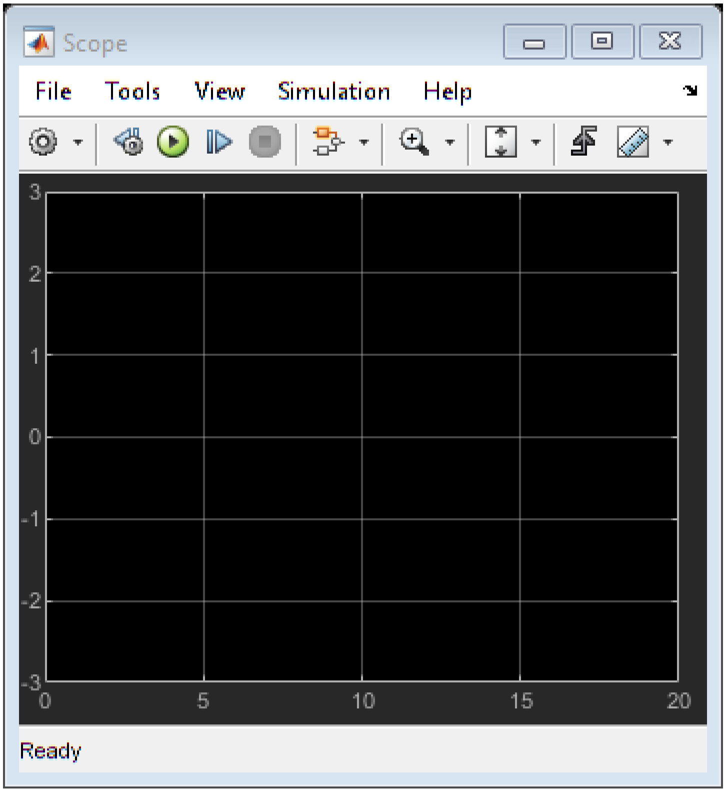
Final State Information in Structure with Dataset Format

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`.

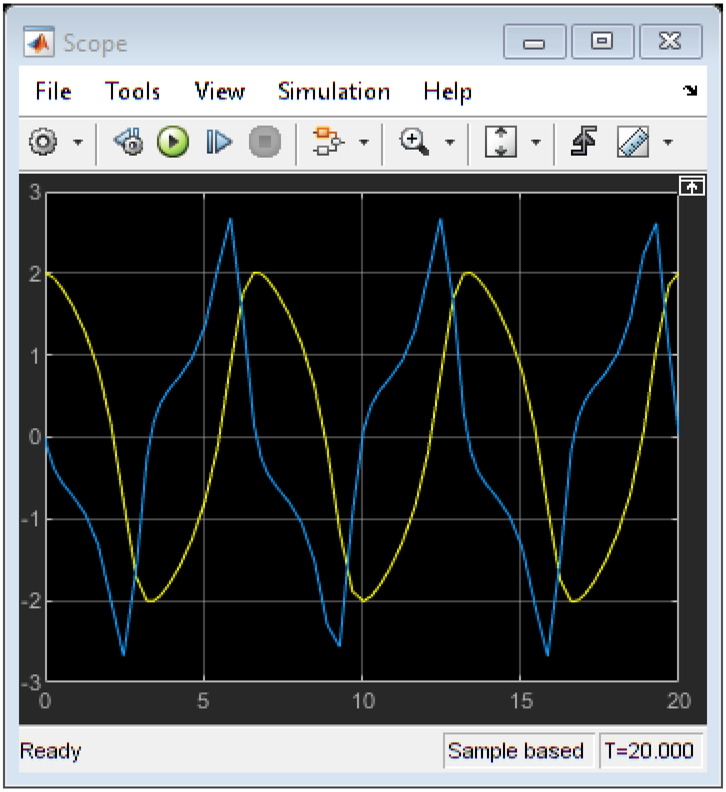
```
open_system('vdp');
set_param(gcs,'SaveFinalState','on','SaveFormat','Dataset');
```





Simulate the `vdp` 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: [1×1 Simulink.SimulationData.BlockPath]
        Label: CSTATE
       Values: [1×1 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 Outport 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

`unitObj` — 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
`Simulink.SimulationData.Unit.setName`	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'
```

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
setModelParameter	Set model parameters to be used for a specific simulation through SimulationInput object.
setBlockParameter	Set block parameters to be used for a specific simulation through SimulationInput object.
setInitialState	Set initial state to be used for a specific simulation through SimulationInput object.
setExternalInput	Set external inputs for a simulation through SimulationInput object.
setVariable	Set variables for a simulation through SimulationInput object.
setPreSimFcn	Specify a MATLAB function to run before start of each simulation through SimulationInput object.
setPostSimFcn	Specify a MATLAB function to run after each simulation is complete through SimulationInput object.
applyToModel	Apply changes to the model specified through a SimulationInput object.
validate	Validate the contents of the SimulationInput 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

Simulation Manager | Simulink.SimulationInput | applyToModel | parsim | setBlockParameter | setExternalInput | setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn | setVariable | validate

Topics

"Run Multiple Simulations"
"Run Parallel Simulations Using parsim"

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

`Simulation Manager` | `Simulink.SimulationInput` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

Topics

"Run Multiple Simulations"
"Run Parallel Simulations Using parsim"

Introduced in R2017a

setExternalInput

Set external inputs for a simulation through `SimulationInput` object, `in`

Syntax

```
in = in.setExternalInput([t, u1,..uN])
in = in.setExternalInput(ds)
in = in.setExternalInput(ts)
```

Description

`in = in.setExternalInput([t, u1,..uN])` allows you to directly specify numerical arrays as inputs to a model if a model has root inports.

`in = in.setExternalInput(ds)` allows you to directly specify dataset objects as external inputs to a model if a model has root inports..

`in = in.setExternalInput(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_mdlref_counter');
```

Create a `SimulationInput` object for this model.

```
in = Simulink.SimulationInput('sldemo_mdlref_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 = in.setExternalInput([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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

Topics

“Run Multiple Simulations”

“Run Parallel Simulations Using parsim”

Introduced in R2017a

setPostSimFcn

Specify a MATLAB function to run after each simulation is complete through SimulationInput object, in

Syntax

```
in = in.setPostSimFcn(@(y) myfunction(arg1, arg2...))
```

Description

in = in.setPostSimFcn(@(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 = in.setPostSimFcn(@(x) postsim(x));
in = in.setModelParameter('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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPreSimFcn` | `setVariable` | `validate`

Topics

"Run Multiple Simulations"
"Run Parallel Simulations Using parsim"

Introduced in R2017a

setPreSimFcn

Specify a MATLAB function to run before start of each simulation through `SimulationInput` object, in

Syntax

```
in = in.setPreSimFcn(@(x) myfunction(arg1, arg2...))
```

Description

`in = in.setPreSimFcn(@(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)
end
```

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) = in(idx).setPreSimFcn(@(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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setVariable` | `validate`

Topics

"Run Multiple Simulations"
"Run Parallel Simulations Using parsim"

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.

You can use getBlockParameter('BlockPath','ParameterName') method to get the value of block parameter and the removeBlockParameter('BlockPath','ParameterName') method to remove block parameter from the Simulink.SimulationInput object.

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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

Topics

“Run Multiple Simulations”

“Run Parallel Simulations Using parsim”

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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

Topics

"Run Multiple Simulations"
"Run Parallel Simulations Using parsim"

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.

You can use `getModelParameter('ParameterName')` method to get the value of model parameter and the `removeModelParameter('ParameterName')` method to remove model parameter from the `Simulink.SimulationInput` object

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);
```

Specify a timeout of 5 seconds and modify model parameters, `StartTime` and `StopTime`

```
in = in.setModelParameter('Timeout',5);
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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate`

Topics

“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

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.

You can use getVariable('VariableName') method to get the value of variable and the removeVariable('VariableName') method to remove variable from the Simulink.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 `sldemo_househeat` 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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `validate`

Topics

"Run Multiple Simulations"
"Run Parallel Simulations Using parsim"

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

`Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable`

Topics

"Run Multiple Simulations"
"Run Parallel Simulations Using parsim"

Introduced in R2017a

Simulink.Simulation.Future class

Package: Simulink

Create `Future` object for simulation

Description

Create a `Simulink.Simulation.Future` when you execute `parsim` with the `'RunInBackground'` argument set to `'on'`. The `parsim` command runs multiple simulations in parallel using the inputs specified with an array of `Simulink.SimulationInput` objects. You can use this object to monitor the status of ongoing simulations, fetch outputs of completed simulations, or cancel ongoing 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 or if Parallel Computing Toolbox is not used.

Construction

`future = parsim(in,'RunInBackground','on')` creates a `Simulink.Simulation.Future` object, `future`, while running multiple simulations in parallel the using the inputs specified in the `Simulink.SimulationInput` object, `in`.

Input Arguments

in — `Simulink.SimulationInput` object array
object (default) | array

A `Simulink.SimulationInput` object or an array of `Simulink.SimulationInput` objects is used to run multiple simulations. Specify parameters and values of a model to run multiple simulations without making it dirty.

Example: `in = Simulink.SimulationInput('vdp')`, `in(1:10) = Simulink.SimulationInput('vdp')`

'RunInBackground' — parsim argument to enable RunInbackground

'off' (default) | 'on'

Set to 'on', to run simulations asynchronously, keeping the MATLAB command prompt available.

Properties

Diary — Log of outputs from the simulation

text file

Text log of outputs from the simulation.

This property is read-only.

ID — Numeric identifier of the future object

scalar integer

ID of the future object, specified as a scalar integer.

This property is read-only.

Read — Whether the outputs have been read

1 | 0

Whether a call to fetchNext or fetchOutputs has read the outputs in the Simulink.Simulation.Future object array, specified as 1 if true and 0 if false.

This property is read-only.

State — Current state of future object array

'pending' | 'queued' | 'running' | 'finished' | 'failed' | 'unavailable'

Current state of future object array, specified as 'pending', 'queued', 'running', 'finished','failed', or 'unavailable'.

This property is read-only.

Methods

Method	Purpose
cancel	Cancel a pending, queued, or running Simulink.Simulation.Future object
fetchNext	Fetch next available unread output from Simulink.Simulation.Future object array
fetchOutputs	Retrieve Simulink.SimulationOutput from Simulink.Simulation.Future
wait	Wait for Simulink.Simulation.Future objects to complete simulation

Examples

Create a Simulink.Simulation.Future Object

This example shows how to create a Simulink.Simulation.Future object array and use it to retrieve outputs and see the status of simulations.

This example runs several simulations of the vdp model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

```
open_system('vdp');
Mu_Values = [0.5:0.25:5];
MuVal_length = length(Mu_Values);
```

Using Mu_Values, initialize an array of Simulink.SimulationInput objects. To preallocate the array, a loop index is made to start from the largest value.

```
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu',...
        'Gain',num2str(Mu_Values(i)));
end
```

Simulate the model using parsim. Set it to 'RunInBackground', to be able to use the command prompt, while simulations are running.

```
Future = parsim(in,'RunInBackground','on');
```

Use the `fetchNext` method on `Future` simulations.

```
for i = 1:MuVal_length
    [completedIdx,simOut] = fetchNext(Future)
end
```

See Also

Functions
`batchsim` | `cancel` | `fetchNext` | `fetchOutputs` | `parsim` | `wait`

Classes
`Simulink.Simulation.Job` | `Simulink.SimulationInput`

Topics

"Multiple Simulation Workflows"

Introduced in R2018a

cancel

Cancel a pending, queued, or running `Simulink.Simulation.Future` object

Syntax

```
cancel(Future)
```

Description

`cancel(Future)` stops the objects of the `Simulink.Simulation.Futures` array, `Future`, that are currently in `'pending'`, `'queued'`, or `'running'` state. For elements of the `Futures` in the `'finished'` state, no action is taken.

Examples

Cancel Simulations of Future Objects

This example shows how to use the `cancel` method on an array of `future` objects to stop the simulations.

This example runs several simulations of the `vdp` model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

```
open_system('vdp');
Mu_Values = [0.5:0.25:1000];
MuVal_length = length(Mu_Values)
```

Using `Mu_Values`, initialize an array of `Simulink.SimulationInput` objects. To preallocate the array, a loop index is made to start from the largest value.

```
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu',...
        'Gain',num2str(Mu_Values(i)));
end
```

Simulate the model using `parsim`. Set to '`RunInBackground`' to enable the use the command prompt, while simulations are running.

```
Future = parsim(in,'RunInBackground','on');
```

Now, assume that you want to run simulations with different values of Mu and cancel the ongoing simulations.

```
cancel(Future)
```

Input Arguments

`Future` — `Simulation.Simulink.Future` object
array

Array of `Simulation.Simulink.Future` objects. To create `Future`, run `parsim` with `'RunInBackground'` option set to `'on'`.

Example: `Future = parsim(in,'RunInBackground','on')`

See Also

Functions
`batch` | `batchsim` | `fetchNext` | `fetchOutputs` | `parsim` | `wait`

Classes
`Simulink.Simulation.Future` | `Simulink.SimulationInput`

Introduced in R2018a

fetchNext

Fetch next available unread output from Simulink.Simulation.Future object array

Syntax

```
[idx,simOut] = fetchNext(Future)
[idx,simOut] = fetchNext(Future, Timeout)
```

Description

[idx,simOut] = fetchNext(Future) waits for the unread element of Simulink.Simulation.Future array, Future, to reach a 'finished' state. It returns the index of the simulation that finished, and the corresponding Simulink.SimulationOutput object.

[idx,simOut] = fetchNext(Future, Timeout) waits for a maximum of Timeout seconds for a result to become available. If the timeout expires before any result is available, simOut is returned as an empty array.

An error is reported if there are no elements in Future with property Read as false. You can check for are any unread futures using anyUnread = ~all([F.Read]).

fetchNext displays an error if any element of Future with a 'finished' state encounters an error during execution. The Read property of that element becomes true allowing any subsequent call to fetchNext to proceed.

Examples

Create a Simulink.Simulation.Future Object

This example shows how to create a Simulink.Simulation.Future object array and use it to retrieve outputs and see the status of simulations.

This example runs several simulations of the vdp model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

```
open_system('vdp');
Mu_Values = [0.5:0.25:5];
MuVal_length = length(Mu_Values);
```

Using `Mu_Values`, initialize an array of `Simulink.SimulationInput` objects. To preallocate the array, a loop index is made to start from the largest value.

```
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu',...
        'Gain',num2str(Mu_Values(i)));
end
```

Simulate the model using `parsim`. Set it to '`RunInBackground`' to be able to use the command prompt, while simulations are running.

```
Future = parsim(in,'RunInBackground','on');
```

Use the `fetchNext` method on `Future`.

```
for i = 1:MuVal_length
    [completedIdx,simOut] = fetchNext(Future)
end
```

Input Arguments

Future — Simulation.Simulink.Future object

array

Array of `Simulation.Simulink.Future` objects. To create `Future`, run `parsim` with `'RunInBackground'` option set to `'on'`.

Example: `Future = parsim(in,'RunInBackground','on')`

Timeout — Number of seconds specified for fetchNext to time out

scalar

Specify a `Timeout` for `fetchNext` to retrieve the results from the `Simulation.Simulink.Future` array, `Future`.

Example: `[idx, simOut] = fetchNext(Future, 45)`

Output Arguments

`idx` — Index of the simulation

integer

When `fetchNext` method is used on an array of `Simulink.Simulation.Future` objects, it returns the index of the simulation whose output is being retrieved.

`simOut` — Simulation object containing logged simulation results

object | array

Array of `Simulink.SimulationOutput` objects that contain 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 are 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** tool can directly log signals.

See Also

Functions

`batch` | `batchsim` | `cancel` | `fetchOutputs` | `parsim` | `wait`

Classes

`Simulink.Simulation.Future` | `Simulink.SimulationInput`

fetchOutputs

Retrieve Simulink.SimulationOutput from Simulink.Simulation.Future objects

Syntax

```
simOut = fetchOutputs(Future)
```

Description

simOut = fetchOutputs(Future) fetches the output from an array of Simulink.Simulation.Future objects, Future, after each element of Future is in a 'finished' state. fetchOutputs returns an array of Simulink.SimulationOutput objects.

Examples

Create a Future and Retrieve Outputs Using fetchOutputs

This example shows how to use the fetchOutputs method on an array of future objects to retrieve a Simulink.SimulationOutput array.

This example runs several simulations of the vdp model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

```
open_system('vdp');
Mu_Values = [0.5,0.75,1,1.25];
MuVal_length = length(Mu_Values);
```

Using Mu_Values, initialize an array of Simulink.SimulationInput objects. To preallocate the array, a loop index is made to start from the largest value.

```
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu',...
        'Gain',num2str(Mu_Values(i)));
end
```

Simulate the model using `parsim`. Set to '`RunInBackground`' to enable the use of command prompt, while simulations are running.

```
Future = parsim(in,'RunInBackground','on');
```

Use the `fetchOutputs` method on `Future`

```
simOut = fetchOutputs(Future)
```

```
simOut =

1x4 Simulink.SimulationOutput array
```

Input Arguments

`Future` — `Simulation.Simulink.Future` object

array

Array of `Simulation.Simulink.Future` objects. To create, `Future`, run `parsim` with `'RunInBackground'` option set to `'on'`.

Example: `Future = parsim(in,'RunInBackground','on')`

Output Arguments

`simOut` — Simulation object containing logged simulation results

object

Array of `Simulink.SimulationOutput` objects that contain 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 are 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** tool can directly log signals.

See Also

Functions
`batch` | `batchsim` | `cancel` | `fetchNext` | `parsim` | `wait`

Classes
`Simulink.Simulation.Future` | `Simulink.SimulationInput`

Introduced in R2018a

wait

Wait for `Simulink.Simulation.Future` objects to complete simulation

Syntax

```
Ok = wait(Future)
```

Description

`Ok = wait(Future)` blocks the command prompt until each element of the `Simulink.Simulation.Future` array, `Future` is in a `'finished'` state.

Examples

Wait for the `Future` Array to Complete Simulations

This example shows how to use the `wait` method on an array of `future` objects.

This example runs several simulations of the `vdp` model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

```
open_system('vdp');
Mu_Values = [0.5:0.25:5];
MuVal_length = length(Mu_Values);
```

Using `Mu_Values`, initialize an array of `Simulink.SimulationInput` objects. To preallocate the array, a loop index is made to start from the largest value.

```
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu',...
        'Gain',num2str(Mu_Values(i)));
end
```

Simulate the model using `parsim`. Set to '`RunInBackground`' to enable the use of command prompt while the simulations are running and to create an array of `Simulink.Simulation.Future` objects.

```
Future = parsim(in,'RunInBackground','on');
```

Use the `wait` method on `Future` to block the execution.

```
Ok = wait(Future)
```

Input Arguments

`Future` — `Simulation.Simulink.Future` object

array

Array of `Simulation.Simulink.Future` objects. To create `Future`, run `parsim` with `'RunInBackground'` option set to `'on'`.

Example: `Future = parsim(in,'RunInBackground','on')`

Output Arguments

`Ok` — Whether the wait is completed successfully

1 | 0

`Ok` is `true` if the wait completes successfully, `false` if any of the `Future` objects failed execution or were canceled. Specified as 1 if true, 0 if false.

See Also

Functions
`batch` | `batchsim` | `cancel` | `fetchNext` | `fetchOutputs` | `parsim`

Classes
`Simulink.Simulation.Future` | `Simulink.SimulationInput`

Introduced in R2017b

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: • 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: • 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	Reason the simulation stopped, represented by one of the following: - `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 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 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. - `StepBackward` – Simulation paused after clicking step backward.

Field Name	Type	Description
		• 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	Superset of information stored in StopEvent and StopEventSource.
ErrorDiagnostic	struct	Error reported during simulation, represented by the following fields: • 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 []. By passing the name–value pair 'CaptureErrors', 'on' to the sim command, errors generated during simulation are reported in ExecutionInfo.ErrorDiagnostic. The sim command does not capture generated errors.

Field Name	Type	Description
WarningDiagnostics	Array of struct	Array of all warnings reported during the simulation. Each array item is represented by the following fields: • Diagnostic – MSLDiagnostic object that includes object paths, ID, message, cause, and stack. • SimulationPhase – Represented as: 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

Field Name	Type	Description
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).

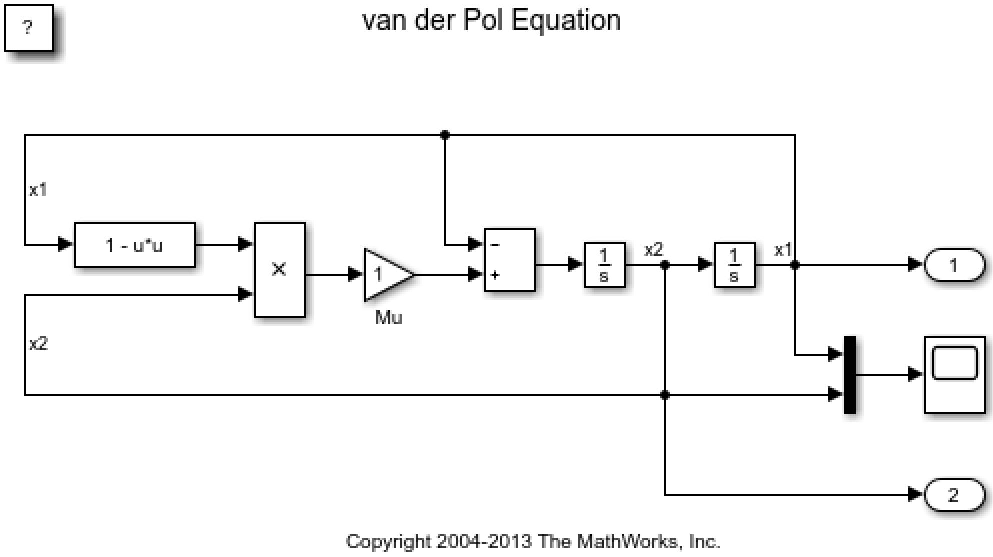
Examples

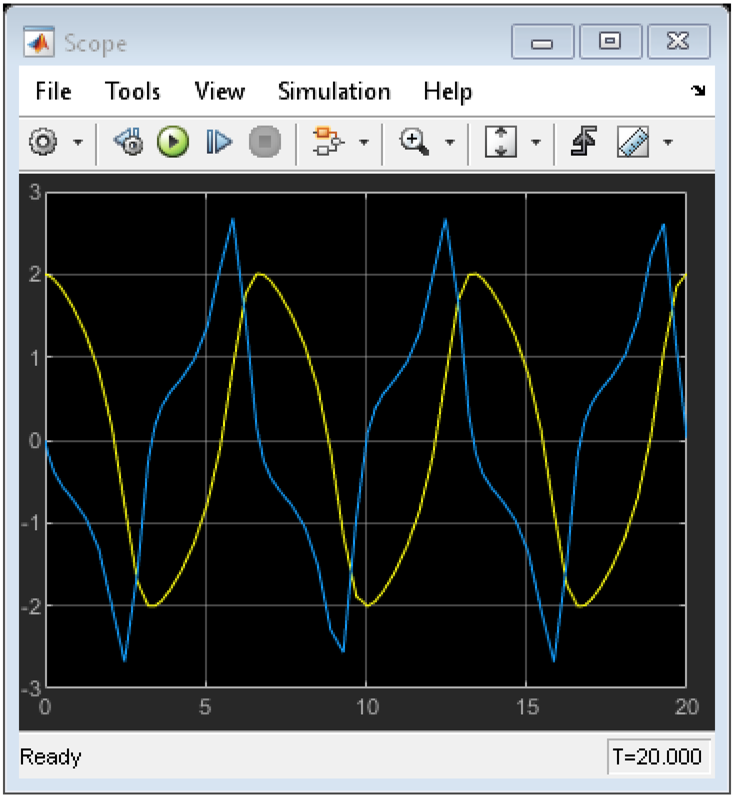
Get a SimulationMetadata Object for vdp Simulation

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');
```





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

find	Access and display values of simulation results
get	Access and display values of simulation results
getSimulationMetadata	Return `SimulationMetadata` object for simulation
setUserData	Store custom data in `SimulationMetadata` object that `SimulationOutput` object contains
setUserString	Store custom character vector in `SimulationMetadata` object that `SimulationOutput` object contains
who	Access and display output variable names of simulation

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: [1×1 struct]
       TimingInfo: [1×1 struct]
    ExecutionInfo: [1×1 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.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 = logsout.Gain.a.Data;
```

or

```
>> logsout.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

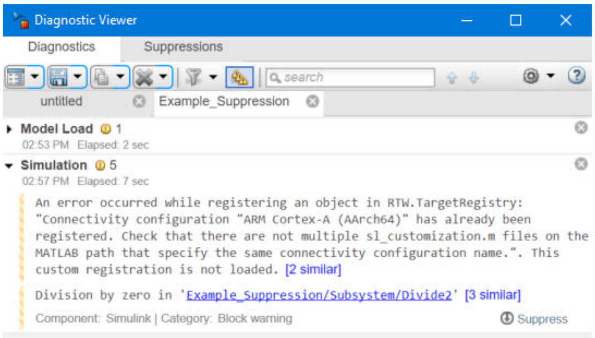
Suppress diagnostic messages from a specified block

Description

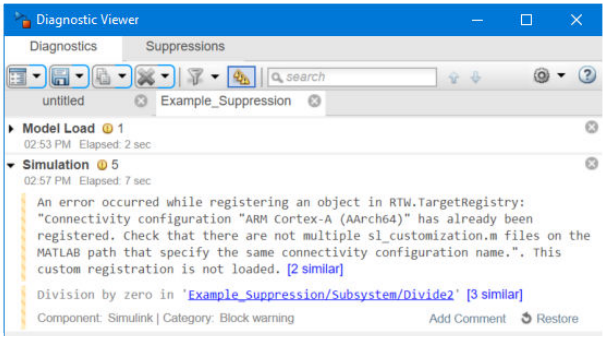
A `Simulink.SuppressedDiagnostic` object contains information related to diagnostic warnings or errors that are suppressed from being thrown during simulation.

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 add a comment for the suppressed diagnostic. You can restore the diagnostic by clicking **Restore**.



```
An error occurred while registering an object in RTW.TargetRegistry:
"Connectivity configuration "ARM Cortex-A (AArch64)" has already been
registered. Check that there are not multiple sl_customization.m files on the
MATLAB path that specify the same connectivity configuration name.". This
custom registration is not loaded. [2 similar]

Division by zero in 'Example_Suppression/Subsystem/Divide2' [3 similar]
```



```
An error occurred while registering an object in RTW.TargetRegistry:
"Connectivity configuration "ARM Cortex-A (AArch64)" has already been
registered. Check that there are not multiple sl_customization.m files on the
MATLAB path that specify the same connectivity configuration name.". This
custom registration is not loaded. [2 similar]

Division by zero in 'Example_Suppression/Subsystem/Divide2' [3 similar]
```

`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` | `string`

`message_id` — message identifier of diagnostic

message identifier

The message identifier of the diagnostic, specified as a character vector or string. 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` | `string`

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 and time the suppression object was last modified

character vector

Date and time 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 added or edited 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
suppress	Suppress diagnostic specified by `Simulink.SuppressedDiagnostic` object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a Simulink.SuppressedDiagnostic Object

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 thrown by the Constant block, `one`.

```
suppression = Simulink.SuppressedDiagnostic('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 = 

  SuppressedDiagnostic 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);
```

See Also

`Simulink.SuppressedDiagnostic` | `Simulink.SuppressedDiagnostic.restore` | `Simulink.getSuppressedDiagnostics` | `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
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
```

See Also

`Simulink.SuppressedDiagnostic` | `Simulink.getSuppressedDiagnostics` | `Simulink.restoreDiagnostic` | `Simulink.suppressDiagnostic`

Topics

"Suppress Diagnostic Messages Programmatically"

Introduced in R2016b

suppress

Class: Simulink.SuppressedDiagnostic
Package: Simulink

Suppress diagnostic specified by `Simulink.SuppressedDiagnostic` object

Syntax

```
suppress(SuppressedDiagnostic)
```

Description

`suppress(SuppressedDiagnostic)` suppresses the specified suppressed diagnostic object.

Input Arguments

`SuppressedDiagnostic` — Suppressed diagnostic object to suppress
`Simulink.SuppressedDiagnostic` object

`Simulink.SuppressedDiagnostic` object

See Also

`Simulink.SuppressedDiagnostic` | `Simulink.getSuppressedDiagnostics` | `Simulink.restoreDiagnostic` | `Simulink.suppressDiagnostic`

Topics

"Suppress Diagnostic Messages Programmatically"

Introduced in R2018a

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 `Simulink.Timeseries` 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 Simulink.Timeseries 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 Simulink.Timeseries 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 Simulink.Timeseries object. If the signal is aperiodic (continuous signal with variable-step solver), this property has a value of NaN. 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 Simulink.Timeseries object, i.e., the length of the arrays referenced by the object's Time and Data 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 Simulink.TimeInfo 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.

```
logsout.b2

Simulink.TsArray (untitled/Bus Creator1):
  Name                   elements   Simulink Class

  x1                        1       Timeseries
  b1                        2       TsArray
```

The `Simulink.ModelDataLogs` object, named `logsout`, 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, (logsout.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 = logsout.b2.x1.Data;
```

or

```
logsout.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 .

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

addConfiguration	Add a new variant configuration to the variant configuration data object
addConstraint	Add a constraint to the variant configuration data object
addControlVariables	Add control variables to an existing variant configuration
addCopyOfConfiguration	Add a copy of an existing variant configuration to the variant configuration data object
addSubModelConfigurations	Add to a variant configuration the names of the configurations to be used for submodels
existsFor	Check if variant configuration data object exists for a model
getConfiguration	Returns the variant configuration with a given name from a variant configuration data object
getDefaultConfiguration	Returns default variant configuration, if any, for a variant configuration data object
getFor	Get existing variant configuration data object for a model
getOrCreateFor	Get existing or create a new variant configuration data object for a model
removeConfiguration	Remove a variant configuration with a given name from the variant configuration data object
removeConstraint	Remove a constraint from the variant configuration data object
removeControlVariable	Remove a control variable from a variant configuration
removeSubModelConfiguration	Remove from a variant configuration, the configuration to be used for a sub model.
setDefaultConfigurationName	Set name of the default variant configuration for a variant configuration data object
validateModel	Validate all variant blocks in the model and submodels in the hierarchy during simulation
VariantConfigurationData	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 = vcdataObj.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 = vcdataObj.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 the 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

variantLegend	Display or control behavior of variant condition legend
reduceModel	Generate reduced model for specified variant configurations
convertToVariant	Convert Subsystem, or Model block, or Variant Model block to a 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 and library blocks are also 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 caseinsensitive whereas, the value string is casesensitive. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

NamedConfigurations — Variant configuration name

{''} | character vector | 'cell array'

Specifies the names of variant configurations. By default, current values of variant control variables are used for reduction.

VariableGroups — Variant control variable value

{''} | 'cell array'

Specifies the variant control variable values to be used for reduction. By default, the current values of variant control variables are used.

The specified values must be a cell array with variant control variable names and their corresponding values.

Note 'VariableGroups' and 'NamedConfigurations' are mutually exclusive.

Consider this example:

```
Simulink.VariantManager.reduceModel('iv_model', ...
                                              'VariableGroups',...
                                              {{'V',1,'W',1},{'V',2,'W',2}})
```

FullRangeVariables — Variant control variable value

{''} | 'cell array'

Specifies the full-range variant control variable values to be used for reduction. This allows you to reduce a model for all valid values of the specified variant control variable. Provide a reference value for variant control variable that results in a successful model compilation.

Consider this example:

```
Simulink.VariantManager.reduceModel('slexVariantReducer',...
                      'VariableGroups',{'V',1},...
                      'FullRangeVariables',{'W',1});
```

You can specify a variant control variable, 'W', as a full-range variant control variable. This allows you to reduce the model for all valid values of variable 'W'. In the example, full-range variant control variable W uses a reference value of 1.

`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.

`GenerateSummary` — Generates a summary html file

true | {false}

When the value is `true`, the Variant Reducer generates a html file with details about the reduced model and any modifications that may be required for masks and callbacks.

Note To generate summary, you must have **Simulink Report Generator** license.

Examples

```
% Reduce model based on its variant control variable values in the base workspace.
Simulink.VariantManager.reduceModel('sldemo_variant_subsystems');

% Reduce the model associated with a variant configuration data object and configurations to be retained in
Simulink.VariantManager.reduceModel('slexVariantManagementExample', ...
                                          'NamedConfigurations', {'LinInterStd',
                                          'NonLinExterHighFid'})

% Reduce the model by specifying variant control variable values. Here, two groups are specified correspondi
% {V==1, W==1}, and {V==2, W==2} respectively.
Simulink.VariantManager.reduceModel('iv_model', ...
                                          'VariableGroups',...
                                          {{'V',1,'W',1},{'V',2,'W',2}});

% Reduce the model by specifying variant control variable values where 'W' is a full-range variant control v
% automatically maps the specification to correspond to the following four explicit groups: {V==1, W==1}, {V
Simulink.VariantManager.reduceModel('slexVariantReducer',...
                                          'VariableGroups',...
                                          {'V',1},...
                                          'FullRangeVariables',{'W',1});
```

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, or Variant Model block to a Variant Subsystem block

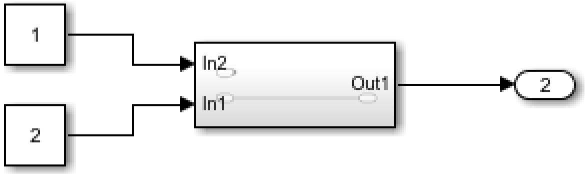
Syntax

```
variant_subsystem = Simulink.VariantManager.convertToVariant(block)
variant_subsystem = Simulink.VariantManager.convertToVariant(
blockHandle)
```

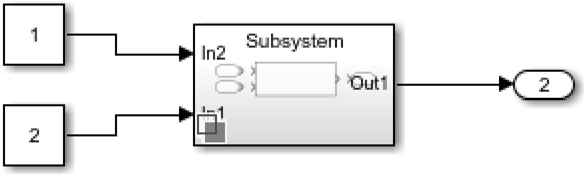
Description

`variant_subsystem = Simulink.VariantManager.convertToVariant(block)` or `variant_subsystem = Simulink.VariantManager.convertToVariant(blockHandle)` converts a Subsystem, or Model block, or Variant Model block to a Variant Subsystem block. A Variant Subsystem can contain Subsystems, Model blocks, or both as choices.

Consider this model with Subsystem block.



You can convert this Subsystem block to a Variant Subsystem block using the `convertToVariant` method.

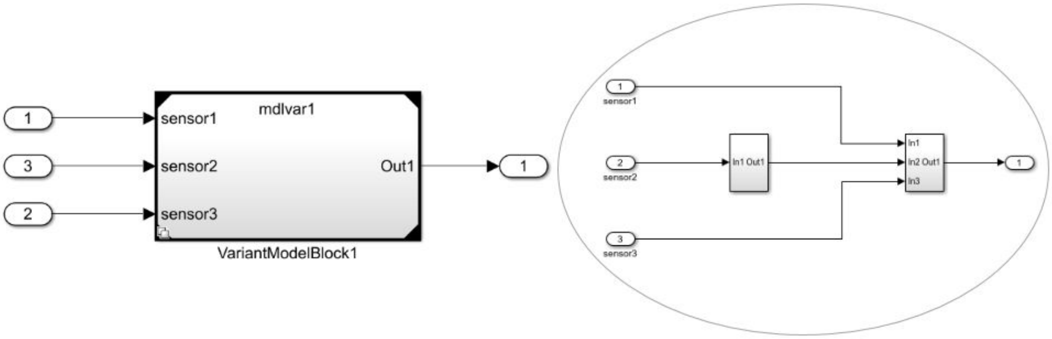


Similarly you can convert a Variant Model block to a Variant Subsystem block.

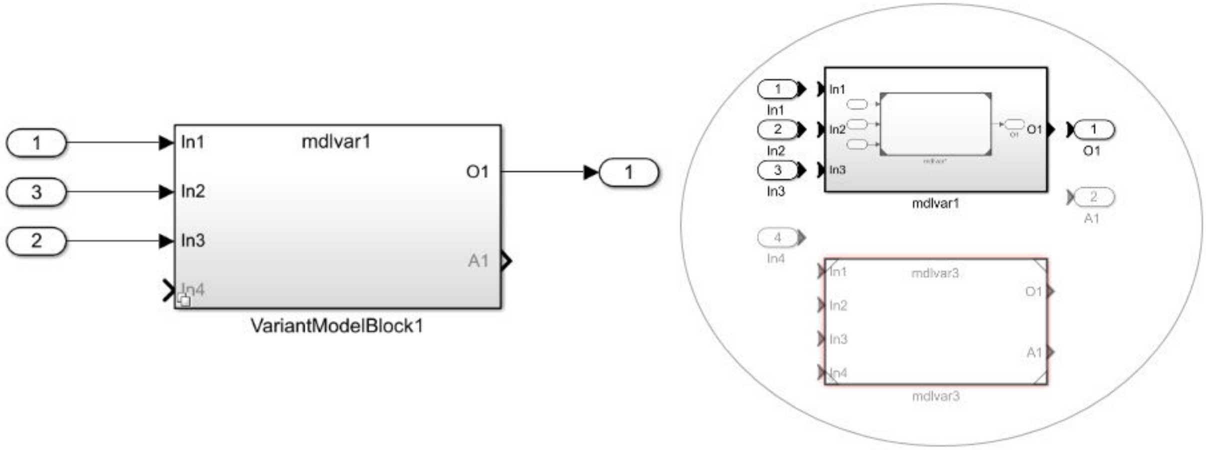
To convert a Variant Model block to a Variant Subsystem block, you can also use the Upgrade Advisor check, **Identify Variant Model blocks and convert those to Variant Subsystem containing Model block choices**. This check provides a **Fix** button to convert Variant Model blocks to Variant Subsystem blocks. For more information on using Upgrade Advisor check to convert a Variant Model block to a Variant Subsystem block, see “Upgrade Advisor Checks”

If there are inconsistencies in the port name or port number in models referenced by Variant Model block, Simulink corrects these inconsistencies while converting the Variant Model block to Variant Subsystem block.

Consider this model with Variant Model block having different port names.



When you convert this Variant Model block to a Variant Subsystem block, the inconsistencies are corrected automatically.



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 choices
- 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 the following condition:

- The Subsystem block is in a Simscape model that has Editing Mode set to **Restricted**.
- The Variant Model block has a mix of control ports or there is a name mismatch in control port types.
- The Variant Model block has control ports with different port numbers and name mapping.

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

`variant_subsystem` — 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('sldemo_mdlref_conversion');
   Simulink.VariantManager.convertToVariant('sldemo_mdlref_conversion/Bus Counter');
```

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

Store information about workspace variables and blocks that use them

Note `Simulink.WorkspaceVar` is not recommended. Use `Simulink.VariableUsage` instead.

Description

A `Simulink.WorkspaceVar` object describes attributes of a workspace variable and lists the blocks that use the variable.

Creation

The `Simulink.findVars` function returns one or more `Simulink.WorkspaceVar` objects that embody the results of searching for variables.

Only the `Simulink.WorkspaceVar` function can set any field value in a `Simulink.WorkspaceVar` object. The fields are otherwise read-only.

Syntax

```
varObj = Simulink.WorkspaceVar(varNames,wkspName)
```

Description

`varObj = Simulink.WorkspaceVar(varNames,wkspName)` creates an array of `Simulink.WorkspaceVar` 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 `Workspace` property of all the objects to the workspace specified by `wkspName`. 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.WorkspaceVar` object for each variable name. You can specify `varNames` with variables that are not used in any loaded models.

Example: `'k'`

Example: `{'k','asdf','fuelFlow'}`

Data Types: `char` | `cell`

`wkspName` — Name of containing workspace

character vector

Name of the workspace that defines the target variables, specified as a character vector. For example, you can specify the MATLAB base workspace. The constructor also determines and sets the `WorkspaceType` property of each of the returned `Simulink.WorkspaceVar` objects.

Example: `'base workspace'`

Example: `'myModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

Properties

`Name` — Name of variable

`''` (empty character vector) (default) | character vector

This property is read-only.

Name of the variable described by the object, returned as a character vector.

`Workspace` — Name of workspace that contains variable

`''` (empty character vector (default) | character vector

This property is read-only.

Name of the workspace that contains the variable, returned as a character vector. For example:

Workspace value	Meaning
`'base workspace'`	The MATLAB base workspace
`'MyModel'`	The model workspace for the model `MyModel`.
`'MyModel/Mask1'`	The mask workspace for the masked block `Mask1` in the model `MyModel`.

`WorkspaceType` — Type of workspace containing variable

`'unknown'` (default) | `'base'` | `'model'` | `'mask'`

This property is read-only.

Type of workspace that contains the variable, returned as a character vector. The possible values are:

- `'base'` — The base workspace
- `'model'` — A model workspace
- `'mask'` — A mask workspace

`UsedByBlocks` — Users of variable

`{}` (empty cell array) (default) | cell array of character vectors

This property is read-only.

Users of the variable, returned as a cell array of character vectors. Each character vector identifies a block that uses the variable. The `Simulink.findVars` function populates this property.

Object Functions

`intersect`	Return intersection of two arrays of Simulink.VariableUsage objects
`setdiff`	Return difference between two arrays of Simulink.VariableUsage objects

Examples

Create Object That Represents Variable in Base Workspace

Return a `Simulink.WorkspaceVar` object for a variable `k` in the base workspace.

```
var = Simulink.WorkspaceVar('k', 'base workspace');
```

Represent All Variables in the Base Workspace

Return an array of `Simulink.WorkspaceVar` objects containing one object for each variable returned by the `whos` command in the base workspace.

```
vars = Simulink.WorkspaceVar(who,WkspName)
```

Represent All Variables in a Model Workspace

Return an array of `Simulink.WorkspaceVar` objects that describes all the variables in a model workspace.

```
hws = get_param('mymodel', 'ModelWorkspace');
vars=Simulink.WorkspaceVar(hws.whos, 'MyModel')
```

Represent All Variables in a Mask Workspace

Return an array 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` | `intersect` | `setdiff`

Introduced in R2010a

Simulink.VariableUsage

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.

Creation

The `Simulink.findVars` function returns `Simulink.VariableUsage` objects.

To create variable usage objects for use as a filter when using `Simulink.findVars`, use the `Simulink.VariableUsage` function described below.

Syntax

```
variableUsageObj = Simulink.VariableUsage(varNames,sourceName)
```

Description

`variableUsageObj = 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

'' (empty character vector) (default) | character vector

This property is read-only.

The name of the variable or enumerated data type the object describes, returned as a character vector.

Source — Name of defining workspace

'' (empty character vector) (default) | character vector

This property is read-only.

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

'unknown source' (default) | 'base workspace' | 'model workspace' | 'mask workspace' | 'data dictionary'

This property is read-only.

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 — Blocks that use the variable or models that use the enumerated type

{} (empty cell array) (default) | cell array of character vectors

This property is read-only.

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 property.

Object Functions

intersect	Return intersection of two arrays of Simulink.VariableUsage objects
setdiff	Return difference between two arrays of Simulink.VariableUsage objects

Examples

Create Object That Represents Variable in Base Workspace

Return a `Simulink.VariableUsage` object for a variable `k` in the base workspace.

```
var = Simulink.VariableUsage('k','base workspace');
```

You can use `var` as a filter for the `Simulink.findVars` function.

Represent All Variables in the Base Workspace

Return an array of `Simulink.VariableUsage` objects containing one object for each variable returned by the `whos` command in the base workspace.

```
vars = Simulink.VariableUsage(whos,'base workspace')
```

Represent All Variables in a Model 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')
```

Represent All Variables in a Mask Workspace

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”

Introduced in R2012b

intersect

Package: Simulink

Return 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.

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);
```

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.

See Also

Simulink.VariableUsage | Simulink.findVars | setdiff

Topics

“Model Exploration”
“Variables”

Introduced in R2012b

setdiff

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.

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');
differentVars = 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);
```

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.

See Also

`Simulink.VariableUsage` | `Simulink.findVars` | `intersect`

Topics

"Model Exploration"
"Variables"

Introduced in R2012b

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

This property is read-only.

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.

`EnableAccessToBaseWorkspace` — Specify whether models can use design data in the base workspace
`false` (default) | `true`

Whether linked models can use design data in the base workspace, specified as `true` or `false`.

To determine whether a dictionary provides access to the base workspace (including through referenced dictionaries), query the `HasAccessToBaseWorkspace` property.

For more information about this property, including restrictions that limit your ability to interact with base workspace data through the dictionary, see “Continue to Use Shared Data in the Base Workspace”.

Data Types: `logical`

`HasAccessToBaseWorkspace` — Query whether models can use design data in the base workspace

`0` (default) | `1`

This property is read-only.

Query whether models can use design data in the base workspace, returned as `1` (true) or `0` (false). If the dictionary or a referenced dictionary has the `EnableAccessToBaseWorkspace` property set to `true`, this property returns `1`.

Use this property to determine whether models that link to the dictionary can use design data in the base workspace. You do not need to query each referenced dictionary to determine whether it has the `EnableAccessToBaseWorkspace` property set to `true`.

Data Types: `logical`

`HasUnsavedChanges` — Indicator of unsaved changes

0 | 1

This property is read-only.

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.

`NumberOfEntries` — Total number of entries in data dictionary

integer

This property is read-only.

Total number of entries in data dictionary, including those in referenced dictionaries, returned as an integer.

Methods

addDataSource	Add reference data dictionary to parent data dictionary
close	Close connection between data dictionary and `Simulink.data.Dictionary` object
discardChanges	Discard changes to data dictionary
filepath	Full path and file name of data dictionary
getSection	Return `Simulink.data.dictionary.Section` object to represent data dictionary section
hide	Remove data dictionary from Model Explorer
importEnumTypes	Import enumerated type definitions to data dictionary
importFromBaseWorkspace	Import base workspace variables to data dictionary
listEntry	List data dictionary entries
removeDataSource	Remove reference data dictionary from parent data dictionary
saveChanges	Save changes to data dictionary
show	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
```

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'
```

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
```

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
```

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.

You cannot use the data dictionary programmatic interface (see “Store Data in Dictionary Programmatically”) to access the **Embedded Coder** section of a data dictionary. Instead, see Embedded Coder Dictionary.

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'
```

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)
```

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 | string array

Enumerated type definitions to import, specified as a cell array of character vectors or a string array. 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: {}
```

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 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 | string array

Names of specific base workspace variables to import, specified as the comma-separated pair consisting of `'varList'` and a cell array of character vectors or a string array. 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.

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 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')
```

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
```

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
```

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.

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.

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)
```

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'
```

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.sldd'

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 target data dictionary entries. Specify this option before any other search criteria or `options` arguments.
`'-and'`, `'-or'`, `'-xor'`, or `'-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 `find` to search for entries or values that are objects of the class or of any subclass derived from the class `className`. Specify `className` 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

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)) == 237
        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')
```

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.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 variable `fuelFlowValue`.

```
fuelFlowValue = getValue(fuelFlowObj)
```

```
fuelFlowValue =

   237
```

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'
```

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.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
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)
```

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)	If you do not specify the property `Headerfile`, export the data type definition to *`model`*`_types.h`, where *`model`* is the model name. If you specify `Headerfile`, import the data type definition from the specified header file.
`Exported`	Export the data type definition to a separate header file. If you do not specify the property `Headerfile`, the header file name defaults to *`type`*`.h`, where *`type`* is the data type name.
`Imported`	Import the data type definition from a separate header file. If you do not specify the property `Headerfile`, the header file name defaults to *`type`*`.h`, where *`type`* is the data type name.

`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'});
```

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'});
```

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'});
```

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.

You cannot use the data dictionary programmatic interface (see “Store Data in Dictionary Programmatically”) to access the **Embedded Coder** section of a data dictionary. Instead, see Embedded Coder Dictionary.

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

addEntry	Add new entry to data dictionary section
assignin	Assign value to data dictionary entry
deleteEntry	Delete data dictionary entry
evalin	Evaluate MATLAB expression in data dictionary section
exist	Check existence of data dictionary entry
exportToFile	Export data dictionary entries from section to MAT-file or MATLAB file
find	Search in data dictionary section
getEntry	Create `Simulink.data.dictionary.Entry` object to represent data dictionary entry
importFromFile	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'
```

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`.

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.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Assign the value 237 to an entry myAssignedEntry in the data dictionary myDictionary_ex_API.sldd. If an entry named myAssignedEntry is not in myDictionary_ex_API.sldd, create it.

```
assignin(dDataSectObj,'myAssignedEntry',237)
```

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')
```

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.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Execute the whos command in the Design Data section of myDictionary_ex_API.sldd.

```
evalin(dDataSectObj,'whos')
```

```
   Name                    Size               Bytes  Class                 Attributes

  fuelFlow               1x1                    8  double
  myRefEntry             1x1                    1  logical
  parameterGain37        1x1                  112  Simulink.Parameter
```

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
```

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.
```

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.sldd'`

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 `find` to search for entries or values that are objects of the class or of any subclass derived from the class `className`. Specify `className` 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')
```

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)
```

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.

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

Use `Simulink.DualScaledParameter` 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 inherits some properties from the `Simulink.Parameter` class. 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.

Creation

Create a `Simulink.DualScaledParameter` object:

- By using the Model Data Editor. Instead of creating a `Simulink.Parameter` object, create a `Simulink.DualScaledParameter` object. See “Interact with a Model That Uses Workspace Variables”.
- By using the Model Explorer:

  1. In the **Model Hierarchy** pane, select a workspace or data dictionary.
  2. On the toolbar, select **Add** > **Add Custom**.
  3. In the Model Explorer — Select Object dialog box, set **Object class** to `Simulink.DualScaledParameter`.
- By using the `Simulink.DualScaledParameter` function, described below.

Syntax

```
DSParam = Simulink.DualScaledParameter
```

Description

`DSParam = Simulink.DualScaledParameter` returns a `Simulink.DualScaledParameter` object with default property values.

Properties

For information about properties in the property dialog box of a `Simulink.DualScaledParameter` object, see “Simulink.DualScaledParameter Property Dialog Box”.

CalibrationValue — Calibration value of this parameter

[ ] (default) | finite, real, double number

Calibration value of this parameter, specified as a finite, real, double number. This value represents the value that you prefer to use.

Before specifying CalibrationValue, you must specify CalToMainCompuNumerator and CalToMainCompuDenominator to define the computation method. The parameter uses the computation method and the calibration value to calculate the main value that Simulink uses.

Corresponds to **Calibration value** in the property dialog box.

Example: 5.34

Data Types: double

CalibrationMin — Calibration minimum value of this parameter

[ ] (default) | finite, real, double, scalar number

Calibration minimum value of this parameter, specified as a finite, real, double, scalar number. The default value, [], means the minimum is unspecified.

Before specifying CalibrationMin, you must specify CalToMainCompuNumerator and CalToMainCompuDenominator 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.

Corresponds to **Calibration minimum** in the property dialog box.

Example: 10.51

Data Types: double

CalibrationMax — Calibration maximum value of this parameter

[ ] (default) | finite, real, double, scalar number

Calibration maximum value of this parameter, specified as a finite, real, double, scalar number. The default value, `[]`, means the maximum is unspecified.

Before specifying `CalibrationMax`, you must specify `CalToMainCompuNumerator` and `CalToMainCompuDenominator` 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.

Corresponds to **Calibration maximum** in the property dialog box.

Example: `-10.51`

Data Types: `double`

`CalToMainCompuNumerator` — Numerator coefficients of the computation method

`[]` (default) | finite, real, double scalar | finite, real, double vector

Numerator coefficients of the computation method, specified as a scalar number or vector of values for 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 `CalToMainCompuNumerator`, you cannot change it.

Corresponds to **CalToMain compute numerator** in the property dialog box.

Example: `[1 1]`

Example: `1`

Data Types: `double`

`CalToMainCompuDenominator` — Denominator coefficients of the computation method

`[]` (default) | finite, real, double scalar | finite, real, double vector

Denominator coefficients of the computation method, specified as a scalar number or vector of values for 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 `CalToMainCompuDenominator`, you cannot change it.

Corresponds to **CalToMain compute denominator** in the property dialog box.

Example: `[1 1]`

Data Types: `double`

`CalibrationName` — Name of the calibration parameter

`''` (empty character vector) (default) | character vector

Name of the calibration parameter, specified as a character vector.

Corresponds to **Calibration name** in the property dialog box.

Example: `'This is a calibration parameter.'`

Data Types: `char`

`CalibrationDocUnits` — Measurement units for this calibration parameter's value

`''` (empty character vector) (default) | character vector

Measurement units for this calibration parameter's value, specified as a character vector.

Corresponds to **Calibration units** in the property dialog box.

Example: `'Fahrenheit'`

Data Types: `char`

`IsConfigurationValid` — Information about validity of configuration

`true` (default) | `false`

This property is read-only.

Information about the validity of the object configuration, returned as `true` (valid) or `false` (invalid). If Simulink detects an issue with the configuration, it sets this field to `false` and provides information in the `DiagnosticMessage` property.

Corresponds to **Is configuration valid** in the property dialog box.

Data Types: `logical`

`DiagnosticMessage` — Diagnostic information about invalid configuration

`''` (empty character vector) (default) | character vector

This property is read-only.

Diagnostic information about an invalid object configuration, returned as a character vector. If you specify invalid property settings, Simulink displays a message in this field. Use the diagnostic information to help you fix an invalid configuration issue.

Corresponds to **Diagnostic message** in the property dialog box.

Data Types: `char`

Examples

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;
```

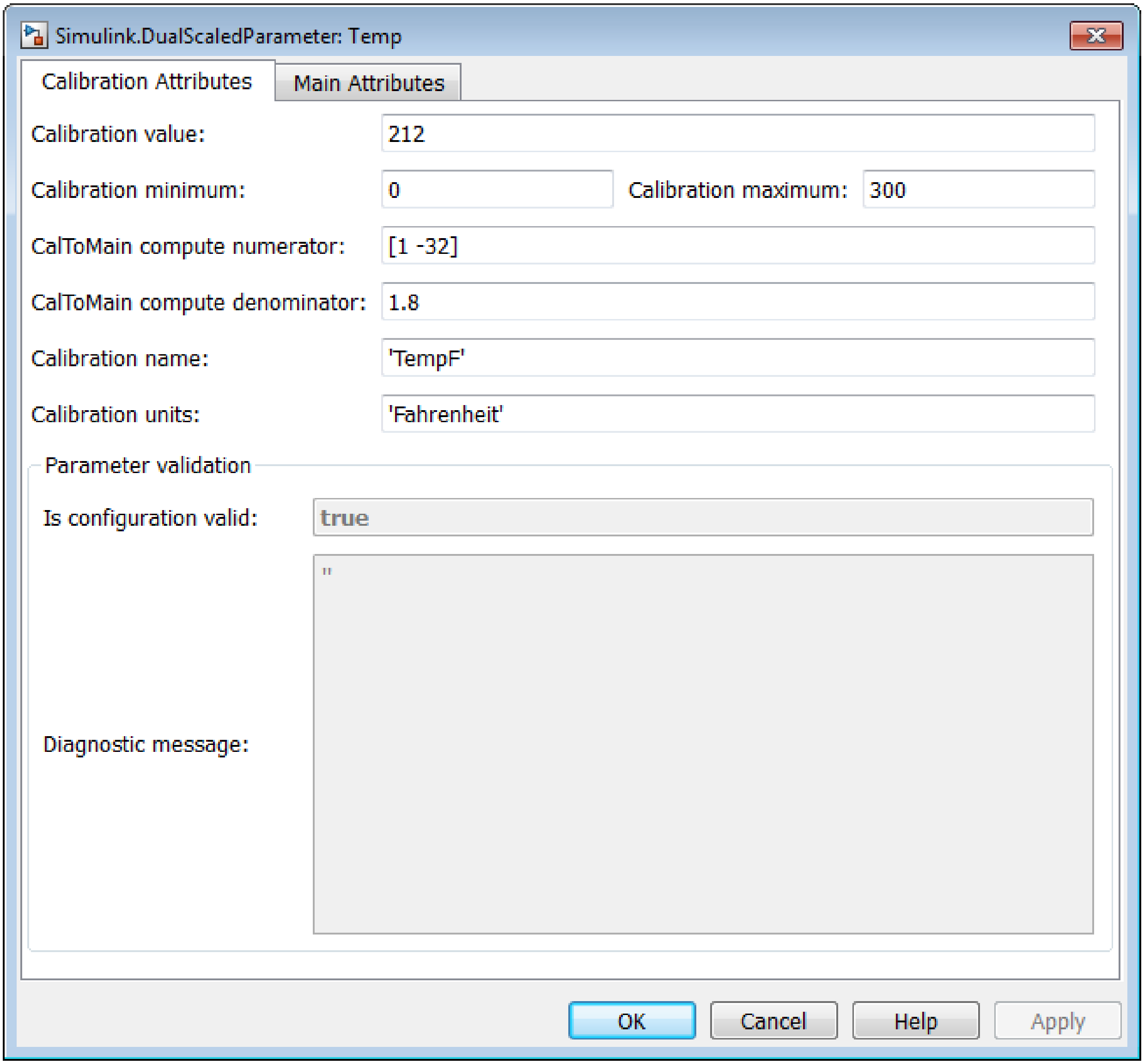
If you specify a calibration value outside this allowable range, Simulink generates a warning.

Specify the units that Simulink uses.

```
Temp.Unit = 'degC';
```

Open the Simulink.DualScaledParameter dialog box.

```
open Temp
```



Simulink.DualScaledParameter: Temp

Calibration value: 212

Calibration minimum: 0 Calibration maximum: 300

CalToMain compute numerator: [1 -32]

CalToMain compute denominator: 1.8

Calibration name: 'TempF'

Calibration units: 'Fahrenheit'

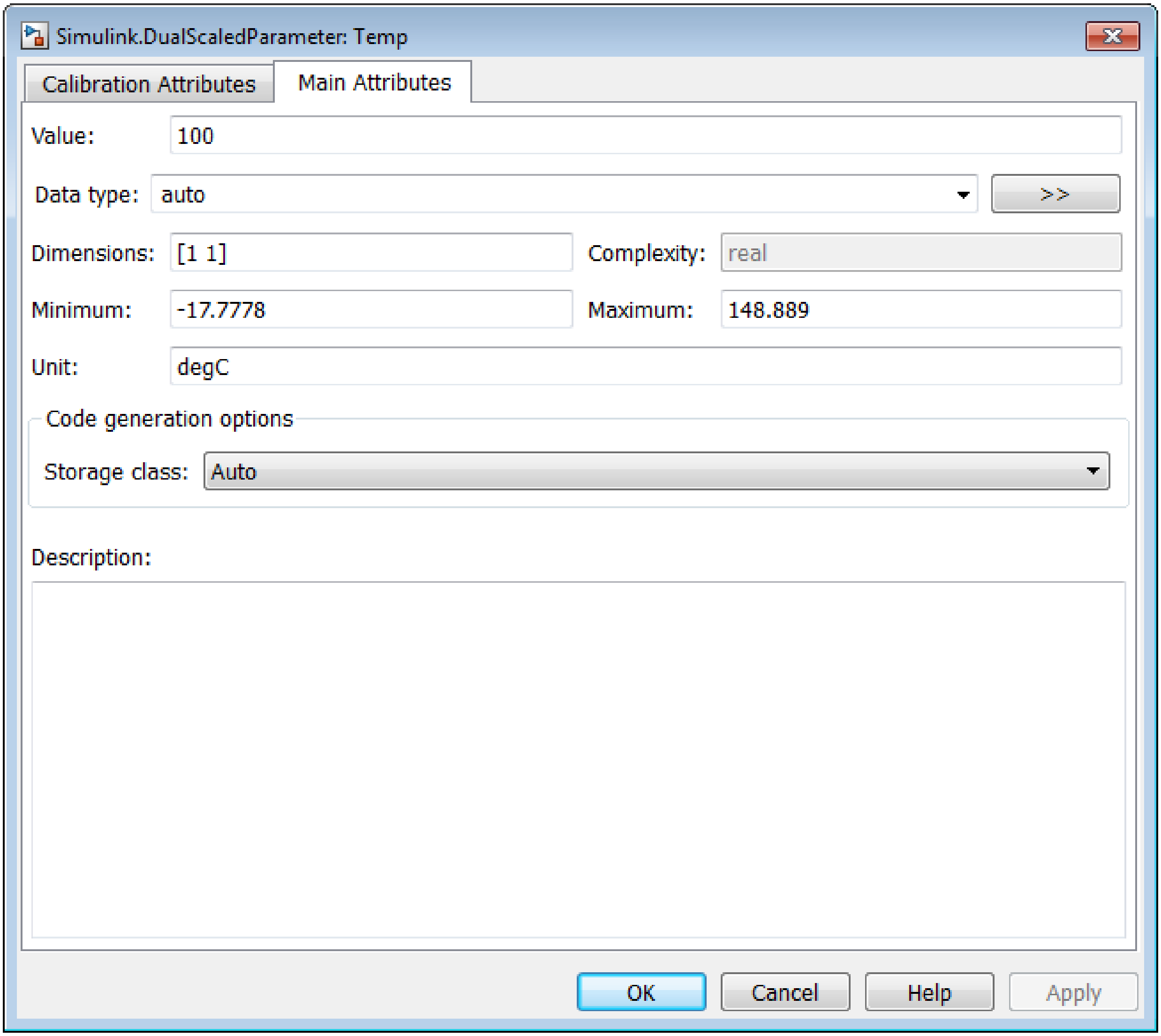
Parameter validation

Is configuration valid: true

Diagnostic message: ''

The **Calibration Attributes** tab displays the calibration value and the computation method that you specified.

In the dialog box, click the **Main Attributes** tab.



Simulink.DualScaledParameter: Temp

Value: 100

Data type: auto

Dimensions: [1 1]

Complexity: real

Minimum: -17.7778

Maximum: 148.889

Unit: degC

Code generation options

Storage class: Auto

Description:

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 | string scalar`

Default: `Empty character vector`

`Description`

Specifies the block description of the associated block.

Type: `character vector | string scalar`

Default: `Empty character vector`

`Help`

Specifies the help text that is displayed for the mask.

Type: `character vector|string scalar`

Default: `Empty character vector`

`Initialization`

Specifies the initialization commands for the associated block.

Type: `character vector|string scalar`

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|string scalar`

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
addParameterConstraint	Add parameter constraint to a mask
removeParameterConstraint	Delete a mask parameter constraint
removeCrossParameterConstraint	Delete a cross-parameter constraint
removeAllParameterConstraints	Delete all mask parameter constraints
removeAllCrossParameterConstraints	Delete all cross-parameter constraints from a mask
getParameterConstraint	Get mask parameter constraint properties
getCrossParameterConstraint	Get cross-parameter constraint
getAssociatedParametersOfConstraint	Get mask parameters associated with a constraint
addCrossParameterConstraint	Add cross-parameter constraint

See Also

Topics

“Control Masks Programmatically”
“Create 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 generates name for the 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 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: generated name

`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

`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.

   ```
   p.addParameter('Type','popup','TypeOptions',...
   {'Red' 'Blue' 'Green'},'Evaluate','off');
   ```

See Also

“Create 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

“Create 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

"Create 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

“Create 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

“Create 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 | string scalar

Type of dialog control element, specified

- `'panel'`
- `'group'`
- `'tabcontainer'`
- `'tab'`

- `'collapsiblepanel'`
- `'text'`
- `'image'`
- `'hyperlink'`
- `'pushbutton'`

`controlIdentifer` — Unique identifier for the element
character vector | string scalar

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

“Create 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

“Create 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

"Create 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

“Create 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

“Create 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

“Create 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 `controlIndentifier`, 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

`controlIdentifer` — Unique identifier for the element
character vector | string scalar

Programmatic identifier for the dialog control element of mask dialog box.

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

"Create 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

`controlIdentifer` — Unique identifier for the parameter
character vector | string scalar

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

“Create 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

“Create 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 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

“Create Block Masks” | `Simulink.Mask`

addParameterConstraint

Class: Simulink.Mask
Package: Simulink

Add parameter constraint to a mask

Syntax

```
paramConstraint = maskObj.addParameterConstraint(Name,Value)
```

Description

`paramConstraint = maskObj.addParameterConstraint(Name,Value)` adds a constraint to the specified mask. Constraints can only be associated to the Edit type mask parameters.

Input Arguments

`maskObj` — Block mask handle

mask object

Block mask handle, specified as a mask object. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information, see `Simulink.Mask.get`

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Name — Mask constraint name

character vector | string

Required field. Must be a valid MATLAB name. Specifies a name for the mask parameter constraint.

`Parameters` — Mask parameter name

cell array of character vector | string

Optional field. Specifies the **Edit** mask parameter name to which you want to associate the constraint.

`Rules` — Rule for mask parameter constraint

DataType | Dimension | Complexity | Sign | Finiteness | Minimum | Maximum | CustomConstraint

Required field. Rules are defined within curly braces. A constraint can have single or multiple rules.

Name-Value Pairs for Rules

Name	Value
`DataType`	`double`, `single`, `numeric`, `integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `enum`, `fixdt`
`Dimension`	`scalar`, `rowvectar`, `columnvectar`, `2dmatrix`, `ndmatrix`
`Complexity`	`real`, `complex`
`Sign`	`positive`, `negative`, `zero`
`Finiteness`	`finite`, `inf`, `-inf`, `NaN`
`Minimum`	character vector
`Maximum`	character vector
`CustomConstraint`	Valid MATLAB expression returning logical true or false.

Output Arguments

`paramConstraint` — Mask parameter constraint

character vector | string

Handle to the mask parameter constraint, returned as a character vector. You can associate a constraint to the mask parameter either during or after creating the constraint.

Examples

Create Mask Constraint with Single Rule

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Create mask constraint with single rule.
paramConstraint = maskObj.addParameterConstraint('Name','const2',...
'Parameters',{'Parameter2'}, 'Rules', {'DataType', 'uint8'})
```

```
ans = 

  Constraints with properties:

               Name: 'const2'
    ConstraintRules: [1×1 Simulink.Mask.ParameterConstraintRules]
```

Create Mask Constraint with Multiple Rules

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Create mask constraint with multiple rules.
paramConstraint = maskObj.addParameterConstraint('Name','const3',...
'Parameters',{'Parameter3'}, 'Rules', {{'DataType', 'uint8'},{'DataType', {'fixdt(1,8,4
```

```
ans = 

  Constraints with properties:

               Name: 'const3'
    ConstraintRules: [1×2 Simulink.Mask.ParameterConstraintRules]
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a

removeParameterConstraint

Class: Simulink.Mask
Package: Simulink

Delete a mask parameter constraint

Syntax

```
maskObj.removeParameterConstraint(paramConstraint)
```

Description

`maskObj.removeParameterConstraint(paramConstraint)` deletes the specified mask parameter constraint.

Input Arguments

`maskObj` — Block mask handle

mask object

Handle to the block mask, specified as mask object. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information, see `Simulink.Mask.get`.

Data Types: `char` | `cell`

`paramConstraint` — Mask constraint name

character vector | string

Name of the mask parameter constraint to be deleted, specified as character vector.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);
```

```
% Remove mask constraint with name 'const1'.
maskObj.removeParameterConstraint('const1')
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a

removeCrossParameterConstraint

Class: Simulink.Mask
Package: Simulink

Delete a cross-parameter constraint

Syntax

```
maskObj.removeCrossParameterConstraint(CrossConstraint)
```

Description

`maskObj.removeCrossParameterConstraint(CrossConstraint)` deletes the specified cross-constraint.

Input Arguments

`maskObj` — Block mask handle

mask object

Handle to the block mask, specified as mask object. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information, see `Simulink.Mask.get`.

Data Types: `char` | `cell`

`CrossConstraint` — Cross constraint name

character vector | string

Name of the cross-constraint to be removed, specified as character vector.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);
```

```
% Remove cross-constraint of name 'const1'.
maskObj.removeCrossParameterConstraint('const1')
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a

removeAllParameterConstraints

Class: Simulink.Mask
Package: Simulink

Delete all mask parameter constraints

Syntax

`maskObj.removeAllParameterConstraints()`

Description

`maskObj.removeAllParameterConstraints()` deletes all the parameter constraints from a mask.

Input Arguments

`maskObj` — Block mask handle
mask object

Handle to the block mask. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information see, `Simulink.Mask.get`

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Remove all constraints from the mask.
maskObj.removeAllParameterConstraints()
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a

removeAllCrossParameterConstraints

Class: Simulink.Mask
Package: Simulink

Delete all cross-parameter constraints from a mask

Syntax

```
maskObj.removeAllCrossParameterConstraints()
```

Description

`maskObj.removeAllCrossParameterConstraints()` deletes all cross-constraints from a mask.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Remove all cross constraints from the mask.
maskObj.removeAllCrossParameterConstraints()
```

See Also

"Create Block Masks" | `Simulink.Mask`

Introduced in R2018a

getParameterConstraint

Class: Simulink.Mask
Package: Simulink

Get mask parameter constraint properties

Syntax

```
paramConstraint = maskObj.getParameterConstraint(
paramConstraintName)
```

Description

`paramConstraint = maskObj.getParameterConstraint(paramConstraintName)` gets the properties of a mask parameter constraint.

Input Arguments

`maskObj` — Block mask handle
mask object

Handle to the block mask, specified as mask object. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information, see `Simulink.Mask.get`.

Data Types: `char` | `cell`

`paramConstraintName` — Mask constraint name
character vector | string

Name of the constraint of which you want get the properties, specified as character vector.

Output Arguments

paramConstraint — Mask constraint property
cell array of character vector

Constraint properties, returned as a cell array.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Find parameters associated with the constraint.
paramConstraint = maskObj.getParameterConstraint('const3')


ans =

  Constraints with properties:

             Name: 'const3'
  ConstraintRules: [1×2 Simulink.Mask.ParameterConstraintRules]
```

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2018a

getCrossParameterConstraint

Class: Simulink.Mask
Package: Simulink

Get cross-parameter constraint

Syntax

```
CrossConstraint = maskObj.getCrossParameterConstraint(
CrossConstraintName)
```

Description

`CrossConstraint = maskObj.getCrossParameterConstraint( CrossConstraintName)` gets the properties of a cross parameter constraint on a mask. Apply a cross-parameter constraint to specify rules between mask parameter values.

Input Arguments

`maskObj` — Block mask handle

mask object

Handle to the block mask, specified as mask object. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information, see `Simulink.Mask.get`.

Data Types: `char` | `cell`

`CrossConstraintName` — Cross-constraint name

character vector | string

Name of the cross-parameter constraint for which you get the constraint properties, specified as the mask object.

Output Arguments

CrossConstraint — Cross-constraint property

cell array

Cross-parameter constraint properties, returned as a cell array.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get cross constraint.
CrossConstraint = maskObj.getCrossParameterConstraint('crossparam1')
```

```
ans =

  CrossParameterConstraints with properties:
            Name: 'crossparam1'
            Rule: 'Parameter2 > Parameter3'
    ErrorMessage: ''
```

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2018a

getAssociatedParametersOfConstraint

Class: Simulink.Mask
Package: Simulink

Get mask parameters associated with a constraint

Syntax

```
maskParam = maskObj.getAssociatedParametersOfConstraint(
paramConstraintName)
```

Description

`maskParam = maskObj.getAssociatedParametersOfConstraint( paramConstraintName)` gets the parameters that are associated with a mask constraint.

Input Arguments

`maskObj` — Block mask handle

mask object

Handle to the block mask, specified as mask object. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information, see `Simulink.Mask.get`.

Data Types: `char` | `cell`

`paramConstraintName` — Mask constraint name

character vector | string

Name of the constraint for which you want to find the associated mask parameters, specified as character vector.

Output Arguments

`maskParam` — Mask parameter name
cell array of character vector

Mask parameter, specified as a cell array.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Find parameters associated with the constraint.
maskParam = maskObj.getAssociatedParametersOfConstraint('const3')
```

```
ans =

  1×1 cell array

    {'Parameter1'}
```

See Also

"Create Block Masks" | `Simulink.Mask`

Introduced in R2018a

addCrossParameterConstraint

Class: Simulink.Mask
Package: Simulink

Add cross-parameter constraint

Syntax

```
CrossConstraint = maskObj.addCrossParameterConstraint(Name,Value)
```

Description

`CrossConstraint = maskObj.addCrossParameterConstraint(Name,Value)` adds a constraint among parameters of a mask.

Input Arguments

`maskObj` — Block mask handle
mask object

Block mask handle, specified as a mask object. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information, see `Simulink.Mask.get`.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the property name and `Value` is the corresponding value. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Name — Cross-parameter constraint name
character vector | string

Cross-parameter constraint name, specified as a character vector. Must be a valid MATLAB value.

`Rule` — Cross-parameter constraint rule

MATLAB expression | string

Cross-parameter constraint rule, specified as a valid MATLAB expression that returns logical true or false. You can specify multiple rules by separating them with a logical operators like &&. For example, `parameter1 > parameter2 && parameter2 > parameter3`. Here, `parameter1`, `parameter2` and `parameter3` are parameters of a mask.

`ErrorMessage` — Error message

character vector | string

Optional field. Specifies the error message to be displayed when the cross parameter constraint rule is not satisfied. You can specify the error message as character vector or as a message catalog ID. If you use the message catalog ID to specify an error, the error message must not have any holes in it. Simulink displays a default error message if no user-defined error message is found.

Output Arguments

`CrossConstraint` — Cross parameter constraint

cell array

Handle to the cross-parameter constraint, returned as a cell array.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Add cross parameter constraint.
CrossConstraint = maskObj.addCrossParameterConstraint('Name','crossconstraint1',...
'Rule','upperbound > lowerbound','ErrorMessage','Incorrect value specified.')
```

```
ans =
  CrossParameterConstraints with properties:

            Name: 'crossconstraint1'
            Rule: 'upperbound > lowerbound'
    ErrorMessage: 'Incorrect value specified.'
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a

Simulink.Mask.Constraints class

Package: Simulink.Mask

Create Mask Constraint

Description

Use an instance of the `Simulink.Mask.Constraint` to add or remove a parameter constraint rule.

Properties

`DataType`

Specifies the data type associated with the constraint rule.

Type: `double, single, numeric, integer, int8, uint8, int16, uint16, int32, uint32, boolean, enum, fixdt`

Default: `Empty`

`Dimension`

Specifies the dimension associated with the constraint rule.

Type: `scalar, rowvector, columnvector, 2dmatrix, ndmatrix`

Default: `Empty`

`Complexity`

Specifies the complexity associated with the constraint rule.

Type: `real, complex`

Default: `Empty`

`Sign`

Specifies the sign associated with the constraint rule.

Type: `positive, negative, zero`

Default: `Empty`

`Finiteness`

Specifies the finiteness associated with the constraint rule.

Type: `finite, inf, -inf, NaN`

Default: `Empty`

`Range`

Specifies the range associated with the constraint rule.

Type: `Minimum, Maximum`

Default: `Empty`

`CustomConstraint`

Specifies the error message associated with the constraint rule.

Type: `Valid MATLAB expression`

Default: `Empty`

Methods

addParameterConstraintRule	Add rules to a parameter constraint
removeParameterConstraintRule	Delete a mask parameter constraint rule

See Also

Introduced in R2018a

addParameterConstraintRule

Class: Simulink.Mask.Constraints
Package: Simulink.Mask

Add rules to a parameter constraint

Syntax

```
paramConstRule = paramConstraint.addParameterConstraintRule(
Name,Value)
```

Description

`paramConstRule = paramConstraint.addParameterConstraintRule(
Name,Value)` adds rule to a parameter constraint.

Input Arguments

`paramConstraint` — Handle to mask constraint
constraint object

Handle to the mask parameter constraint for which you want to add constraint rules, specified as constraint object.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Name-Value Pair for Rules

Name	Value
`DataType`	`double`, `single`, `numeric`, `integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `enum`, `fixdt`
`Dimension`	`scalar`, `rowvectar`, `columnvectar`, `2dmatrix`, `ndmatrix`
`Complexity`	`real`, `complex`
`Sign`	`positive`, `negative`, `zero`
`Finiteness`	`finite`, `inf`, `-inf`, `NaN`
`Minimum`	`string`
`Maximum`	`string`
`CustomConstraint`	Valid MATLAB expression

Output Arguments

`paramConstRule` — Mask constraint rule
cell array

Mask constraint rule, specified as as a cell array.

Examples

```
% Get mask constraint handle
paramConstraint = maskObj.getParameterConstraint('const3');

% Add rules to the constraint.
paramConstRule = paramConstraint.addParameterConstraintRule('DataType','int8')
```

```
ans =

  ParameterConstraintRules with properties:

            DataType: 'int8'
```

```
           Dimension: {0×1 cell}
          Complexity: {0×1 cell}
                Sign: {0×1 cell}
          Finiteness: {0×1 cell}
             Minimum: ''
             Maximum: ''
    CustomConstraint: ''
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a

removeParameterConstraintRule

Class: Simulink.Mask.Constraints
Package: Simulink.Mask

Delete a mask parameter constraint rule

Syntax

```
paramConstraint.removeParameterConstraintRule(RuleIndex)
```

Description

`paramConstraint.removeParameterConstraintRule(RuleIndex)` deletes the specified constraint rule from a mask parameter constraint.

Input Arguments

`paramConstraint` — Handle to constraint

constraint object

Handle to mask parameter constraint of which you want to remove constraint rule, specified as an object.

Data Types: `char` | `cell`

`RuleIndex` — Constraint rule index

integer

Index number of the mask constraint rule, specified as an integer.

Examples

```
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);
```

```
% Get mask constraint handle.
paramConstraint = maskObj.getParameterConstraint('const3');

% Remove mask constraint rule.
paramConstraint.removeParameterConstraintRule(1)
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a

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 the mask parameter. The mask parameter name must not match the built-in parameter name.

Type: `character vector`

Default: `Auto generated`

`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`

`DialogControl`

Specifies the layout options of mask dialog.

Type: `Class of type Simulink.dialog.parameter.<Typeofparameter>`

Values:
`'edit'|'checkbox'|'popup'|'min'|'max'|'promote'|'combobox'|'radiobutton'|'unidt'|'slider'|'dial'|'spinbox'`

Default: `Edit`

`ConstraintName`

Indicates the constraint associated with a parameter. To associate a constraint programmatically the constraint must be already available. To associate a constraint from a mat file use the format `<matfilename>:<constraintname>`.

Type: `character vector`

Default: `Empty character vector`

Methods

set	Set properties of mask parameters

See Also

Topics

"Control Masks Programmatically"
"Create 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 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

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.

   ```
   a.set('Type','popup','TypeOptions',{'Red' 'Blue' 'Green'},...
   'Evaluate','off');
   ```

See Also

"Create 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` | “Create 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 filed.

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'`

AlignPrompts

Allows you to align the parameters vertically on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'off'`

DialogControls

Specifies the child dialog controls contained in the container.

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.Tab` | `Simulink.dialog.TabContainer` | `Simulink.dialog.Control` | “Create 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'`

`controlIdentifer` — 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 | “Create 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 `controlIndentifier`, 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

`controlIdentifer` — 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` | "Create 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` | “Create 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` | “Create 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` | “Create 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` | “Create 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

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.Tab` | `Simulink.dialog.Container` | `Simulink.dialog.Control` | “Create 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` | “Create 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` | “Create 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` | “Create 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` | “Create 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` | “Create 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 the MATLAB path that has the 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 the MATLAB path that has the same name 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 | string

Names of the model in the path, specified as a string or 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 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

run	Perform checks on S-functions
generateReport	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: [1×1 mex.CompilerConfiguration]
                Data: [4×4 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 | string

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 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 | string

Search paths to extra S-function source files that are referenced by `SfcnFile`, specified as a cell array or a string.

`ExtraSrcFileList` — Extra source file list

cell array of character vectors | string

List of extra S-function source files, specified as a cell array or string.

`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 | string

Specify paths to include folders for header files, specified as a cell array or a string.

`LibFileList` — External libraries list

cell array of character vectors | string

List of external library files used for building, specified as a cell array or string.

`LibPaths` — Search paths for external libraries

cell array of character vectors | string

Search paths for external library files used for building, specified as a cell array or a string.

`PreProcDefList` — Preprocessor directives

cell array of character vectors | string

List of the preprocessor directives, specified as a cell array or a string.

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 — Polyspace Code Prover 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

current working directory (default) | character array

Path to the generated report directory.

`ModelSimTimeOut` — Maximum model simulation time

`10` (default) | scalar

Maximum model simulation time in seconds.

`EnableUsePublishedOnly` — Use of documented APIs

`False` (default) | `True`

Check to indicate if any undocumented S-function APIs are used in the code.

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
character vector | string vector

A Simulink model or library in path specified as a string or a character vector.

Output Arguments

`sfuns` — A list of all eligible S-functions
cell array of character vectors

Eligible S-functions in the model, specified as a cell array of character vectors.

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.
```

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);
```

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 `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_mdlref_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.

```
el = logsout.get(1);
```

Access Dataset Elements with Characters

Access the element whose name matches `name`.

```
el = logsout.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 = logsout.get({'my_name'});
```

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 `logsout` dataset.

```
logsout{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_mdlref_counter_bus')));
sim('ex_bus_logging')
el_names = topOut.getElementNames
```

```
el_names = 

    'COUNTERBUS'
    'OUTPUTBUS'
    'INCREMENTBUS'
    'inner_bus'
```

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 5-1060.

`-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')
```

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 `logsout` dataset.

```
logsout{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()
```

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');
```

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
```

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 `logsout` dataset.

```
logsout{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

getHardwareImplementation	Get handle of copy of hardware implementation object
getStdLibInfo	Get standard library information
getTargetLang	Get target code generation language
getToolchainInfo	Returns handle of copy of toolchain information object
isCodeGenTarget	Determine if build configuration represents specified target
isMatlabHostTarget	Determine if hardware implementation object target is MATLAB host computer

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(buildContext)
            if  buildContext.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end

        function updateBuildInfo(buildInfo, buildContext)
            % Get file extensions for the current platform
            [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();

            % Add file paths
            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 for packaging
            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
                % Add the required include statements to the generated function code
                coder.cinclude('adder.h');
                coder.cinclude('adder_initialize.h');
                coder.cinclude('adder_terminate.h');
                c = 0;

                % Because MATLAB Coder generated adder, use the
                % housekeeping functions before and after calling
                % adder with coder.ceval.

                coder.ceval('adder_initialize');
                c = coder.ceval('adder', a, b);
                coder.ceval('adder_terminate');
            end
        end
    end
end
```

See Also

`coder.ExternalDependency` | `coder.HardwareImplementation` | `coder.target`

Topics

"Develop Interface for External C/C++ Code" (MATLAB Coder)
"Build Process Customization" (MATLAB Coder)

Introduced in R2013b

coder.ExternalDependency class

Package: coder

Interface to external code

Description

`coder.ExternalDependency` is an abstract class for developing an interface between external code and MATLAB code intended for code generation. You can 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.

To define a class derived from `coder.ExternalDependency`, create a subclass. For example:

```
classdef myClass < coder.ExternalDependency
```

You must define all of the methods listed in “Methods” on page 5-1077. These methods are static and are not compiled. The code generator invokes these methods in MATLAB after code generation is complete to configure the build for the generated code. The `RTW.BuildInfo` and `coder.BuildConfig` objects that describe the build information and build context are automatically created during the build process. The `updateBuildInfo` method provides access to these objects. For more information on build information customization, see “Build Process Customization” (MATLAB Coder).

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.

Methods

getDescriptiveName	Return descriptive name for external dependency
isSupportedContext	Determine if build context supports external dependency
updateBuildInfo	Update build information

Examples

Encapsulate the Interface to an External C Dynamic 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(buildContext)
    if  buildContext.isMatlabHostTarget()
        tf = true;
    else
        error('adder library not available for this target');
    end
end

function updateBuildInfo(buildInfo, buildContext)
    % Get file extensions for the current platform
    [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();

    % Add file paths
    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 for packaging
    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
        % Add the required include statements to the generated function code
        coder.cinclude('adder.h');
        coder.cinclude('adder_initialize.h');
        coder.cinclude('adder_terminate.h');
        c = 0;
```

```
                % Because MATLAB Coder generated adder, use the
                % housekeeping functions before and after calling
                % adder with coder.ceval.

                coder.ceval('adder_initialize');
                c = coder.ceval('adder', a, b);
                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

“Develop Interface for External C/C++ Code” (MATLAB Coder)
“Build Process Customization” (MATLAB Coder)
“Integrate External/Custom Code” (MATLAB Coder)

Introduced in R2013b

TimeScopeConfiguration

Control Scope block 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.

Properties

Name — Title on a scope window

block name (default) | character vector

Title on a scope window, specified as a character vector.

`NumInputPorts` — Number of input ports

`'1'` (default) | character vector

Number of input ports on a scope block, specified as a character vector. The maximum number of input ports is 96.

UI Use

Select **File** > **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. For multi-column layouts, the displays are numbered down and then across.

Dependency

Setting this property selects the display for setting the properties `ShowGrid`, `ShowLegend`, `Title`, `PlotAsMagnitudePhase`, `YLabel`, and `YLimits`.

UI Use

Open the **Configuration Properties**. On the **Display** tab, set **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`

UI Use

Open the **Configuration Properties**. On the **Main** tab, set **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'`.

UI Use

Open the **Configuration Properties**. On the **Main** tab, set **Number of updates**.

`DataLogging` — Save scope data

`false` (default) | `true`

Set this property to `true` to save scope data to a variable in the MATLAB workspace.

This property does not apply to floating scopes and scope viewers.

Dependency

If this property is set to `true`, you must also specify the properties `DataLoggingVariableName` and `DataLoggingSaveFormat`.

UI Use

Open the **Configuration Properties**. On the **Logging** tab, set **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 a character vector. This property does not apply to floating scopes and scope viewers.

Dependency

Activate this property by setting `DataLogging` to `true`.

UI Use

Open the **Configuration Properties**. On the **Logging** tab, set **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 multirate 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 multirate data.

This property does not apply to floating scopes and scope viewers.

Dependency

Activate this property by setting `DataLogging` to `true`.

UI Use

Open the **Configuration Properties**. On the **Logging** tab, set **Save format**.

`DataLoggingLimitDataPoints` — Limit buffered data

`false` (default) | `true`

Set to `true` to limit buffered data before plotting and saving data.

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`.

UI Use

Open the **Configuration Properties**. On the **Logging** tab, set **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 that are plotted and saved are from the end of a simulation. For example, setting this property to 100 saves the last 100 data points.

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`.

UI Use

Open the **Configuration Properties**. On the **Logging** tab, set the text box to the right of **Limit data points to last**.

`DataLoggingDecimateData` — Reduce scope data

`false` (default) | `true`

Set this property to `true` to reduce scope data before plotting and saving.

Dependency

If this property is set to `true`, you must also specify the `DataLoggingDecimation` property.

UI Use

Open the **Configuration Properties**. On the **Logging** tab, set **Decimation**.

`DataLoggingDecimation` — Decimation factor

`'1'` (default) | character vector

Decimation factor applied to the signal data 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`.

UI Use

Open the **Configuration Properties**. On the **Logging** tab, set the text box to the right of **Decimation**.

`FrameBasedProcessing` — Frame-based processing of signals

`false` (default for Time Scope block) | `true` (default for Scope block)

Set this property to `true` to process signals as frame-based.

- `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.

UI Use

Open the **Configuration Properties**. On the **Main** tab, set **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 16-by-16.

- 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.

UI Use

Open the **Configuration Properties**. On the **Main** tab, select the **Layout** button.

`MaximizeAxes` — Maximize size of signal plots

`'Auto'` (default) | `'On'` | `'Off'`

Specify whether or not to maximize the size of signal plots:

- `'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.

UI Use

Open the **Configuration Properties**. On the **Main** tab, set **Maximize axes**.

`MinimizeControls` — Hide menu and toolbar

`false` (default) | `true`

Set this property to `true` to hide the menu and toolbar.

If you dock the scope, this property is inactive.

`OpenAtSimulationStart` — Open scope when starting simulation

`true` (default for Time Scope) | `false` (default for Scope)

Set this property to `true` to open the scope when the simulation starts.

UI Use

Select **File** > **Open at Start of Simulation**

`PlotAsMagnitudePhase` — Magnitude and phase plots

`false` (default) | `true`

Specify whether or not to display the magnitude and phase plots:

- `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 the `ActiveDisplay` property before setting this property.

UI Use

Open the **Configuration Properties**. On the **Display** tab, set **Plot signals as magnitude and phase**.

`Position` — Size and location of the 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`

Set this property to true to display vertical and horizontal grid lines.

Dependency

Set the `ActiveDisplay` property before setting this property.

UI Use

Open the **Configuration Properties**. On the **Display** tab, set **Show grid**.

`SampleTime` — Time interval

`'-1'` (default) | 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.

UI Use

Open the **Configuration Properties**. On the **Main** tab, set **Sample Time**.

`ShowLegend` — Signal legend

`false` (default) | `true`

Set this property to `true` to display the legend.

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 the `ActiveDisplay` property before setting this property.

UI Use

Open the **Configuration Properties**. On the **Display** tab, set **Show legend**.

`ShowTimeAxisLabel` — Display or hide x-axis labels

`true` (default for Time Scope block) | `false` (default for Scope block)

Set this property to `true` to display the *x*-axis labels.

Dependency

Set the `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.

UI Use

Open the **Configuration Properties**. On the **Time** tab, set **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 displays.
- `'Bottom'` — Display *x*-axis labels only on the bottom display.
- `'None'` — Do not display labels and deactivate `ShowTimeAxisLabel` property.

Dependency

Set the `ActiveDisplay` property before specifying this property.

Set `ShowTimeAxisLabel` to `true` and set `Maximize axes` to `'Off'`.

UI Use

Open the **Configuration Properties**. On the **Time** tab, set **Time-axis labels**.

`TimeDisplayOffset` — x-axis range offset

`'0'` (default) | character vector

x-axis range offset number, specified as a character vector. For input signals with multiple channels, enter a scalar or vector of offsets.

- Scalar — Offset all channels of an input signal by the same value.
- Vector — Independently offset the channels.

UI Use

Open the **Configuration Properties**. On the **Time** tab, set **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 — Any value less than the total simulation time specified as a character vector.
- `'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`.

UI Use

Open the **Configuration Properties**. On the **Time** tab, set **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.

UI Use

Open the **Configuration Properties**. On the **Time** tab, set **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.

UI Use

Open the **Configuration Properties**. On the **Time** tab, set **Time units**.

`Title` — Title for display

`'%<SignalLabel>'` (default) | character vector

Title for a display, specified as a character vector. The default value `%<SignalLabel>` uses the input signal name for the title.

Dependency

Set the `ActiveDisplay` property before setting this property.

UI Use

Open the **Configuration Properties**. On the **Display** tab, set **Title**.

`Visible` — Visibility of scope window

`true (default)` | `false`

Set this property to true to make the scope window visible.

`YLabel` — Y-axis label

`''` (default) | character vector

y-axis label for active display, specified as a character vector.

Dependency

Set the `ActiveDisplay` property before setting this property.

If `PlotAsMagnitudePhase` is `true`, the value of `YLabel` is hidden and plots are labeled `Magnitude` and `Phase`.

UI Use

Open the **Configuration Properties**. On the **Display** tab, set **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 the `ActiveDisplay` property 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]`.

UI Use

Open the **Configuration Properties**. On the **Display** tab, set **Y-limits (Minimum)** and **Y-limits (Maximum)**.

Examples

Create Scope Configuration Object

This example creates a scope configuration object using the `'vdp'` model which models the van der Pol equation.

```
open_system('vdp')
myScopeConfiguration = get_param('vdp/Scope','ScopeConfiguration');
myScopeConfiguration.NumInputPorts = '2';
```

See Also

Floating Scope | Scope | Time Scope

Topics

“Control Scope Blocks Programmatically”

Introduced in R2013a

Simulink.Simulation.Job class

Package: Simulink

Execute batchsim to create a Simulink.Simulation.Job object, simJob

Description

Execute batchsim command with a parallel pool to create a Simulink.Simulation.Job object. The batchsim command offloads multiple simulations to run in batche using the inputs specified with an array of Simulink.SimulationInput objects. You can use this object to monitor the status of a batch job, fetch outputs of a completed batch job, or cancel one or more jobs.

The batchsim command uses the Parallel Computing Toolbox™ license to run the simulations on compute cluster. If a parallel pool cannot be created, batchsim runs the simulations in serial . In the absence ofParallel Computing Toolbox license, batchsim errors out.

Construction

simJob = batchsim(in) creates a Simulink.Simulation.Job object, simJob, while running multiple simulations in batches on a compute cluster using the inputs specified in the Simulink.SimulationInput object, in.

createSimulationJob(batchJob) creates a Simulink.Simulation.Job object from a parallel.job object, batchJob. Use the createSimulationJob command to retrieve the job object returned by the batchsim command.

Input Arguments

in — Simulink.SimulationInput object array
object (default) | array

A Simulink.SimulationInput object or an array of Simulink.SimulationInput objects used to run multiple simulations. Specify parameters and values of a model to run multiple simulations without making it dirty.

Example: `in = Simulink.SimulationInput('vdp')`, `in(1:10) = Simulink.SimulationInput('vdp')`

Properties

`AdditionalPaths` — Folders to add to MATLAB search path of workers

character vector | string | string array | array of character vectors

Specified folders to add to MATLAB search path of parallel workers.

This property is read-only.

`AttachedFiles` — Files and folders sent to the workers

character vector | string | string array | array of character vectors

Files and folders to send to the parallel workers.

This property is read-only.

`AutoAddClientPath` — Whether user-added entries on client path are added to each worker path

true (default) | false

Whether user-added entries on the client path are added to each parallel worker path at the start of the batch job, specified as true or false.

`CreateDateTime` — Date and time of `simJob` creation

datetime object

Date and time at which the batch job was created, specified as a datetime object.

This property is read-only.

`EnvironmentVariables` — Environment variables sent to the workers

character vector | string | string array | array of character vectors

Defines the names of environment variables that are copied from a client session to the parallel workers.

This property is read-only after job submission.

`FinishDateTime` — Date and time of `simJob` completion

datetime object

Date and time at when the batch job completes execution, specified as a datetime object.

This property is read-only.

`ID` — Numeric identifier of the `simJob` object

scalar integer

ID of the `future` object, specified as a scalar integer.

This property is read-only.

`Name` — `simJob` name

string | character vector

Name of the `job` object, specified as a string.

`Parent` — Cluster object containing `simJob`

`parallel.cluster` object

Cluster object that contains the `Simulink.Simulation.Job` object, `simJob`.

`StartDateTime` — Date and time when `simJob` starts running

datetime object

Date and time when the `Simulink.Simulation.Job` starts running, specified as a datetime object.

This property is read-only.

`State` — Current state of `future` object array

`'pending'` | `'queued'` | `'running'` | `'finished'` | `'failed'`

Current state of `future` object array, specified as `'pending'`, `'queued'`, `'running'`, `'finished'`, or `'failed'`.

This property is read-only.

`SubmitDateTime` — Date and time of `simJob` submission

datetime object

Date and time when the `Simulink.Simulation.Job` is submitted, specified as a datetime object.

This property is read-only.

Tag — Label associated with `simJob`
string

Label associated with `Simulink.Simulation.Job` object.

Type — Job Type
`independent` | `pool`

Type of the `Simulink.Simulation.Job` object, specified as `independent` or `pool`.

`UserData` — Data associated with `simJob`
string

Stores any data associated with a job object. The data is stored in a client MATLAB session, and is not available on the workers.

`UserName` — Name of the user who creates the `simJob` object
string | character vector

Name of the user who creates the `Simulink.Simulation.Job` object.

Methods

Method	Purpose
`cancel`	Cancel a pending, queued, or running `Simulink.Simulation.Job` object
`diary`	Display or save Command Window text of batch job
`fetchOutputs`	Retrieve an array of `Simulink.SimulationOutput` objects from all simulations in `Simulink.Simulation.Job`
`listAutoAttachedFiles`	List of files automatically attached to job, task, or parallel pool

Method	Purpose
wait	Wait for Simulink.Simulation.Job object to change state

Examples

Run Parallel Simulations With batchsim to Create Simulink.Simulation.Job

This example shows how to run parallel simulations in batch using the sldemo_househeat model. batchsim offloads simulations to the compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

Observe the model behavior for different temperature set points.

1. Open the model.

```
open_system('sldemo_househeat');
```

2. Define a set of values for different temperatures.

```
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

3. Using the setPointValues, initialize an array of Simulink.SimulationInput objects.

```
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',...
        'Value',num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. NumWorkers displays how many workers are running the simulations – the number of workers specified in the 'Pool' argument plus an additional head worker.

```
simJob = batchsim(in,'Pool',3)

                  ID: 1
                Type: pool
          NumWorkers: 4
            Username: #####
               State: running
      SubmitDateTime: ##-###-#### ##:##:##
       StartDateTime: 
    Running Duration: 0 days 0h 0m 0s
```

See Also

Functions
batchsim | cancel | diary | fetchOutputs | getSimulationJobs | listAutoAttachedFiles | parcluster | parsim | wait

Classes
Simulink.Simulation.Job | Simulink.SimulationInput | parallel.Cluster

Topics

“Multiple Simulation Workflows”
“Job Monitor” (Parallel Computing Toolbox)
“Batch Processing” (Parallel Computing Toolbox)

Introduced in R2018b

cancel

Package: Simulink

Cancel a pending, queued, or running `Simulink.Simulation.Job` object

Syntax

```
cancel(simJob)
cancel(simJob,'Message')
```

Description

`cancel(simJob)` stops the `Simulink.Simulation.Job` object, `simJob`, that is currently in `'pending'`, `'queued'`, or `'running'` state.

`cancel(simJob,'Message')` stops the `Simulink.Simulation.Job` object, `simJob`, that is in `'pending'`, `'queued'`, or `'running'` state and displays a user-specified message.

The `State` property of `Simulink.Simulation.Job` object is set to finished, and other pending simulations are canceled. Canceling a job object disables you to fetch results from it. A canceled job object cannot be started again.

Note On canceling the job, the results of the completed simulations in the job are also lost.

Examples

Cancel a Batch Job

This example shows how to use the `cancel` method on a `simJob` object to stop simulations. The example runs several simulations of the `vdp` model, varying the value of the gain Mu.

1. Open the model and define a vector of Mu values.

```
open_system('vdp');
mu_Values = [0.5:0.25:1000];
muVal_Length = length(mu_Values);
```

2. Using mu_Values, initialize an array of Simulink.SimulationInput objects.

```
in(1:muVal_Length) = Simulink.SimulationInput('vdp');
for i = 1:1:muVal_Length
    in(i) = in(i).setBlockParameter('vdp/Mu',...
        'Gain',num2str(mu_Values(i)));
end
```

3. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The simulations are offloaded onto the default cluster profile.

```
simJob = batchsim(in,'Pool',3);
```

4. Now, assume that you want to run simulations with different values of Mu and cancel the ongoing simulations.

```
cancel(simJob)
```

Input Arguments

simJob — Simulink.Simulation.Job object

object

A Simulink.Simulation.Job object. To create a simJob, run batchsim.

Example: simJob = batchsim(in,'Pool',6)

See Also

Functions

batch | batchsim | diary | fetchOutputs | getSimulationJobs | listAutoAttachedFiles | parsim | wait

Classes
`Simulink.Simulation.Job` | `Simulink.SimulationInput`

Topics

"Multiple Simulation Workflows"
"Batch Processing" (Parallel Computing Toolbox)

Introduced in R2018b

diary

Package: Simulink

Display or save Command Window text of `Simulink.Simulation.Job` object

Syntax

```
diary(simJob)
diary(simJob,'filename')
```

Description

`diary(simJob)` displays the Command Window output from the `Simulink.Simulation.Job` object, `simJob`, in the MATLAB Command Window. The Command Window output is captured only if the `batchsim` command includes the `'CaptureDiary'` argument with a value of `true`.

`diary(simJob,'filename')` causes the Command Window output from the batch job to be appended to the specified file. Open the file, `filename`, with any text editor.

Examples

Display the Diary of simJob

This example uses `sldemo_househeat` model to show how to display the diary of the `Simulink.Simulation.Job` object, `simJob`. To create a `simJob`, you run parallel simulations using the `batchsim` command.

1. Open the model.

```
open_system('sldemo_househeat');
```

2. Define a set of values for different temperatures.

```
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

3. Using `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.

```
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',...
        'Value',num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job `ID` to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations – the number of workers specified in the `'Pool'` argument plus an additional head worker.

```
simJob = batchsim(in,'Pool',3)
```

```
                 ID: 1
               Type: pool
         NumWorkers: 4
           Username: #####
              State: running
     SubmitDateTime: ##-###-#### ##:##:##
      StartDateTime: 
   Running Duration: 0 days 0h 0m 0s
```

5. Use the `diary` method of the `Simulink.Simulation.Job` object to display the output of the batch job in the MATLAB command window.

Note that the diary is not displayed here because this is an example model.

```
diary(simJob)
```

Input Arguments

`simJob` — `Simulink.Simulation.Job` object

object

A `Simulink.Simulation.Job` object. To create a `simJob`, run `batchsim`.

Example: `simJob = batchsim(in,'Pool',4)`

`'filename'` — File to append with Command Window output text
string

Specify a file to append with Command Window output text from the `Simulink.Simulation.Job` object.

Example: `diary(simJob,'abc.txt')`

See Also

Functions
`batchsim` | `cancel` | `fetchOutputs` | `getSimulationJobs` | `listAutoAttachedFiles` | `parcluster` | `parsim` | `wait`

Classes
`Simulink.Simulation.Job` | `Simulink.SimulationInput` | `parallel.Cluster`

Topics

"Multiple Simulation Workflows"
"Batch Processing" (Parallel Computing Toolbox)

Introduced in R2018b

fetchOutputs

Package: Simulink

Retrieve an array of `Simulink.SimulationOutput` objects from all simulations in `Simulink.Simulation.Job`

Syntax

```
out = fetchOutputs(simJob)
```

Description

`out = fetchOutputs(simJob)` returns an array of `Simulink.SimulationOutput` objects containing the results of the simulations in a batch job, `simJob`.

`fetchOutputs` reports an error if the job is not in the `'finished'` state, or if one of its simulations encounters an error during execution. Use the `wait` method to wait for the job to complete before fetching outputs.

Examples

Fetch Outputs of the Batch Simulations

This example shows how to run parallel simulations in batch and fetch the resulting `Simulink.SimulationOutput` objects from the `Simulink.Simulation.Job` object. `batchsim` command offloads simulations to the compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

1. Open the model.

```
open_system('sldemo_househeat');
```

2. Define a set of values for different temperatures.

```
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

3. Using `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.

```
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',...
        'Value',num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job `ID` to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations – the number of workers specified in the `'Pool'` argument plus an additional head worker.

```
simJob = batchsim(in,'Pool',3)
```

```
                 ID: 1
               Type: pool
         NumWorkers: 4
           Username: #####
              State: running
     SubmitDateTime: ##-###-#### ##:##:##
      StartDateTime: 
   Running Duration: 0 days 0h 0m 0s
```

5. Access the results of the batch job using the `fetchOutputs` method. `fetchOutputs` returns an array of `Simulink.SimulationOuput` objects. You can fetch outputs only once `simJob` is in `finished` state.

```
out = fetchOutputs(simJob)
```

```
1x11 Simulink.SimulationOutput array
```

Input Arguments

`simJob` — `Simulink.Simulation.Job` object
object

A `Simulink.Simulation.Job` object. To create `simJob`, run `batchsim`.

Example: `simJob = batchsim(in,'Pool',4)`

Output Arguments

out — Simulation object containing logged simulation results
object

Array of `Simulink.SimulationOutput` objects that contain all of the logged simulation results. The size of the array is equal to the size of the array of `Simulink.SimulationInput` objects passed to `batchsim`.

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 are 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** tool can directly log signals.

See Also

Functions
`batchsim` | `cancel` | `diary` | `getSimulationJobs` | `listAutoAttachedFiles` | `parsim` | `wait`

Classes
`Simulink.Simulation.Job` | `Simulink.SimulationInput` | `Simulink.SimulationOutput`

Topics

"Multiple Simulation Workflows"
"Batch Processing" (Parallel Computing Toolbox)

Introduced in R2018b

listAutoAttachedFiles

Package: Simulink

List of files automatically attached to the `Simulink.Simulation.Job` object or parallel pool

Syntax

```
listAutoAttachedFiles(simJob)
```

Description

`listAutoAttachedFiles(simJob)` performs a dependency analysis on the `Simulink.Simulation.Job` job object, `simJob`. Then it displays a list of the code files that are already attached or are going to be automatically attached to the job object, `simJob`.

Examples

Run Parallel Simulations with `batchsim` and List Attached Files

This example shows how to run parallel simulations in batch and list any attached files. `batchsim` offloads simulations to the compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

1. Open the model.

```
open_system('sldemo_househeat');
```

2. Define a set of values for different temperatures.

```
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

3. Using the `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.

```
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',...
        'Value',num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job `ID` to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations – the number of workers specified in the `'Pool'` argument plus an additional head worker.

```
simJob = batchsim(in,'Pool',3)
```

```
                  ID: 1
                Type: pool
          NumWorkers: 4
            Username: #####
               State: running
      SubmitDateTime: ##-###-#### ##:##:##
       StartDateTime: 
    Running Duration: 0 days 0h 0m 0s
```

If `AutoAttachFiles` property of `Simulink.Simulation.Job` is set to true in the cluster profile, then the job running on the cluster has the necessary code files automatically attached to it. Use the `listAutoAttachedFiles` method to view the attached files.

Note that the list of attached files is not displayed here because this is an example model.

```
listAutoAttachedFiles(simJob)
```

Input Arguments

`simJob` — `Simulink.Simulation.Job` object

object

A `Simulink.Simulation.Job` object. To create a `simJob`, run `batchsim`.

Example: `simJob = batchsim(in,'Pool',4)`

See Also

Functions
`batchsim` | `cancel` | `diary` | `fetchOutputs` | `getSimulationJobs` | `parsim` | `wait`

Classes
`Simulink.Simulation.Job` | `Simulink.SimulationInput`

Topics

"Multiple Simulation Workflows"
"Batch Processing" (Parallel Computing Toolbox)

Introduced in R2018b

wait

Package: Simulink

Wait for Simulink.Simulation.Job object to change state

Syntax

```
wait(simJob)
wait(simJob,'stateOfJob')
wait(simJob,'stateOfJob',timeout)
```

Description

wait(simJob) blocks execution in a client session until the job identified by the object simJob reaches the 'finished' state or fails. This occurs when all the simulations finish execution on the workers.

wait(simJob,'stateOfJob') blocks execution in the client session until the specified job object changes state to the value of 'state'. The valid states to wait for are 'queued', 'running', and 'finished'. If the object is currently or has already been in the specified state, wait is not performed and execution returns immediately. For example, if you execute wait(simJob,'queued') for a job already in the 'finished' state, the call returns immediately.

wait(simJob,'stateOfJob',timeout) blocks execution until either the job reaches the specified 'state', or timeout seconds elapse, whichever happens first.

Examples

Wait for Simulations in simJob to Finish

This example shows uses the sldemo_househeat model to show how to wait for the batch simulations to finish.

1. Open the model.

```
open_system('sldemo_househeat');
```

2. Define a set of values for different temperatures.

```
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

3. Using the `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.

```
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',...
        'Value',num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job `ID` to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations – the number of workers specified in the `'Pool'` argument plus an additional head worker.

```
simJob = batchsim(in,'Pool',3)
```

```
                 ID: 1
               Type: pool
         NumWorkers: 4
           Username: #####
              State: running
     SubmitDateTime: ##-###-#### ##:##:##
      StartDateTime: 
   Running Duration: 0 days 0h 0m 0s
```

5. Now, wait for the job to finish before retrieving the outputs.

```
wait(simJob)
```

Input Arguments

`simJob` — `Simulink.Simulation.Job` object

object

A `Simulink.Simulation.Job` object. To create a `simJob`, run `batchsim`.

Example: `simJob = batchsim(in,'Pool','5')`

`'stateOfJob'` — Value of the simulation job object `State` property to wait for

'pending' | 'queued' | 'running' | 'finished' | 'failed' | 'unavailable'

Value of the `State` property of `Simulink.Simulation.Job` object to wait for.

Example: `wait(simJob,'queued')`

`timeout` — Maximum time to wait, in seconds

object

Specify a timeout for `wait` to block execution in seconds.

Example: `wait(simJob, 5)`

See Also

Functions

`batchsim` | `cancel` | `diary` | `fetchOutputs` | `listAutoAttachedFiles` | `parsim`

Classes

`Simulink.Simulation.Job` | `Simulink.SimulationInput`

Topics

"Multiple Simulation Workflows"

Introduced in R2018b

getSimulationJobs

Get all `Simulink.Simulation.Job` objects from cluster

Syntax

```
jobs = getSimulationJobs(myCluster)
```

Description

`jobs = getSimulationJobs(myCluster)` returns an array of `Simulink.Simulation.Job` objects that correspond to the jobs created by executing of `batchsim` on cluster, `myCluster`.

Examples

Get a `Simulink.Simulation.Job` Object from the Cluster

This example shows how to access a `batchsim` job that was submitted to a cluster. Assume that `myCluster` is a `parallel.Cluster` object on which the `Simulink.Simulation.Job` object is running.

```
jobs = getSimulationJobs(myCluster)
```

```
 1×2 Job array:

        ID           Type        State              FinishDateTime  Username
     -----------------------------------------------------------------------
   1     3           pool       queued                              #######
   2     4           pool       queued                              #######
```

The output displays all the `Simulink.Simulation.Job` objects on cluster, `myCluster`.

Input Arguments

`myCluster` — `parallel.Cluster` object
object

Cluster object representing parallel cluster compute resources.

Output Arguments

`jobs` — Array of `Simulink.Simulation.Job` objects
array

Array of `Simulink.Simulation.Job` objects submitted by executing `batchsim` command.

See Also

Functions
`batchsim` | `cancel` | `diary` | `fetchOutputs` | `listAutoAttachedFiles` | `parcluster` | `parsim` | `wait`

Classes
`Simulink.Simulation.Job` | `Simulink.SimulationInput` | `parallel.Cluster`

Topics

"Multiple Simulation Workflows"
"Batch Processing" (Parallel Computing Toolbox)
"Job Monitor" (Parallel Computing Toolbox)

Introduced in R2018b

Model and Block Parameters

Model Parameters

In this section...

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** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`

Parameter	Description	Values
`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'`
`ArtificialAlgebraic-LoopMsg`	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. Set by **Minimize algebraic loop** on the **Solver** section of the **Diagnostics** pane in the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`AssertControl`	Enable model verification blocks in the current model either globally or locally. Set by **Model Verification block enabling** on the Configuration Parameters dialog box.	`{'UseLocalSettings'} | 'EnableAll' | 'DisableAll'`

Parameter	Description	Values
`AutoInsertRateTranBlk`	Specify whether Simulink software inserts hidden Rate Transition blocks between blocks that have different sample rates. Set by **Automatically handle rate transition for data transfer** on the **Solver** pane of the Configuration Parameters dialog box.	`'on' | {'off'}`
`BlockDescription-StringDataTip`	Specifies whether to display the user description for a block as a data tip. In the Simulink Editor, set by **Description** on the **Display > Blocks > Block Tool Tip Options** menu.	`'on' | {'off'}`
`BlockNameDataTip`	Specifies whether to display the block name as a data tip. In the Simulink Editor, set by **Block Name** on the **Display > Blocks > Block Tool Tip Options** menu.	`'on' | {'off'}`
`BlockParametersDataTip`	Specifies whether to display a block parameter in a data tip. In the Simulink Editor, set by **Parameter Names & Values** on the **Display > Blocks > Block Tool Tip Options** menu.	`'on' | {'off'}`

Parameter	Description	Values
`BlockPriority-ViolationMsg`	Select the diagnostic action to take if Simulink software detects a block priority specification error. Set by **Block priority violation** on the **Solver** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	`{'warning'} | 'error'`
`BlockReduction`	Enables block reduction optimization. Set by **Block reduction** on the Configuration Parameters dialog box.	`{'on'} | 'off'`
`BlockReductionOpt`	See `BlockReduction` parameter for more information.	
`BooleanDataType`	Enable Boolean mode. Set by **Implement logic signals as Boolean data (vs. double)** on the Configuration Parameters dialog box.	`{'on'} | 'off'`
`BrowserLookUnderMasks`	Show masked subsystems in the Model Browser. In the Simulink Editor, set by **Include Systems with Mask Parameters** on the **View > Model Browser** menu.	`'on' | {'off'}`

Parameter	Description	Values
`BrowserShowLibraryLinks`	Show library links in the Model Browser. In the Simulink Editor, set by **Include Library Links** on the **View > Model Browser** menu.	`'on' | {'off'}`
`BufferReusableBoundary`	Insert buffers at reusable subsystem boundaries if needed.	`'on' | {'off'}`
`BufferReuse`	Enable reuse of block I/O buffers. Set by **“Reuse local block outputs” (Simulink Coder)** on the **Optimization** pane of the Configuration Parameters dialog box.	`{'on'} | 'off'`

`BusNameAdapt`	Repair broken selections in the Bus Selector and Bus Assignment block parameters dialog boxes that are due to upstream bus hierarchy changes.
	Set by **“Repair bus selections”** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box.
		`{'WarnAndRepair'} |`
		`'ErrorWithoutRepair'`

Parameter	Description	Values
BusObjectLabelMismatch	Select the diagnostic action to take if the name of a bus element does not match the name specified by the corresponding bus object. Set by **Element name mismatch** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'
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 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'

Parameter	Description	Values
`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'}`
`CheckSSInitialOutputMsg`	Enable checking for undefined initial subsystem output. Set by **Check undefined subsystem initial output** on the Configuration Parameters dialog box.	`{'on'} | 'off'`
`CloseFcn`	Set the close callback function, which can be a command or a variable. Set by **Model close function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	`{''}`
`CompiledBusType`	Return information about whether the signal connected to a port is not a bus, or whether it is a virtual or nonvirtual bus. (Read-only) Get with the `get_param` command. Specify a port or line handle. See “Display Information About Buses”.	Return values are `'NOT_BUS'`, `'VIRTUAL_BUS'`, and `'NON_VIRTUAL_BUS'`

Parameter	Description	Values
`CompiledModelBlockNormalModeVisibility`	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.	Return values indicate which Model blocks have normal mode visibility enabled.
`ConditionallyExecute-Inputs`	Enable conditional input branch execution optimization. Set by **Conditional input branch execution** on the Configuration Parameters dialog box.	`{'on'} | 'off'`
`ConsecutiveZCsStepRelTol`	Relative tolerance associated with the time difference between zero-crossing events. Set by **Time tolerance** on the **Solver** pane of the Configuration Parameters dialog box.	`{'10*128*eps'}`
`ConsistencyChecking`	Select the diagnostic action to take if S-functions have continuous sample times, but do not produce consistent results when executed multiple times. Set by **Solver data inconsistency** on the **Solver** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`

Parameter	Description	Values
`ContinueFcn`	Continue simulation callback. Set by **Simulation continue function** on the **Callbacks** pane of the Model Properties dialog box.	`{''}`
`CovCompData`	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. Set by the **Additional data to include in report** field in the **Advanced parameters** section of the **Coverage > Results** pane of the Configuration Parameters dialog box.	`{''}`

Parameter	Description	Values
CovCumulativeReport	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. If set to on, the Simulink Coverage™ software displays the coverage results from successive simulations in the report. If set to off, the software displays the coverage results for the last simulation in the report. Set by the **Include cumulative data in coverage report** option in the **Advanced parameters** section of the **Coverage > Results** pane of the Configuration Parameters dialog box.	'on' | {'off'}
CovDataFileName	If CovEnable is set to on, specifies the name of the file to which Simulink Coverage saves the coverage data results.	{''}

Parameter	Description	Values
`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'}`
`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'`

Parameter	Description	Values
`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	If CovHtmlReporting is set to on, use this parameter to select from a set of display options for the resulting model coverage report. Select these options in the **Results Explorer** settings after you record coverage for a model.	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): • '-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 • '-xEM=0' — Exclude execution metric details from report • '-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
		- '-xEM=0' — Filter Execution metric from report
CovIncludeTopModel	Option to include the top-level model in the coverage analysis.	{'on'} | 'off'
CovHtmlReporting	Set to on to tell the Simulink Coverage software to create an HTML report containing the coverage data at the end of simulation. Set by **Generate report automatically after analysis** on the **Coverage** > **Results** pane of the Configuration Parameters dialog box.	'on' | {'off'}
CovLogicBlockShortCircuit	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.	'on' | {'off'}

Parameter	Description	Values
`CovMcdcMode`	Determines the definition of Modified Condition Decision Coverage (MCDC) to use during coverage analysis. To record model coverage using the masking MCDC definition, set `CovMcdcMode` to `'Masking'`. To record model coverage using the unique-cause MCDC definition, set `CovMcdcMode` to `'UniqueCause'`.	`{'Masking'} | 'UniqueCause'`

Parameter	Description	Values
`CovMetricSettings`	Selects coverage metrics for a coverage report. 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. 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. Disable option `'e'` by selecting **Display coverage results using model coloring** in the **Results Explorer** settings after you record coverage for a model. **Note** The metrics and options set by this parameter can also be set by the following parameters: • CovHighlightResults • CovLogicBlockShortCircuit • CovMetricStructuralLevel • CovMetricLookupTable	`{'dwe'}` Each order-independent value enables a coverage metric or option as follows: • `'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
	• 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`	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`. Set by **Coverage for referenced models** on the **Coverage** pane of the Configuration Parameters dialog box.	`'on' | {'off'} | 'all' | 'filtered'`
`CovModelRefExcluded`	If `CovModelRefEnable` is set to `filtered`, this parameter stores a comma-separated list of referenced models for which coverage is disabled. Set by selecting **Coverage for referenced models** on the **Coverage** pane of the Configuration Parameters dialog box and then clicking **Select Models**.	`{''}`

Parameter	Description	Values
`CovNameIncrementing`	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.) 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.	`'on' | {'off'}`
`CovOutputDir`	If `CovEnable` is set to `on`, specifies the directory in which Simulink Coverage saves the coverage output files.	`{''}`
`CovPath`	Model path of the subsystem for which the Simulink Coverage software gathers and reports coverage data. Set by selecting **Subsystem** on the **Coverage** pane of the Configuration Parameters dialog box and then clicking **Select Subsystem**.	`{'/'}`

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	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. 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.	{'covdata'}

Parameter	Description	Values
CovSaveSingleTo-WorkspaceVar	If set to on, the Simulink Coverage software saves the results of the last simulation run in the workspace variable specified by CovSaveName. Set by **Save last run in workspace variable** on the **Coverage > Results** pane of the Configuration Parameters dialog box.	'on' | {'off'}
CovScope	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.	{'EntireSystem'} | 'ReferencedModels' | 'Subsystem'
CovSFcnEnable	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 Custom C/C++ Code in Simulink Models” (Simulink Coverage).	'on' | {'off'}
CovShowResultsExplorer	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.	{'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.	
CurrentSimState	Save the SimState snapshot of the simulation if the simulation is in a paused state.	Simulink.SimState.ModelSimState object | {[]}

Parameter	Description	Values
DataDictionary	Simulink data dictionary to which this model is linked. Set by **Data Dictionary** and **Base Workspace** on the **Data** pane of the Model Properties dialog box. For basic information about data dictionaries, see “What Is a Data Dictionary?”. To use this parameter programmatically, see “Store Data in Dictionary Programmatically”.	{''}
DataLoggingOverride	A Simulink.SimulationData.ModelLoggingInfo object that specifies the signal logging override settings for a model. See “Override Signal Logging Settings”.	Simulink.SimulationData.ModelLoggingInfo — {'OverrideSignals'} | 'LogAllAsSpecifiedInModel'
DatasetSignalFormat	Format for logged Dataset leaf elements. For details, see “Dataset signal format”.	'timetable' | {'timeseries'}
DataTransfer	A Simulink.GlobalDataTransfer object that configures data transfers for models configured for concurrent execution.	'on' | {'off'}

Parameter	Description	Values
`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'`
`Decimation`	Specify that Simulink software output only every `N` points, where `N` is the specified decimation factor. Set by **“Decimation”** on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`{'1'}`
`DefaultParameterBehavior`	Enable inlining of block parameters in generated code. Set by **Default parameter behavior**, see “Default parameter behavior” (Simulink Coder).	`'Inlined' | {'Tunable'}`

`DefaultAnnotationFontName`
`DefaultBlockFontName`
`DefaultLineFontName`
	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'`.
	Set with **Diagram > Format > Font Styles for Model** dialog box.
		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 FontSize property is set to -1. 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 FontAngle property is set to 'auto'. 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 FontWeight property is set to 'auto'. Set with **Diagram** > **Format** > **Font Styles for Model** dialog box.	{'normal'} | 'bold'

Parameter	Description	Values
DefaultUnderspecifiedDataType	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 **Math and Data Types** 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.	
`DisplayVectorAndFunctionCounts`	For internal use.	
`DisplayVectPropagationResults`	For internal use.	
`ExecutionContextIcon`	Show execution context bars on conditional subsystems that do not propagate execution context across the subsystem boundaries. In the Simulink Editor, set by **Execution Context Indicator** on the **Display** > **Signals & Ports** menu.	`'on' | {'off'}`
`ExplicitPartitioning`	Specifies whether or not to manually map tasks (explicit mapping) or use the rate-based tasks.	`'on' | {'off'}`
`ExpressionFolding`	Enables expression folding. Set by **“Eliminate superfluous local variables (Expression folding)” (Simulink Coder)** on the Configuration Parameters dialog box.	`{'on'} | 'off'`

Parameter	Description	Values
`ExternalInput`	Names of MATLAB workspace variables used to designate data and times to be loaded from the workspace. Set by the **Input** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`{'[t, u]'}`
`ExtMode...`	Parameters whose names start with `ExtMode` apply to Simulink external mode simulations. For more information, see “Host-Target Communication with External Mode Simulation” (Simulink Coder).	
`ExtrapolationOrder`	Extrapolation order of the `ode14x` implicit fixed-step solver. Set by **Extrapolation order** on the **Solver** pane of the Configuration Parameters dialog box.	integer — `1 | 2 | 3 | {4}`
`FastRestart`	Enable or disable fast restart mode. In the Simulink Editor toolbar, click the **Fast restart** button on or off.	`{'on'} | 'off'`

Parameter	Description	Values
`FcnCallInpInside-ContextMsg`	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. Set by **Context-dependent inputs** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	`{'Error'} | 'Warning'`
`FileName`	For internal use.	
`FinalStateName`	Names of final states to save to the workspace after a simulation ends. Set by the **Final states** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`{'xFinal'}`
`FixedStep`	Fixed-step size. Set by **Fixed-step size (fundamental sample time)** on the **Solver** pane of the Configuration Parameters dialog box.	`{'auto'}`

Parameter	Description	Values
FixptConstOverflowMsg	Specifies diagnostic action to take when a fixed-point constant overflow occurs during simulation. Set by **Detect overflow** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	{'none'} | 'warning' | 'error'
FixptConstPrecisionLossMsg	Specifies diagnostic action to take when a fixed-point constant precision loss occurs during simulation. Set by **Detect precision loss** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	{'none'} | 'warning' | 'error'
FixptConstUnderflowMsg	Specifies diagnostic action to take when a fixed-point constant underflow occurs during simulation. Set by **Detect underflow** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	{'none'} | 'warning' | 'error'
FixPtInfo	For internal use.	
FollowLinksWhen-OpeningFromGotoBlocks	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 (#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.	{'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`	Message behavior when the sample time is inherited. Set by **Source block specifies -1 sample time** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`InitFcn`	Function that is called when this model is first compiled for simulation. Set by **Model initialization function** on the **Callbacks** pane of the Model Properties dialog box. See "Create Model Callbacks" for more information.	`{''}`
`InitialState`	Initial state name or values. Set by the **Initial state** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	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'}`
`InitInArrayFormatMsg`	Message behavior when the initial state is an array. You set with the initial state with the **Initial state** configuration parameter. 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 `error`. Alternatively, you can set the message behavior using the **“InitInArrayFormatMsg”** on the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`

Parameter	Description	Values
`InsertRTBMode`	Control whether the Rate Transition block parameter **Ensure deterministic data transfer (maximum delay)** is set for auto-inserted Rate Transition blocks. Set by **Deterministic data transfer** on the **Solver** pane of the Configuration Parameters dialog box.	`'Always' | {'Whenever possible'} | 'Never (minimum delay)'`
`InspectSignalLogs`	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. Set by **“Record logged workspace data in Simulation Data Inspector”** on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`'on' | {'off'}`
`InstrumentedSignals`	Returns a `Simulink.HMI.InstrumentedSignals` 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.	object — `Simulink.HMI.InstrumentedSignals`

Parameter	Description	Values
`Int32ToFloatConvMsg`	Specify message behavior when a 32-bit integer is converted to a single-precision float. Set by **32-bit integer to single precision float conversion** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'}`
`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'`

Parameter	Description	Values
`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
`LibraryLinkDisplay`	Displays the blocks in the model that are linked or have disabled or modified links. In the Simulink Editor, set by **Library Links** on the **Display** menu.	`'none'| {'disabled'} |'user' | 'all'` Set to `none`, does not display the link badge on the block. Set to `disabled`, displays the disabled link badge on the block. Set to `user`, displays only links to the user libraries. Set to `all`, displays all links.
`LibraryType`	For internal use.	

Parameter	Description	Values
`LifeSpan`	Specify how long (in days) an application that contains blocks depending on elapsed or absolute time should be able to execute before timer overflow. Set by **Application lifespan (days)** on the **Math and Data Types** pane of the Configuration Parameters dialog box.	`{'auto'} |` any positive, nonzero scalar value
`LimitDataPoints`	Specify that the number of data points exported to the MATLAB workspace be limited to the number specified. Set by the **Limit data points** configuration parameter	`{'on'} | 'off'`
`LinearizationMsg`	For internal use.	
`Lines`	For internal use.	
`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.	

Parameter	Description	Values
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'}
LoggingToFile	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. Use this feature when logging large amounts of data that can cause memory issues. For more information about logging large data sets, see "Log Data to Persistent Storage".	'on' | {'off'}

Parameter	Description	Values
`MAModelExclusionFile`	Specifies the location of the Model Advisor exclusion file. Set by the **File Name** field on the Model Advisor Exclusion Editor dialog box.	`{' '}`
`MaskedZcDiagnostic`	Control diagnostic messages related to zero-crossings that are being masked.	`'none' | {'warning'} | 'error'`
`MaxConsecutiveMinStep`	Maximum number of minimum step size violations allowed during simulation. This option appears when the solver type is `Variable-step` and the solver is an ode one. Set by **Number of consecutive min steps** on the **Solver** pane of the Configuration Parameters dialog box.	`{'1'}`
`MaxConsecutiveZCs`	Maximum number of consecutive zero crossings allowed during simulation. This option appears when the solver type is `Variable-step` and the solver is an ode one. Set by **Number of consecutive zero crossings** on the **Solver** pane of the Configuration Parameters dialog box.	`{'1000'}`

Parameter	Description	Values
MaxConsecutiveZCsMsg	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 Variable-step and the solver is an ode one. Set by **Consecutive zero crossings violation** on the **Solver** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	'none' | 'warning' | {'error'}
MaxDataPoints	Maximum number of output data points to save. Set by the **Limit data points to last** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	{'1000'}
MaxMDLFileLineLength	Controls the line lengths in the model file. Use this to avoid line-wrapping, which can be important for source control tools. Specifies the maximum length in bytes, which may different from the number of characters in Japanese, and is different from the number of columns when tabs are present.	integer — -1 (unlimited) or >= 80. Default is 120.

Parameter	Description	Values
`MaxNumMinSteps`	Maximum number of times the solver uses the minimum step size.	`{'-1'}`
`MaxOrder`	Maximum order for `ode15s`. Set by **Maximum order** on the **Solver** pane of the Configuration Parameters dialog box.	`'1' | '2' | '3' | '4' | {'5'}`
`MaxStep`	Maximum step size. Set by **Max step size** on the **Solver** pane of the Configuration Parameters dialog box.	`{'auto'}`
`MdlSubVersion`	For internal use	
`MergeDetectMultiDriving-BlocksExec`	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. Set by **Detect multiple driving blocks executing at the same time step** on the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`
`Metadata`	Names and attributes of arbitrary data associated with the model. To extract this metadata structure without needing to load the model, use the method `Simulink.MDLInfo.getMetadata`.	Structure. Fields can be character vectors, numeric matrices of type "double", or more structures.

Parameter	Description	Values
MinMaxOverflow-ArchiveData	For internal use	
MinMaxOverflow-ArchiveMode	Logging type for fixed-point logging. Set by **Overwrite or merge model simulation results** in the Fixed-Point Tool.	{'Overwrite'} | 'Merge'
MinMaxOverflowLogging	Setting for fixed-point logging. Set by **Fixed-point instrumentation mode** in the Fixed-Point Tool.	{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'
MinStep	Minimum step size for the solver. Set by **Min step size** on the **Solver** pane of the Configuration Parameters dialog box.	{'auto'}
MinStepSizeMsg	Message shown when minimum step size is violated. Set by **Min step size violation** on the **Solver** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	{'warning'} | 'error'

Parameter	Description	Values
`ModelBlockNormalModeVisibility`	Use with `set_param` to set normal mode visibility on for the specified Model blocks. You can set this parameter with the Model Block Normal Mode Visibility dialog box. For details, see “Specify the Instance Having Normal Mode Visibility”.	With `set_param`, use an array of `Simulink.BlockPath` objects or cell array of cell arrays of character vectors of paths to blocks or models. With `set_param`, an empty array specifies to use the Simulink default selection for the instance to have normal mode visibility enabled.
`ModelBlockNormaModeVisibilityBlockPath`	Return information about which Model blocks have normal mode visibility enabled. Use with a model that you are editing.	Return values indicate which Model blocks have normal mode visibility enabled. See “Simulate Multiple Referenced Model Instances in Normal Mode”.
`ModelBrowserVisibility`	Show the Model Browser. In the Simulink Editor, set by **Model Browser** on the **View** menu.	`'on' | {'off'}`
`ModelBrowserWidth`	Width of the Model Browser pane in the model window. To display the Model Browser pane, see the `ModelBrowserVisibility` 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`	This parameter is maintained for compatibility purposes only. Do not use this parameter. You can use the Model Advisor to identify models referenced in Accelerator mode for which Simulink ignores certain configuration parameters. **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. For more information, see "Diagnostic Configuration Parameters Ignored in Accelerator Mode".	`{'none'} | 'warning' | 'error'` Simulink ignores this parameter if you set it to `warning` or `error`.
`ModelReferenceData-LoggingMessage`	Message shown when there is unsupported data logging. Set by **Unsupported data logging** on the **Model Referencing Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`

Parameter	Description	Values
`ModelReferenceExtra-NoncontSigs`	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. Set by **Extraneous discrete derivative signals** on the **Solver** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | 'warning' | {'error'}`
`ModelReferenceIO-MismatchMessage`	Message shown when there is a port and parameter mismatch. Set by **Port and parameter mismatch** on the **Model Referencing Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`
`ModelReferenceIOMsg`	Message shown when there is an invalid root Inport or Outport block connection. Set by **Invalid root Inport/ Outport block connection** on the **Model Referencing Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`

Parameter	Description	Values
`ModelReferenceMin-AlgLoopOccurrences`	Toggles the minimization of algebraic loop occurrences. Set by **Minimize algebraic loop occurrences** on the **Model Referencing** pane of the Configuration Parameters dialog box.	`'on' | {'off'}`
`ModelReferenceNum-InstancesAllowed`	Total number of model reference instances allowed per top model. Set by **Total number of instances allowed per top model** on the **Model Referencing** pane of the Configuration Parameters dialog box.	`'Zero' | 'Single' | {'Multi'}`
`ModelReferencePass-RootInputsByReference`	Toggles the passing of scalar root inputs by value. Set by “Pass fixed-size scalar root inputs by value for code generation”on the **Model Referencing** pane of the Configuration Parameters dialog box.	`{'on'} | 'off'`
`ModelReferenceSim-TargetVerbose`	This parameter is deprecated and has no effect. Use `AccelVerboseBuild` 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.	
`ModelReferenceVersion-MismatchMessage`	Message shown when there is a model block version mismatch. Set by **Model block version mismatch** on the **Model Referencing Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`
`ModelVersion`	Version number of model.	`{'1.1'}`
`ModelVersionFormat`	Format of model's version number. Set by **Model version** on the **History** pane of the Model Properties dialog box. See “Model Information and History” for more information.	`{'1.%<AutoIncrement: 0>'}`
`ModelWorkspace`	References this model's model workspace object.	an instance of the `Simulink.ModelWorkspace` class
`ModifiedByFormat`	Format for the display of last modifier. Set by **Last saved by** on the **History** pane of the Model Properties dialog box. See “Model Information and History” for more information. Can also be set by **Last saved by** on the **Model history** field on the **History** pane of the Model Explorer.	`{'%<Auto>'}`

Parameter	Description	Values
ModifiedComment	Field for user comments.	{''}
ModifiedDateFormat	Format used to generate the value of the LastModifiedDate parameter. Set by **Last saved on** on the **History** pane of the Model Properties dialog box. See "Model Information and History" for more information.	{'%<Auto>'}
ModifiedHistory	Area for keeping notes about the history of the model. Set by the **Model history** field on the **History** pane of the Model Properties dialog box. See "Model Information and History" for more information. Can also be set by the **Model history** field on the **History** pane of the Model Explorer.	{''}
MultiTaskCondExecSysMsg	Select the diagnostic action to take if Simulink software detects a subsystem that might cause data corruption or nondeterministic behavior. Set by **Multitask conditionally executed subsystem** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	'none' | 'warning' | {'error'}

Parameter	Description	Values
`MultiTaskDSMMsg`	Specifies diagnostic action to take when one task reads data from a Data Store Memory block to which another task writes data. Set by **Multitask data store** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | 'warning' | {'error'}`
`MultiTaskRateTransMsg`	Specifies diagnostic action to take when an invalid rate transition takes place between two blocks operating in multitasking mode. Set by **Multitask rate transition** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	`'warning' | {'error'}`
`Name`	Model name.	character vector
`NonBusSignalsTreatedAsBus`	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. **“Non-bus signals treated as bus signals”** on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`

Parameter	Description	Values
`NumberNewtonIterations`	Number of Newton's method iterations performed by the `ode14x` implicit fixed-step solver. Set by **Number Newton's iterations** on the **Solver** pane of the Configuration Parameters dialog box.	integer — `{1}`
`NumStatesForStiffnessChecking`	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`. For more information, see “Use Auto Solver to Select a Solver”.	`{''}`
`ObjectParameters`	Names and attributes of model parameters.	structure
`Open`	For internal use.	
`OptimizeBlockIOStorage`	Enables signal storage reuse optimization. Set by **Signal storage reuse** on the Configuration Parameters dialog box.	`{'on'} | 'off'`

Parameter	Description	Values
`OutputOption`	Time step output options for variable-step solvers. Set by **Output options** parameter under **Configuration Parameters** > **Data Import/Export** > **Additional parameters**.	`'AdditionalOutputTimes' | {'RefineOutputTimes'} | 'SpecifiedOutputTimes'`
`OutputSaveName`	Workspace variable to store the model outputs. Set by the **Output** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`{'yout'}`
`OutputTimes`	Output times set when Set by **Output options** parameter under **Configuration Parameters** > **Data Import/Export** > **Additional parameters** is set to `Produce additional output`. Set using the **Output times** parameter.	`{'[]'}` **Note** If the value of **Output options** is `Produce additional output` or `Produce specified output only`, set to a value other than the default value of `'[]'`.
`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	Paper position mode. • auto 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. • manual 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. • tiled Enables tiled printing. See “Tiled Printing” for more information.	{'auto'} | 'manual' | 'tiled'
PaperSize	Size of PaperType in PaperUnits.	vector — [width height] (read only)

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`	Specify if you want the Simulink software to perform a consistency check on the parallel pool before starting a parallel build. If you set the parameter to `on`, the client and the remote workers must meet the following criteria for the parallel build to initiate: • The parallel pool is open. • The pool is `spmd` 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. If you set the parameter to `off`, the software displays a warning for the first condition that fails and then performs a sequential build.	`{'on'} | 'off'`

Parameter	Description	Values
`ParameterArgumentNames`	List of parameters used as arguments when this model is called as a reference. Set by checking the **Argument** column for variables in the model workspace of the referenced model. See “Parameterize Instances of a Reusable Referenced Model”.	`{''}`
`ParameterDowncastMsg`	Specifies diagnostic action to take when a parameter downcast occurs during simulation. Set by **Detect downcast** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | 'warning' | {'error'}`
`ParameterOverflowMsg`	Specifies diagnostic action to take when a parameter overflow occurs during simulation. Set by **Detect overflow** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | 'warning' | {'error'}`

Parameter	Description	Values
`ParameterPrecisionLossMsg`	Specifies diagnostic action to take when parameter precision loss occurs during simulation. Set by **Detect precision loss** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`ParameterTunabilityLossMsg`	Specifies diagnostic action to take when a parameter cannot be tuned because it uses unsupported functions or operators. Set by **Detect loss of tunability** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`ParameterUnderflowMsg`	Specifies diagnostic action to take when a parameter underflow occurs during simulation. Set by **Detect underflow** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`{'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`	Pause simulation callback. Set by **Simulation pause function** on the **Callbacks** pane of the Model Properties dialog box.	`{''}`
`PortDataTypeDisplayFormat`	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. In the Simulink Editor, set by **Display** > **Signals and Ports** > **Port Data Type Display Format**.	`{'AliasTypeOnly'} | 'BaseTypeOnly' | 'BaseAndAliasTypes'`
`PositivePriorityOrder`	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. Set by **Higher priority value indicates higher task priority** on the **Solver** pane of the Configuration Parameters dialog box.	`'on' | {'off'}`

Parameter	Description	Values
`PostLoadFcn`	Function invoked just after this model is loaded. Set by **Model post-load function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	`{''}`
`PostSaveFcn`	Function invoked just after this model is saved to disk. Not executed for blocks inside library links. Set by **Model post-save function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	`{''}`
`PreLoadFcn`	Preload callback. Set by **Model pre-load function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	`{''}`

Parameter	Description	Values
PreSaveFcn	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 save_system(A, B, 'BreakUserLinks', 'on'). Set by **Model pre-save function** on the **Callbacks** pane of the Model Properties dialog box. See "Create Model Callbacks" for more information.	{''}
ProdBitPerChar	Describes the length in bits of the C char data type supported by the hardware board to be used by this model. Set by **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — {8}
ProdBitPerInt	Describes the length in bits of the C int data type supported by the hardware board to be used by this model. Set by **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — {32}

Parameter	Description	Values
ProdBitPerLong	Describes the length in bits of the C long data type supported by the hardware board to be used by this model. Set by **Number of bits: long** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — {32}
ProdBitPerLongLong	Describes the length in bits of the C long data type supported by the hardware board to be used by this model. Set by **Number of bits: long long** on the **Hardware Implementation** pane of the Configuration Parameters dialog box. The value of this parameter must be greater than or equal to the value of ProdBitPerLong.	integer — {64}
ProdBitPerShort	Describes the length in bits of the C short data type supported by the hardware board to be used by this model. Set by **Number of bits: short** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — {16}

Parameter	Description	Values
`ProdEndianess`	Describes the significance of the first byte of a data word of the hardware board to be used by this model. Set by **Byte ordering** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	`{'Unspecified'} | 'LittleEndian' | 'BigEndian'`
`ProdEqTarget`	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.	`{'on'} | 'off'`
`ProdHWDeviceType`	Predefined hardware device to specify the C language constraints for your microprocessor. Set by **Device vendor** and **Device type** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	`{'Generic->Unspecified (assume 32-bit Generic)'}`
`ProdIntDivRoundTo`	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. Set by **Signed integer division rounds to** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	`'Floor' | 'Zero' | {'Undefined'}`

Parameter	Description	Values
ProdLargestAtomicFloat	Specify the largest floating-point data type that can be atomically loaded and stored on the hardware board. Set by **Largest atomic size: floating-point** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	'Float' | 'Double' | {'None'}
ProdLargestAtomicInteger	Specify the largest integer data type that can be atomically loaded and stored on the hardware board. Set this parameter to 'LongLong' only if the production hardware supports the C long long data type and you have set ProdLongLongMode to 'on'. Set by **Largest atomic size: integer** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	{'Char'} | 'Short' | 'Int' | 'Long' | 'LongLong'
ProdLongLongMode	Specify that your C compiler supports the C long long data type. Most C99 compilers support long long. Set by **Support long long** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	'on' | {'off'}

Parameter	Description	Values
`ProdShiftRightIntArith`	Describes whether the C compiler that creates production code for this model implements a signed integer right shift as an arithmetic right shift. Set by **Shift right on a signed integer as arithmetic shift** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	`{'on'} | 'off'`
`ProdWordSize`	Describes the word length in bits of the hardware board to be used by this model. Set by **Number of bits: native** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`
`Profile`	Enables the simulation profiler for this model. In the Simulink Editor, set by **Show Profiler Report** on the **Analysis** menu.	`'on' | {'off'}`

Parameter	Description	Values
PropagateSignalLabelsOutOfModel	Pass propagated signal names to output signals of Model block. Set by **Propagate all signal labels out of the model** on the **Model Referencing** pane of the Configuration Parameters dialog box. See "Propagate all signal labels out of the model" for more information.	{'on'} | 'off'
PropagateVarSize	Select how variable-size signals propagate through referenced models. Set by **Propagate sizes of variable-size signals** on the **Model Referencing** pane of the Configuration Parameters dialog box. See **"Model Configuration Parameters: Model Referencing"** for more information.	| 'Infer from blocks in model' | 'Only when enabling' | 'During execution'
ReadBeforeWriteMsg	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. Set by **Detect read before write** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	{'UseLocalSettings'} | 'DisableAll' | 'EnableAllAsWarning' | 'EnableAllAsError'

Parameter	Description	Values
RecordCoverage	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: CovCompData CovCumulativeReport CovCumulativeVarName CovHTMLOptions CovHtmlReporting CovMetricSettings CovModelRefEnable CovModelRefExcluded CovNameIncrementing CovPath CovReportOnPause CovSaveCumulativeToWork-SpaceVar CovSaveName CovSaveSingleToWorkspac eVar If set to off, no model coverage data is collected or reported.	'on' | {'off'}

Parameter	Description	Values
	Set by **Entire System** on the **Coverage** pane of the Configuration Parameters dialog box.	
`Refine`	Refine factor. Set by **Refine factor** parameter under parameter under **Configuration Parameters > Data Import/ Export > Additional parameters**.	`{'1'}`
`RelTol`	Relative error tolerance. Set by **Relative tolerance** on the **Solver** pane of the Configuration Parameters dialog box.	`{'1e-3'}`
`RemoveDisableFunc`	For model referencing contexts for ERT targets, remove the generated disable functions that cannot be reached from anywhere in the generated code. Set by the **“Remove disable function” (Embedded Coder)** configuration parameter.	`'on' | {'off'}`

Parameter	Description	Values
`RemoveResetFunc`	For model referencing contexts for ERT targets, remove the generated reset functions that cannot be reached from anywhere in the generated code. Set by the **“Remove reset function” (Embedded Coder)** configuration parameter.	`{'on'} | 'off'`
`ReportName`	Name of the associated file for the Report Generator.	`{'simulink-default.rpt'}`
`ReqHilite`	Highlights all the blocks in the Simulink diagram that have requirements associated with them. In the Simulink Editor, set by **Highlight Model** on the **Analysis > Requirements** menu.	`'on' | {'off'}`
`RequirementInfo`	For internal use.	
`RootOutportRequire-BusObject`	Specifies diagnostic action to take when a bus enters a root model Outport block for which a bus object has not been specified. Set by **Unspecified bus object at root Outport block** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`

Parameter	Description	Values
RTPrefix	Specifies diagnostic action to take when Simulink software encounters an object name that begins with rt. Set by **"rt" prefix for identifiers** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	'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	This option appears when the solver type is Fixed-step. Set by **Periodic sample time constraint** on the **Solver** pane of the Configuration Parameters dialog box.	{'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 `slCharacterEncoding` 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`	Save simulation output to workspace. Set by the **Output** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box. Do not use a variable name that is the same as a `Simulink.SimulationOutput` object function name or property name.	`{'on'} | 'off'`
`SaveState`	Save states to workspace. Set by the **States** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`'on' | {'off'}`
`SaveTime`	Save simulation time to workspace. Set by the **Time** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box. Do not use a variable name that is the same as a `Simulink.SimulationOutput` object function name or property name.	`{'on'} | 'off'`

Parameter	Description	Values
`SaveWithDisabledLinksMsg`	Specifies diagnostic action to take when saving a block diagram having disabled library links. Set by **Block diagram contains disabled library links** on the **Saving Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`SaveWithParameterized-LinksMsg`	Specifies diagnostic action to take when saving a block diagram having parameterized library links. Set by **Block diagram contains parameterized library links** on the **Saving Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`ScreenColor`	Background color of the model window. In the Simulink Editor, set by **Canvas Color** on the **Diagram > Format** menu.	`'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	Specifies diagnostic action to take when S-function upgrades are needed. Set by **S-function upgrades needed** on the **Compatibility Diagnostics** pane of the Configuration Parameters dialog box.	{'none'} | 'warning' | 'error'
SFExecutionAtInitializationDiag	Select the diagnostic action to take when Stateflow detects triggered or enabled charts that are not running at initialization. Set by **“Execute-at-Initialization disabled in presence of input events”** on the **Compatibility Diagnostics** pane of the Configuration Parameters dialog box.	

Parameter	Description	Values
SFInvalidInputDataAccess-InChartInitDiag	Select the diagnostic action to take when a chart: - Has the ExecuteAtInitialization property set to true - Accesses input data on a default transition or associated state entry actions, which execute at chart initialization Set by **Invalid input data access in chart initialization** on the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'
SFMachineParented-DataDiag	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. Set by **“Use of machine-parented data instead of Data Store Memory”** on the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box.	'none'|{'warning'}|'error'

Parameter	Description	Values
SFNoUnconditionalDefault-TransitionDiag	Select the diagnostic action to take when a chart does not have an unconditional default transition to a state or a junction. Set by **No unconditional default transitions** on the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'
SFSelfTransitionDiag	Select the diagnostic action to take when you can remove a self-transition on a leaf state. Set by **“Self transition on leaf state”** on the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box.	'none'|{'warning'}|'error'
SFSimEcho	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. Set by **Echo expressions without semicolons** on the **Simulation Target** pane of the Configuration Parameters dialog box.	{'on'} | 'off'

Parameter	Description	Values
SFTemporalDelaySmallerThanSampleTimeDiag	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. Set by **“Absolute time temporal value shorter than sampling period”** on the **Simulation Target** pane of the Configuration Parameters dialog box.	'none'|{'warning'}|'error'
SFTransitionActionBeforeConditionDiag	Select the diagnostic action to take when a transition action is specified before a condition action in a transition path containing multiple segmented transitions. Set by **“Transition action specified before condition action”** on the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'
SFTransitionOutsideNaturalParentDiag	Select the diagnostic action to take when a chart contains a transition that loops outside the parent state or junction. Set by **Transition outside natural parent** on the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'

Parameter	Description	Values
SFUndirectedBroadcastEventsDiag	Select the diagnostic action to take when a chart contains undirected local event broadcasts. Set by **Undirected event broadcasts** on the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'
SFUnexpectedBacktrackingDiag	Select the diagnostic action to take when a chart junction: • Does not have an unconditional transition path to a state or a terminal junction • Has multiple transition paths leading to it Set by **Unexpected backtracking** on the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'
SFUnreachableExecutionPathDiag	Select the diagnostic action to take when there are chart constructs not on a valid execution path. Set by **“Unreachable execution path”** on the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'

Parameter	Description	Values
`SFUnusedDataAndEventsDiag`	Select the diagnostic action to take for detection of unused data and events in a chart. Set by **Unused data and events** on the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`ShapePreserveControl`	At each time step, use derivative information to improve integration accuracy. Set by **Shape preservation** on the **Solver** pane of the Configuration Parameters dialog box.	`'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`	Show signal dimensions on this model's block diagram. In the Simulink Editor, set by **Signal Dimensions** on the **Display > Signal & Ports** menu.	`'on' | {'off'}`
`ShowLineDimensions-OnError`	For internal use.	
`ShowLineWidths`	Deprecated. Use `ShowLineDimensions` instead.	
`ShowLoopsOnError`	Highlight invalid loops graphically.	`{'on'} | 'off'`

Parameter	Description	Values
`ShowModelReference-BlockIO`	Toggles display of I/O mismatch on block. In the Simulink Editor, set by **Block I/O Mismatch for Referenced Model** on the **Display** > **Blocks** menu.	`'on' | {'off'}`
`ShowModelReference-BlockVersion`	Toggles display of version on block. In the Simulink Editor, set by **Block Version for Referenced Models** on the **Display** > **Blocks** menu.	`'on' | {'off'}`
`Shown`	For internal use.	
`ShowPageBoundaries`	Toggles display of page boundaries on the Simulink Editor canvas. In the Simulink Editor, set by **Show Page Boundaries** on the **File** > **Print** menu.	`'on' | {'off'}`
`ShowPortDataTypes`	Show data types of ports on this model's block diagram. In the Simulink Editor, set by **Port Data Types** on the **Display** > **Signals & Ports** menu.	`'on' | {'off'}`
`ShowPortDataTypesOnError`	For internal use.	

Parameter	Description	Values
`ShowPortUnits`	Show units of ports, subsystem, and model block icons on the model block diagram. In the Simulink Editor, set **Port Units** on the **Display** > **Signals & Ports** menu.	`'on' | {'off'}`
`ShowStorageClass`	Show storage classes of signals on this model's block diagram. In the Simulink Editor, set by **Storage Class** on the **Format** > **Signals & Ports** menu.	`'on' | {'off'}`
`ShowTestPointIcons`	Show test point icons on this model's block diagram. In the Simulink Editor, set by **Testpoint & Logging Indicators** on the **Display** > **Signals & Ports** menu.	`{'on'} | 'off'`
`ShowViewerIcons`	Show viewer icons on this model's block diagram. In the Simulink Editor, set by **Viewer Indicator** on the **Display** > **Signals & Ports** menu.	`{'on'} | 'off'`

Parameter	Description	Values
`SignalHierarchy`	If the signal is a bus, returns the name and hierarchy of the signals in the bus. (Read-only) Get with the `get_param` command. Specify a port or line handle. See "Display Information About Buses".	Return values reflect the structure of the signal that you specify.
`SignalInfNanChecking`	Specifies diagnostic action to take when the value of a block output is `Inf` or `NaN` at the current time step. Set by **Inf or NaN block output** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`
`SignalLabelMismatchMsg`	Specifies diagnostic action to take when a signal label mismatch occurs. Set by **Signal label mismatch** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`
`SignalLogging`	Globally enable signal logging for this model. Set by the **Signal logging** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`{'on'} | 'off'`

Parameter	Description	Values
`SignalLoggingName`	Name for saving signal logging data to a workspace. Set by the **Signal logging** field on the **Data Import/Export** pane of the Configuration Parameters dialog box. Do not use a variable name that is the same as a `Simulink.SimulationOutput` object function name or property name.	`{'logsout'}`
`SignalLoggingSaveFormat`	Format for saving signal logging data.	`{'Dataset'}` `'ModelDataLogs'` is supported for backward compatibility. However, when you open a model in R2016a or later, signal logging uses `Dataset` format, regardless of the setting of this parameter.
`SignalNameFromLabel`	Propagate signal names for Bus Creator block input signals whenever you change the name of an input signal programmatically. Set with the `set_param` command, using either a port or line handle and a character vector specifying the signal name to propagate.	`{''}`

Parameter	Description	Values
SignalRangeChecking	Select the diagnostic action to take when signals exceed specified minimum or maximum values. Set by **Simulation range checking** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	{'none'} | 'warning' | 'error'
SignalResolutionControl	Control which named states and signals get resolved to Simulink signal objects. Set by **Signal resolution** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	'None' | {'UseLocalSettings'} | 'TryResolveAll' | 'TryResolveAllWithWarning'
SigSpecEnsureSample-TimeMsg	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. Set by **Enforce sample times specified by Signal Specification blocks** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	'none' | {'warning'} | 'error'

Parameter	Description	Values
SimBuildMode	Specifies how you build the simulation target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by **Simulation target build mode** on the **Simulation Target** pane of the Configuration Parameters dialog box.	{'sf_incremental_build'} | 'sf_nonincremental_build' | 'sf_make' | 'sf_make_clean' | 'sf_make_clean_objects'
SimCompilerOptimization	Specifies the compiler optimization level during acceleration code generation. Set by **Compiler optimization level** on the Configuration Parameters dialog box.	'on' | {'off'}
SimCtrlC	Enables responsiveness checks in code generated for MATLAB Function blocks. Set by **“Ensure responsiveness”** on the **Simulation Target** pane of the Configuration Parameters dialog box.	{'on'} | 'off'

Parameter	Description	Values
`SimCustomHeaderCode`	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. Set by **Header file** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	`{''}`
`SimCustomInitializer`	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. Set by **Initialize function** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	`{''}`
`SimCustomSourceCode`	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. Set by **Source file** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	`{''}`

Parameter	Description	Values
`SimCustomTerminator`	Enter code statements that execute at the end of simulation for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by **Terminate function** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	`{''}`
`SimIntegrity`	Detects violations of memory integrity while building MATLAB Function blocks and stops simulation with a diagnostic. Set by **"Ensure memory integrity"** on the **Simulation Target** pane of the Configuration Parameters dialog box.	`{'on'} | 'off'`
`SimParseCustomCode`	Specify whether or not to parse the custom code and report unresolved symbols in the model. Set by **Parse custom code symbols** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	`{'on'} | 'off'`

Parameter	Description	Values
`SimReservedNameArray`	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. Set by **Reserved names** on the **Simulation Target > Symbols** pane of the Configuration Parameters dialog box.	character vector array — `{{}}`
`SimulationCommand`	Executes a simulation command. **Note** You cannot use `set_param` to run a simulation in a MATLAB session that does not have a display, i.e., if you used `matlab -nodisplay` to start the session.	`'start' | 'stop' | 'pause' | 'continue' | 'step' | 'update' | 'WriteDataLogs' | 'SimParamDialog' | 'connect' | 'disconnect' | 'WriteExtModeParamVect' | 'AccelBuild'`
`SimulationMode`	Indicates whether Simulink software should run in Normal, Accelerator, Rapid Accelerator, SIL, PIL, or External mode. In the Simulink Editor, set by the **Simulation > Mode** menu.	`{'normal'} | 'accelerator' | 'rapid-accelerator' | 'external' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'`

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`	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.
	Set by **Include directories** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.
		`{''}`
		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,
		`'C:\Project "C:\Custom Files"'`

`SimUserLibraries`	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. Set by **Libraries** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	`{''}`
`SimUserSources`	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. Set by **Source files** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	`{''}`

Parameter	Description	Values
`SingleTaskRateTransMsg`	Specifies diagnostic action to take when a rate transition takes place between two blocks operating in single-tasking mode. Set by **Single task rate transition** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`
`Solver`	Solver used for the simulation. Set by the **Solver** drop-down list on the **Solver** pane of the Configuration Parameters dialog box.	`'VariableStepDiscrete' | {'ode45'} | 'ode23' | 'ode113' | 'ode15s' | 'ode23s' | 'ode23t' | 'ode23tb' | 'FixedStepDiscrete' |'ode8'| 'ode5' | 'ode4' | 'ode3' | 'ode2' | 'ode1' | 'ode14x'`
`EnableMultiTasking`	Solver mode for this model. This option appears when the solver type is `Fixed-step`. Set by **“Treat each discrete rate as a separate task”** on the **Solver** pane of the Configuration Parameters dialog box.	`'On' | {'Off'}`
`SolverName`	Solver used for the simulation. See `Solver` parameter for more information.	

Parameter	Description	Values
`SolverPrmCheckMsg`	Enables diagnostics to control when Simulink software automatically selects solver parameters. This option notifies you if: • 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** Set by **Automatic solver parameter selection** on the **Solver** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`
`SolverResetMethod`	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)`. Set by **Solver reset method** on the **Solver** pane of the Configuration Parameters dialog box.	`{'Fast'} | 'Robust'`

Parameter	Description	Values
`SolverType`	Solver type used for the simulation. Set by **Type** on the **Solver** pane of the Configuration Parameters dialog box.	`{'Variable-step'} | 'Fixed-step'`
`SortedOrder`	Show the sorted order of this model's blocks on the block diagram. In the Simulink Editor, set by **Sorted Execution Order** on the **Display** > **Blocks** menu.	`'on' | {'off'}`
`StartFcn`	Start simulation callback. Set by **Simulation start function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	`{''}`
`StartTime`	Simulation start time. Set by **Start time** on the **Solver** pane of the Configuration Parameters dialog box.	`{'0.0'}`

Parameter	Description	Values
`StateNameClashWarn`	Select the diagnostic action to take when a name is used for more than one state in the model. Set by **State name clash** on the **Solver** section of the **Diagnostics** pane of the Configuration Parameters dialog box.	`'none' | {'warning'}`
`StateSaveName`	State output name to be saved to workspace. Set by the **States** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	`{'xout'}`
`StatusBar`	Has no effect in Simulink Editor. This parameter will be removed in a future release. In the Simulink Editor, set by **Status Bar** on the **View** menu.	`{'on'} | 'off'`
`StiffnessThreshold`	Threshold value to determine if the model is stiff. A model is stiff if the stiffness exceeds the `StiffnessThreshold` value. The default value for this parameter is 1000. For more information, see “Use Auto Solver to Select a Solver”.	`{''}`

Parameter	Description	Values
`StopFcn`	Stop simulation callback. Set by **Simulation stop function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	`{''}`
`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'`

Parameter	Description	Values
`SupportModelReferenceSimTargetCustomCode`	Use custom C code with Stateflow or with MATLAB Function blocks during referenced model simulation (SIM) target build for accelerator mode **Caution** Using custom C code for referenced models in accelerator mode can produce different results than if you simulate the model without using the custom code. If the custom code includes declarations of structures for buses or enumerations, the SIM target generation fails if the build results in duplicate declarations of those structures. Also, if the custom code uses a structure representing a bus or enumeration, you could get unexpected simulation results.	`'on' | {'off'}`
`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 `long` data type supported by the hardware used to test generated code.	integer — `{32}`
`TargetBitPerLongLong`	Describes the length in bits of the C `long long` 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 `TargetBitPerLong`.	integer — `{64}`
`TargetBitPerShort`	Describes the length in bits of the C `short` data type supported by the hardware used to test generated code.	integer — `{16}`
`TargetEndianess`	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'
TargetTypeEmulation WarnSuppressLevel	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`	Specifies diagnostic action to take when tasks have equal priority. Set by **Tasks with equal priority** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	`'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`	Specifies diagnostic action to take if Simulink software makes a minor adjustment to a sample hit time while running the model. Set by **Sample hit time adjusting** on the Configuration Parameters dialog box.	`{'none'} | 'warning'`
`TimeSaveName`	Simulation time name. Set by the **Time** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	variable — `{'tout'}`
`TLC...`	Parameters whose names begin with TLC are used for code generation. See the Simulink Coder documentation for more information.	

Parameter	Description	Values
`ToolBar`	Has no effect in Simulink Editor. This parameter will be removed in a future release. 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.	`{'on'} | 'off'`
`TryForcingSFcnDF`	This flag is used for backward compatibility with user S-functions that were written prior to R12.	`'on' | {'off'}`
`TunableVars`	List of global (tunable) parameters. Set in the Model Parameter Configuration dialog box.	`{''}`
`TunableVarsStorageClass`	List of storage classes for their respective tunable parameters. Set in the Model Parameter Configuration dialog box.	`{''}`
`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'}`

Parameter	Description	Values
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'
UnderSpecifiedDataTypeMsg	Detect usage of heuristics to assign signal data types. Set by **Underspecified data types** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	{'none'} | 'warning' | 'error'

Parameter	Description	Values
`UnderspecifiedInitializationDetection`	Select how Simulink software handles initialization of initial conditions for conditionally executed subsystems, Merge blocks, subsystem elapsed time, and Discrete-Time Integrator blocks. Set by **Underspecified initialization detection** on the Configuration Parameters dialog box.	`{'classic'} | 'simplified'`
`UniqueDataStoreMsg`	Specifies diagnostic action to take when the model contains multiple Data Store Memory blocks that specify the same data store name. Set by **Duplicate data store names** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning' | 'error'`
`UnknownTsInhSupMsg`	Detect blocks that have not set whether they allow the model containing them to inherit a sample time. Set by **Unspecified inheritability of sample time** on the **Diagnostics > Sample Time** pane of the Configuration Parameters dialog box.	`'none' | {'warning'} | 'error'`

Parameter	Description	Values
`UnnecessaryDatatypeConvMsg`	Detect unnecessary data type conversion blocks. Set by **Unnecessary type conversions** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	`{'none'} | 'warning'`
`UpdateHistory`	Specifies when to prompt the user about updating the model history. 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. See "Model Information and History" for more information.	`{'UpdateHistoryNever'} | 'UpdateHistoryWhenSave'`
`UpdateModelReferenceTargets`	Specify whether to rebuild simulation and Simulink Coder targets for referenced models before updating, simulating, or generating code for this model. Set by **Rebuild** on the **Model Referencing** pane of the Configuration Parameters dialog box.	`'IfOutOfDate' | 'Force' | 'AssumeUpToDate' | {'IfOutOfDateOrStructuralChange'}`
`UseAnalysisPorts`	For internal use.	

Parameter	Description	Values
`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)
`VersionLoaded`	Simulink version that last saved the model, e.g., `'7.6'`. If you are using a service pack, the `ver` function returns an additional digit, e.g., 7.4.1 (R2009bSP1). See also `SavedSinceLoaded`. To get version information without loading the block diagram into memory, see `Simulink.MDLInfo`.	double (read only)

Parameter	Description	Values
`WideLines`	Draws lines that carry vector or matrix signals wider than lines that carry scalar signals. In the Simulink Editor, set by **Wide Nonscalar Lines** on the **Display** > **Signals & Ports** menu.	`'on' | {'off'}`
`WideVectorLines`	Deprecated. Use `WideLines` instead.	
`WriteAfterReadMsg`	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. Set by **Detect write after read** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`{'UseLocalSettings'} | 'DisableAll' | 'EnableAllAsWarning' | 'EnableAllAsError'`
`WriteAfterWriteMsg`	Specifies diagnostic action to take when the model attempts to store data in a data store twice in succession in the current time step. Set by **Detect write after write** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	`{'UseLocalSettings'} | 'DisableAll' | 'EnableAllAsWarning' | 'EnableAllAsError'`

Parameter	Description	Values
`ZCThreshold`	Specifies the deadband region used during the detection of zero crossings. Signals falling within this region are defined as having crossed through zero. Set by **Signal threshold** on the **Solver** pane of the Configuration Parameters dialog box.	`{'auto'} |` any real number greater than or equal to zero
`ZeroCross`	For internal use.	
`ZeroCrossAlgorithm`	Specifies the algorithm to detect zero crossings when you select a variable-step solver. Set by **Algorithm** on the **Solver** pane of the Configuration Parameters dialog box.	`{'Nonadaptive'} | 'Adaptive'`
`ZeroCrossControl`	Enable zero-crossing detection. Set by **Zero-crossing control** on the **Solver** pane of the Configuration Parameters dialog box.	`{'UseLocalSettings'} | 'EnableAll' | 'DisableAll'`

Parameter	Description	Values
ZoomFactor	Zoom factor of the Simulink Editor window expressed as a percentage of normal (100%) or by the keywords FitSystem or FitSelection. In the Simulink Editor, set by the zoom commands on the **View** menu.	{'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

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 `model` if it is in an open Simulink block diagram. Returns `library` 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 `close_system` is run on block.	function | character vector

Property	Description	Values
`Commented`	Exclude block from simulation.	`{'off'} | 'on' | 'through'`

`CompiledPort-ComplexSignals`	Complexity of port signals after updating diagram. You must compile the model before querying this property. For example:
	`vdp([],[],[],'compile');`
	`d = get_param(gcb,'CompiledPortComplexSignals');`
	`vdp([],[],[],'term');`
		structure array

`CompiledIsActive`	Specifies whether the block status is active or not at compile time. `CompiledIsActive` returns `off` if any one of these conditions is true at compile time: • 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. `CompiledisActive` returns `off` for inactive choices and returns `on` for active choices of Variant Subsystem.	`'off' | 'on'`

Property	Description	Values
`CompiledPortDataTypes`	Data types of port signals after updating diagram. You must compile the model before querying this property. See `CompiledPortComplexSignals` .	structure array

`CompiledPortDesignMin`	Design minimum of port signals after updating diagram. You must compile the model before querying this property. For example:
	`feval(gcs, [],[],[],'compile');`
	`ports = get_param(gcb,'PortHandles');`
	`min = get_param(ports.Outport, 'CompiledPortDesignMin')`
	`feval(model, [],[],[],'term');`
		structure array

`CompiledPortDesignMax`	Design maximum of port signals at compile time. You must compile the model before querying this property. For example:
	`feval(gcs, [],[],[],'compile');`
	`ports = get_param(gcb,'PortHandles');`
	`max = get_param(ports.Outport, 'CompiledPortDesignMax')`
	`feval(model, [],[],[],'term');`
		structure array

`CompiledPortDimensions`	Dimensions of port signals after updating diagram. You must compile the model before querying this property. For details, see “Get Compiled Port Dimensions”.	numeric array

Property	Description	Values
CompiledPortDimensionsMode	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".	double number. 0 indicates the signal does not have a variable size. 1 indicates the signal has a variable size.
CompiledPortFrameData	Frame mode of port signals after updating diagram. You must compile the model before querying this property.	structure array
CompiledPortWidths	Structure of port widths after updating diagram. You must compile the model before querying this property.	structure array
CompiledSampleTime	Block sample time after updating diagram. You must compile the model before querying this property.	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

Property	Description	Values
`DestroyFcn`	Function called when block is destroyed. See “Block Callback Parameters” for details.	MATLAB expression
`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 `log` indicates the block uploads signals. A value of `none` indicates the block does not upload signals. The value `monitor` 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'`

Property	Description	Values
FontAngle	Font angle.	'normal' | 'italic' | 'oblique' | {'auto'}
FontName	Font name.	character array
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.	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. 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. Possible color values are 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.
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 — `{'1.1'}`
`LineHandles`	Handles of lines connected to block.	structure
`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., `set_param(gcb,'linkData',[])`. See “Restore Disabled or Parameterized Links”.	cell array

Property	Description	Values
LinkStatus	Link status of block. Updates out-of-date linked blocks when queried using get_param. 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 ModelCloseFcn is called prior to the block's DeleteFcn and DestroyFcn 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
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'}

Property	Description	Values
`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`	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: • `Type` Specifies the port's type and/or number. The value of this field can be: • *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` The value of this field is a two-element vector, [*x y*], that specifies the port's position. • `SrcBlock` Handle of the block connected to this port. This	structure array

Property	Description	Values
	field is null for output ports and `-1` for unconnected input ports. `SrcBlock` property is a valid source handle for Variant Subsystem blocks. • `SrcPort` Number of the port connected to this port, starting at zero. This field is null for both output ports and unconnected input ports. • `DstBlock` 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. • `DstPort` 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.	

Property	Description	Values
`PortHandles`	The value of this property is a structure that specifies the handles of the block's ports. The structure has the following fields: • `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
	- `Ifaction` Handle of the block's action port. - `Reset` Handle of the block’s reset port.	
`PortRotationType`	Type of port rotation used by this block. This is a read-only property.	`'default' | 'physical'`
`Ports`	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: - `Inport` - `Outport` - `Enable` - `Trigger` - `State` - `LConn` - `RConn` - `Ifaction` - `Reset`	vector

Property	Description	Values
`Position`	Position of block in model window. To help with block alignment, the position you set can differ from the actual block position by a few pixels. Use `get_param` to return the actual position.	vector of coordinates, in pixels: `[left top right bottom]` 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.
`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 get_param. See also LinkStatus.	'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 UserData 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 6-130
- Discontinuities Library Block Parameters on page 6-135
- Discrete Library Block Parameters on page 6-139
- Logic and Bit Operations Library Block Parameters on page 6-156
- Lookup Tables Library Block Parameters on page 6-160
- Math Operations Library Block Parameters on page 6-173
- Model Verification Library Block Parameters on page 6-197
- Model-Wide Utilities Library Block Parameters on page 6-201
- Ports & Subsystems Library Block Parameters on page 6-203
- Signal Attributes Library Block Parameters on page 6-236
- Signal Routing LIbrary Block Parameters on page 6-244
- Sinks Library Block Parameters on page 6-254
- String Library Block Parameters
- Sources Library Block Parameters on page 6-259
- User-Defined Functions Library Block Parameters on page 6-271
- Additional Discrete Library Block Parameters on page 6-272
- Additional Math: Increment - Decrement Library Block Parameters on page 6-275

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 6-276).

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 6-276).

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	Wrap state	{'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)		
ICSourceX	Initial condition source x	{'internal'}| 'external'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
State-Space (StateSpace)		
A	A	matrix — {'1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
B	B	matrix — {'1'}
C	C	matrix — {'1'}
D	D	matrix — {'1'}
InitialCondition	Initial conditions	vector — {'0'}
AbsoluteTolerance	Absolute tolerance	character vector, scalar, or vector — {'auto'} | {'–1'} | any real scalar or vector
ContinuousStateAttributes	State Name	{''} | user-defined
Transfer Fcn (TransferFcn)		
Numerator	Numerator coefficients	vector or matrix — {'[1]'}
Denominator	Denominator coefficients	vector — {'[1 1]'}
AbsoluteTolerance	Absolute tolerance	character vector, scalar, or vector — {'auto'} | {'–1'} | any real scalar or vector
ContinuousStateAttributes	State Name	{''} | user-defined
Transport Delay (TransportDelay)		
DelayTime	Time delay	scalar or vector — {'1'}
InitialOutput	Initial output	scalar or vector — {'0'}
BufferSize	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'}
Variable Time Delay (VariableTimeDelay)		
VariableDelayType	Select delay type	'Variable transport delay' | {'Variable time delay'}
MaximumDelay	Maximum delay	scalar or vector — {'10'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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
Zero-Pole (ZeroPole)		
Zeros	Zeros	vector — {'[1]'}
Poles	Poles	vector — {'[0 -1]'}
Gain	Gain	vector — {'[1]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
AbsoluteTolerance	Absolute tolerance	character vector, scalar, or vector — {'auto'} | {'–1'} | any positive real scalar or vector
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	{'[]'}
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>'

Block (Type)/Parameter	Dialog Box Prompt	Values
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)'
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	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
StateSignalObject	Signal object class Code generation storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal

Block (Type)/Parameter	Dialog Box Prompt	Values
`StateStorageClass`	Code generation storage class	`{'Auto'} | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'`
`CodeGenStateStorageTypeQualifier`	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	
`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
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)'
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 to max or min when overflows occur	{'off'} | 'on'
Discrete FIR Filter (Discrete FIR Filter)		

Block (Type)/Parameter	Dialog Box Prompt	Values
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]'}
InputProcessing	Input processing	'Columns as channels (frame based)' | {'Elements as channels (sample based)'}
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	{'[]'}
TapSumDataTypeStr	Tap sum data type	{'Inherit: Same as input'} | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
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 Code generation storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} | 'Model default' | '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 Code generation storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'
RTWStateStorageTypeQualifier	Code generation storage type qualifier	{''}
Discrete Transfer Fcn (DiscreteTransferFcn)		
Numerator	Numerator coefficients	vector — {'[1]'}
Denominator	Denominator coefficients	vector — {'[1 0.5]'}
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'} | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'`
`RTWStateStorageTypeQualifier`	Code generation storage type qualifier	`{''}`
Discrete Zero-Pole (`DiscreteZeroPole`)		
`Zeros`	Zeros	vector — `{'[1]'}`
`Poles`	Poles	vector — `{'[0 0.5]'}`
`Gain`	Gain	`{'1'}`
`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 Code generation storage class	`Simulink.Signal` object Object of a class that is derived from `Simulink.Signal`
`StateStorageClass`	Code generation storage class	`{'Auto'} | 'Model default' | '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'
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 Code generation storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} | 'Model default' | '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 Code generation storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} | 'Model default' | '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	{''}
StateMustResolveTo SignalObject	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'} | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'
CodeGenStateStorageType Qualifier	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'`
`ZeroCross`	Enable zero-crossing detection	`'off' | {'on'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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)		
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}
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)'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'`
`CheckOORBitShift`	Check for out-of-range 'Bits to shift' in generated code	`{'off'} | 'on'`
`nBitShiftRight`	Deprecated in R2011a	
`nBinPtShiftRight`	Deprecated in R2011a	

Lookup Table Library 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'} | 'Vector' | '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:
		`set_param('myModel/myInterpBlock',...`
		`'TableSpecification',...`
		`'Lookup table object',...`
		`'LookupTableObject','myLUTObject')`

LookupTableObject	Name of lookup table object	{''}
InterpMethod	Interpolation method	'Flat' | {'Linear point-slope'} | 'Nearest' | 'Linear Lagrange'
ExtrapMethod	Extrapolation method	'Clip' | {'Linear'}

Block (Type)/Parameter	Dialog Box Prompt	Values
ValidIndexMayReachLast	Valid index input may reach last index	{'off'} | 'on'
DiagnosticForOutOfRange Input	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	{'[]'}
IntermediateResultsDataTypeStr	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)		
NumberOfTable-Dimensions	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	{'Table and breakpoints'} | 'Lookup table object'
		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:
		`set_param('myModel/myLookupBlock',...`
		`'DataSpecification','Lookup table...`
		`object',...`
		`'LookupTableObject','myLUTObject')`

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
...	...	...
BreakpointsFor-Dimension30Spacing	Spacing	{'1'}
BreakpointsFor-Dimension1	Breakpoints 1	{'[10,22,31]'}
BreakpointsForDimension2	Breakpoints 2	{'[10,22,31]'}
BreakpointsFor-Dimension3	Breakpoints 3	{'[5, 7]'}
...	...	...
BreakpointsFor-Dimension30	Breakpoints 30	{'[1:3]'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InterpMethod	Interpolation method	'Flat' | 'Nearest' | {'Linear point-slope'} | 'Linear Lagrange' | 'Cubic spline'
ExtrapMethod	Extrapolation method	'Clip' | {'Linear'} | 'Cubic spline'
UseLastTableValue	Use last table value for inputs at or above last breakpoint	{'off'} | 'on'
DiagnosticForOutOfRange Input	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'}
BeginIndexSearchUsing PreviousIndexResult	Begin index search using previous index result	{'off'} | 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
UseOneInputPortForAllInputData	Use one input port for all input data	{'off'} | 'on'
SupportTunableTableSize	Support tunable table size in code generation	{'off'} | 'on'
MaximumIndicesForEachDimension	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	{'[]'}
BreakpointsForDimension1DataTypeStr	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)'
BreakpointsForDimension1Min	Breakpoints 1 > Minimum	{'[]'}
BreakpointsForDimension1Max	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	
Lookup Table Dynamic (Lookup Table Dynamic) (masked subsystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
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	{'Explicit values'} | 'Even spacing' | 'Breakpoint object' 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: set_param('myModel/myPrelookupBlock',... 'BreakpointsSpecification',... 'Breakpoint object',... 'BreakpointObject','myBPObject')
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'
DiagnosticForOutOfRangeInput	Diagnostic for out-of-range input	{'None'} | 'Warning' | 'Error'

Block (Type)/Parameter	Dialog Box Prompt	Values
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)		
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)'

Block (Type)/Parameter	Dialog Box Prompt	Values
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	{'++'}
CollapseMode	Sum over	{'All dimensions'} | 'Specified dimension'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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'`
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'}`
`IndexOptions`	See the **IndexOptionArray** parameter for more information.	
`Indices`	See the **IndexParamArray** parameter for more information.	

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutputSizes`	See the **OutputSizeArray** 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'
ShowOutputForNonzero InputValues	Show output port for nonzero input values	{'off'} | 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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'
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)		
Operator	Function	{'exp'} | 'log' | '10^u' | 'log10' | 'magnitude^2' | 'square' | 'pow' | 'conj' | 'reciprocal' | 'hypot' | 'rem' | 'mod' | 'transpose' | 'hermitian'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
SaturateOnIntegerOverflow	Saturate on integer overflow	'off' | {'on'}
Matrix Concatenate (Concatenate)		
NumInputs	Number of inputs	{'2'}
Mode	Mode	'Vector' | {'Multidimensional array'}
ConcatenateDimension	Concatenate dimension	{'2'}
MinMax (MinMax)		

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
SaturateOnIntegerOverflow	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]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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	{'[]'}
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)'

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'
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	{'[]'}
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)'

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'
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	{'[]'}
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'
SaturateOnIntegerOverflow	Saturate on integer overflow	'off' | {'on'}
IntermediateResultsDataTypeStr	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
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'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} | 'on'
OutputDataTypeScalingMode	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	`{''}`
`CMTag10`	CMTag10	`{''}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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	`{''}`
`PermitHierarchical Resolution`	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'`
`PropExecContext OutsideSubsystem`	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	"Memory section for initialize/ terminate functions" on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	{'Inherit from model'} | 'Default' | list of memory sections from model's package
RTWMemSecFuncExecute	"Memory section for execution functions" on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	{'Inherit from model'} | 'Default' | list of memory sections from model's package
RTWMemSecDataConstants	"Memory section for constants" on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	{'Inherit from model'} | 'Default' | list of memory sections from model's package
RTWMemSecDataInternal	"Memory section for internal data" on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	{'Inherit from model'} | 'Default' | list of memory sections from model's package
RTWMemSecDataParameters	"Memory section for parameters" on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	{'Inherit from model'} | 'Default' | list of memory sections from model's package

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'
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	{''}
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	{''}
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`	“Memory section for initialize/terminate functions” on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	`{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`RTWMemSecFuncExecute`	“Memory section for execution functions” on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	`{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`RTWMemSecDataConstants`	“Memory section for constants” on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	`{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`RTWMemSecDataInternal`	“Memory section for internal data” on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	`{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`RTWMemSecDataParameters`	“Memory section for parameters” on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	`{'Inherit from model'}` | `'Default'` | list of memory sections from model's package

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	{''}
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	{''}

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	{''}
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`	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'}
IterationVariable DataType	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	{''}
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	{''}

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)	{''}
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
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	`{''}`
`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	`{''}`
`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' | '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'
LatchByDelaying OutsideSignal	Latch input by delaying outside signal	{'off'} | 'on'
LatchInputFor FeedbackSignals	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'}
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>'
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
ModelName	The value of ModelNameDialog stripped of any filename extension that you provided. For backward compatibility, setting ModelName programatically 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.	character vector — Set automatically when ModelNameDialog is set.
ModelFile	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.	character vector — Set automatically when ModelNameDialog is set.
ProtectedModel	Read-only boolean indicating whether the model referenced by the block is protected (on) or unprotected (off).	boolean — 'off' | 'on' — Set automatically when ModelNameDialog is set.
ParameterArgumentNames	Names of model arguments that the referenced model defines. Corresponds to the **Name** column in the table under **Arguments** > **Model arguments**.	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
`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)'`
`Variant`	Specifies whether the Model block references variant models or Variant Subsystems.	`{'off'} | 'on'`
`VariantConfigurationObject`	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 `Simulink.VariantConfigurationData` object.
`OverrideUsingVariant`	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.
`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.

Block (Type)/Parameter	Dialog Box Prompt	Values
GeneratePreprocessor Conditionals	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'}
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'`
`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`
`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'} | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'`
`SignalName`	Signal name	character vector

Block (Type)/Parameter	Dialog Box Prompt	Values
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	{''}
PermitHierarchical Resolution	Permit hierarchical resolution	{'All'} | 'ExplicitOnly' | 'None'
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'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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)		

Block (Type)/Parameter	Dialog Box Prompt	Values
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	{''}
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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
Trigger (TriggerPort)		
TriggerType	Trigger type	{'rising'} | 'falling' | 'either' | 'function-call'
IsSimulinkFunction	Configure the Function-call subsystem to be a Simulink Function	{'off'} | 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
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	{'[]'}
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)		

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	{''}
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	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	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
`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		
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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

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)'}
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]'}
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'}
StateMustResolveTo SignalObject	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	{''}
DataLoggingLimit DataPoints	Limit data points to last	'off' | {'on'}
DataLoggingMaxPoints	Limit data points to last	non-zero integer {5000}
DataLoggingDecimateDat a	Decimation	'off' | {'on'}
DataLoggingLimit DataPoints	Decimation	non-zero integer {2}
StateStorageClass	Storage class	{'Auto'} | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'
StateSignalObject	Signal object class Storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
RTWStateStorageType Qualifier	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)		
DataStoreElements	Corresponds to the parameters on the **Element Selection** tab of the block dialog box. See “Specification using the command line”.	
DataStoreName	Data store name	{'A'}
SampleTime	Sample time	{'0'}
Data Store Write (DataStoreWrite)		
DataStoreElements	Corresponds to the parameters on the **Element Assignment** tab of the block dialog box. See “Specification using the command line”.	
DataStoreName	Data store name	{'A'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Demux (Demux)		

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'}`
`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	`{'[]'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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)		

Block (Type)/Parameter	Dialog Box Prompt	Values
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}'}
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'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} | 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
AllowDiffInputSizes	Allow different data input sizes (Results in variable-size output signal)	{'off'} | 'on'
Mux (Mux)		
Inputs	Number of inputs	{'2'}
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 the **IndexOptionArray** parameter for more information.	
Indices	See the **IndexParamArray** parameter for more information.	
OutputSizes	See the **IndexParamArray** parameter for more information.	

Block (Type)/Parameter	Dialog Box Prompt	Values
Switch (Switch)		
Criteria	Criteria for passing first input	{'u2 >= Threshold'} | 'u2 > Threshold' | 'u2 ~= 0'
Threshold	Threshold	{'0'}
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'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} | 'on'
AllowDiffInputSizes	Allow different input sizes (Results in variable-size output signal)	{'off'} | 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
Variant Source (tVariantSource)		
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'
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'

Block (Type)/Parameter	Dialog Box Prompt	Values
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.
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'}
YMin		{'-5'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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 Outport 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 Outport Blocks" (Simulink Coder).	{'Auto'} | 'Model default' | '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'
Scope (Scope)		

Block (Type)/Parameter	Dialog Box Prompt	Values
`Floating`		`{'off'} | 'on'`
Location		vector — `{'[188 390 512 629]'}`
`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		
Terminator		

Block (Type)/Parameter	Dialog Box Prompt	Values
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: SlDemoSign'}
Value	Value	{'SlDemoSign.Positive'} | 'SlDemoSign.Zero' | 'SlDemoSign.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'
LatchByDelayingOutsideSignal	Latch input by delaying outside signal	{'off'} | 'on'
LatchInputForFeedbackSignals	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'}
InitialOutput	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)'
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'
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>'
OutputDataTypeScalingMode	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'}
ActiveSignal	Signals	{'Signal 1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}
Samples	Samples per period	{'10'}
Offset	Number of offset samples	{'0'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}

String Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Compose String (Compose String)		
Format	Format	scalar {""%d %f""}| ""%d"" | ""%i"" | ""%o"" | ""%x"" | ""%X"" | ""%f"" | ""%e"" | ""%E"" | ""%g"" | ""%G"" | ""%s"" | optional operators and text
OutDataTypeStr	Output data type	{"stringtype(255)"} | "stringtype(*N*)" | "string"
Scan String (Scan String)		
Format	Format	scalar {""%d %f""}| ""%d"" | ""%i"" | ""%o"" | ""%x"" | ""%X"" | ""%f"" | ""%e"" | ""%E"" | ""%g"" | ""%G"" | ""%s"" | optional operators and text
String Compare (String Compare)		
CaseSensitive	Case sensitivity for string compare	'off' | {'on'}
CompareOption	Amount of characters to compare	{"Entire string"} | "First N characters"
NumberOfCharacters	Number of characters to compare	{'1'} | scalar
String Concatenate (String Concatenate)		
Inputs	Number of inputs	{"2"}
OutDataTypeStr	Output data type	{"stringtype(128)"} | "stringtype(*N*)" | "string"
String Constant (String Constant)		
String	String	{"Hello!"} | scalar

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{"stringtype(128)"} | "stringtype(*N*)" | "string"
String to ASCII (String to ASCII)		
MaximumLength	Maximum length	{"31"} | scalar
String to Double (String to Double)		
Format	Format	scalar {""%d %f""}| ""%d"" | ""%i"" | ""%o"" | ""%x"" | ""%X"" | ""%f"" | ""%e"" | ""%E"" | ""%g"" | ""%G"" | ""%s"" | optional operators and text
String to Enum (String to Enum)		
OutDataTypeStr	Output data type	{"Enum: SlDemoSign"} | <data type expression>
String to Single (String to Single)		
Format	Format	scalar {""%f""}| ""%d"" | ""%i"" | ""%o"" | ""%x"" | ""%X"" | ""%f"" | ""%e"" | ""%E"" | ""%g"" | ""%G"" | ""%s"" | optional operators and text
Substring (Substring)		
InheritMaximumLength	Inherit maxiimum length from input	'off' | {'on'}
OutDataTypeStr	Output data type	{"stringtype(31)"} | "stringtype(*N*)" | "string"
StringFromIdxToEnd	Output string from 'idx' to end	{'off'} | 'on'

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`	Arguments	`{''}`
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 Library Block 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]'}`
`InitialCondition`	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)		
vinit	Initial condition	{'0.0'}

Block (Type)/Parameter	Dialog Box Prompt	Values
`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 Library 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'}`
`MaskValues`	Dialog parameter values.	cell array `{'[]'}`

Parameter	Description/Prompt	Values
`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.

7

Tools and Apps — Alphabetical List

Simulation Data Inspector

Inspect and compare data and simulation results to validate and iterate model designs

Description

The Simulation Data Inspector visualizes and compares multiple kinds of data.

Using the Simulation Data Inspector, you can inspect and compare time series data at multiple stages of your workflow. This example workflow shows how the Simulation Data Inspector supports all stages of the design cycle:

1. “View Data with the Simulation Data Inspector”.

   Run a simulation in a model configured to log data to the Simulation Data Inspector, or import data from the workspace or a MAT-file. You can view and verify model input data or inspect logged simulation data while iteratively modifying your model diagram, parameter values, or model configuration.
2. “Inspect Simulation Data”.

   Plot signals on multiple subplots, zoom in and out on specified plot axes, and use data cursors to understand and evaluate the data. “Create Plots Using the Simulation Data Inspector” to tell your story.
3. “Compare Simulation Data”

   Compare individual signals or simulation runs and analyze your comparison results with relative, absolute, and time tolerances. The compare tools in the Simulation Data Inspector facilitate iterative design and allow you to highlight signals that do not meet your tolerance requirements. For more information about the comparison operation, see “How the Simulation Data Inspector Compares Data”.
4. “Save and Share Simulation Data Inspector Data and Views”.

   Share your findings with others by saving Simulation Data Inspector data and views.

You can also harness the capabilities of the Simulation Data Inspector from the command line. For more information, see “Inspect and Compare Data Programmatically”.

Open the Simulation Data Inspector

- Simulink Editor toolbar: Click the Simulation Data Inspector icon.
- Click the streaming badge on a signal to open the Simulation Data Inspector and plot the signal.
- MATLAB command prompt: Enter `Simulink.sdi.view`.

Examples

Add Signals to a Run

This example shows how to use `Simulink.sdi.createRunOrAddToStreamedRun` to add data to an existing run for a model. In this example, you add logged states data to the run created through simulation.

Simulate the Model

Simulate the model to generate data. The model `slexAircraftExample` is configured to log outputs, so the Simulation Data Inspector automatically creates a run with the logged outport data. Using this simulation syntax, `out` contains the outport data (`yout`) and the states data (`xout`).

```
load_system('slexAircraftExample')
out = sim('slexAircraftExample','ReturnWorkspaceOutputs','on',...
             'SaveFormat','Dataset');
```

Add Logged States Data to Run

The Simulation Data Inspector automatically created a run for the logged outport data. Add the logged state data to the existing run using `Simulink.sdi.createRunOrAddToStreamedRun`.

```
Simulink.sdi.createRunOrAddToStreamedRun('slexAircraftExample','Run 1',...
               {'out'},{out});
```

Open the Simulation Data Inspector to View Results

Using `Simulink.sdi.createRunOrAddToStreamedRun` avoids redundancy in the data shown in the Simulation Data Inspector. Using `Simulink.sdi.createRun` to bring the states data into the Simulation Data Inspector creates a second run. `Simulink.sdi.addToRun` creates a duplicate signal from the outport data. Using `Simulink.sdi.createRunOrAddToStreamedRun`, you can include all simulation data in a single run without duplicating any signals.

```
Simulink.sdi.view
```

Modify Parameter for Several Runs

This example shows how to modify a parameter for all the runs in the Simulation Data Inspector programmatically.

Generate Runs

Load the `vdp` model and mark the `x1` and `x2` signals for logging. Then, run several simulations.

```
% Clear all data from the Simulation Data Inspector repository
Simulink.sdi.clear

% Load the model and mark signals of interest for streaming
load_system('vdp')
Simulink.sdi.markSignalForStreaming('vdp/x1',1,'on')
Simulink.sdi.markSignalForStreaming('vdp/x2',1,'on')

% Simulate the model with several Mu values
for gain = 1:5
    gainVal = num2str(gain);
    set_param('vdp/Mu','Gain',gainVal)
    sim('vdp')
end
```

Use `Simulink.sdi.getRunCount` to Assign Tolerance to x1 Signals

```
count = Simulink.sdi.getRunCount;

for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
    vdpRun = Simulink.sdi.getRun(runID);
    sig = vdpRun.getSignalByIndex(1);
    sig.AbsTol = 0.1;
end
```

```
% Open the Simulation Data Inspector to view your data
Simulink.sdi.view
```

- “View Data with the Simulation Data Inspector”
- “Inspect Simulation Data”
- “Compare Simulation Data”
- “Iterate Model Design Using the Simulation Data Inspector”

Programmatic Use

`Simulink.sdi.view` opens the Simulation Data Inspector from the MATLAB command line.

See Also

Functions
`Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.snapshot`

Topics

“View Data with the Simulation Data Inspector”
“Inspect Simulation Data”
“Compare Simulation Data”
“Iterate Model Design Using the Simulation Data Inspector”

Introduced in R2010b

Fixed-Point Tool

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

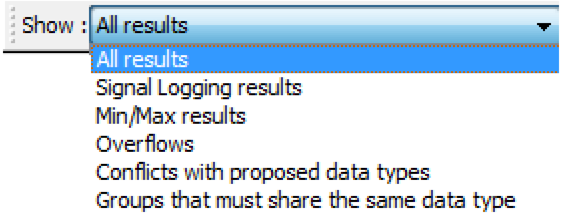
The Fixed-Point Tool's main toolbar appears near the top of the Fixed-Point Tool window under the Fixed-Point Tool's menu.

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.
DT	Propose data types. Propose fraction lengths for specified word lengths or propose word lengths for specified fraction lengths.
DT✓	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:


Show :
All results
All results
Signal Logging results
Min/Max results
Overflows
Conflicts with proposed data types
Groups that must share the same data type

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 `Force Off`.
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
Conflicts with proposed data types	For the selected tree node, displays results that have potential data typing or scaling issues. **Prerequisites**: This information is available only after you propose data types. The Fixed-Point Tool marks these results with a yellow or red icon, as shown here: ⚠ The proposed data type poses potential issues for this object. Open the Result Details tab to review these issues. ❌ 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.
Groups that must share the same data type	For the selected tree node, displays blocks that must share the same data type because of data type propagation rules. **Prerequisites**: This information is available only after you propose fraction lengths. 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: • If the selected tree node is the model root 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. • If the selected tree node is a subsystem 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.

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	
`mmo`	`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 8-8.

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 `sfix` 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	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. **Note** The Fixed-Point Tool uses this parameter when it proposes data types.

Column Label	Description
InitValueMin	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. **Note** The Fixed-Point Tool uses this parameter when it proposes data types.
LogSignal	Check box that allows you to enable or disable signal logging for an object.
ModelRequiredMin	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. **Note** The Fixed-Point Tool uses this parameter when it proposes data types.
ModelRequiredMax	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. **Note** The Fixed-Point Tool uses this parameter when it proposes data types.
Name	Identifies path and name of block.
OverflowWraps	Number of overflows that wrap during simulation.
ProposedDT	Data type that the Fixed-Point Tool proposes.

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 `sfix` 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 8-8
- "Changing Column Order and Width" on page 8-10
- "Sorting by Columns" on page 8-10

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.

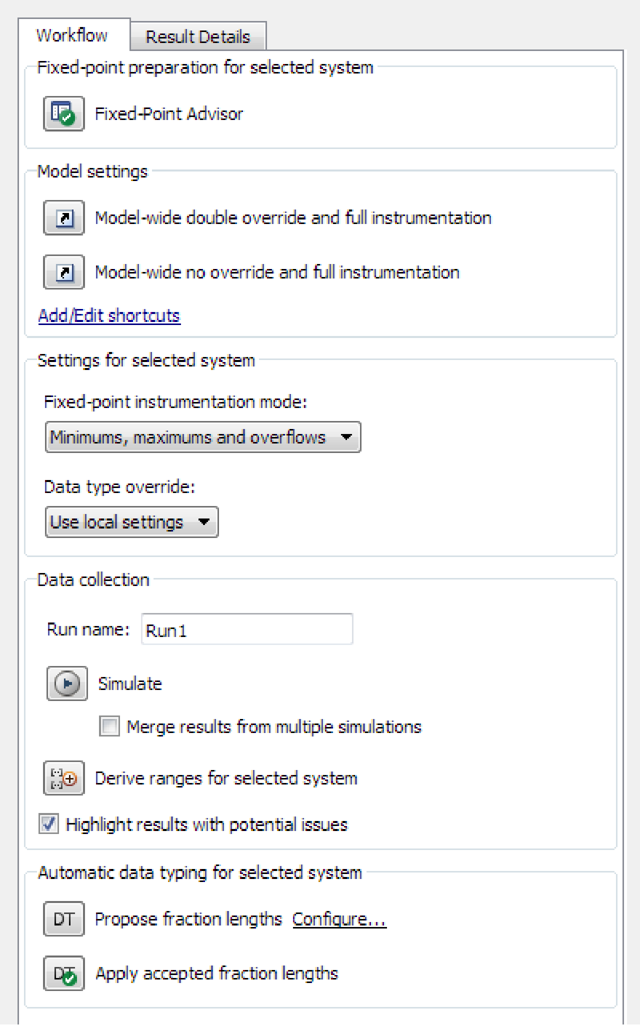
Sorting by Columns

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.



Workflow | Result Details

Fixed-point preparation for selected system

Fixed-Point Advisor

Model settings

Model-wide double override and full instrumentation

Model-wide no override and full instrumentation

Add/Edit shortcuts

Settings for selected system

Fixed-point instrumentation mode: Minimums, maximums and overflows

Data type override: Use local settings

Data collection

Run name: Run1

Simulate

□ Merge results from multiple simulations

Derive ranges for selected system

☑ Highlight results with potential issues

Automatic data typing for selected system

DT Propose fraction lengths Configure...

Apply accepted fraction lengths

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 8-12.
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 8-13.
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
Range collection using double override	Use this configuration to observe ideal numeric behavior of the model and collect ranges for data type proposals. This configuration sets: • **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` By default, a button for this configuration appears in the **Configure model settings** pane.

Factory Default Configuration	Description
Range collection with specified data types	Use this configuration to collect ranges of actual model and to validate current behavior. This configuration sets: • **Run name** to `NoOverride` • **Fixed-point instrumentation mode** to `Minimums, maximums and overflows` • **Data type override** to `Use local settings` By default, a button for this shortcut appears in the **Configure model settings** pane.
Remove overrides and disable range collection	Use this configuration to cleanup settings after finishing fixed-point conversion and to restore maximum simulation speed. This configuration sets: • **Fixed-point instrumentation mode** to `Off` • **Data type override** to `Use local settings` By default, a button for this shortcut appears in the **Configure model settings** pane.

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 8-18

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

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.

9

Model Advisor Checks

Simulink Checks

In this section...

In this section...

In this section...

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 9-32
- “Check usage of Outport blocks” on page 9-35
- “Check usage of Discrete-Time Integrator blocks” on page 9-47
- “Check model settings for migration to simplified initialization mode” on page 9-48

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.	Select the following optimization check boxes on the **Optimization** pane in the Configuration Parameters dialog box: • "Remove root level I/O zero initialization" (Simulink Coder) • "Remove internal data zero initialization" (Simulink Coder) Select the following optimization check boxes on the **Optimization** pane in the Configuration Parameters dialog box: • "Inline invariant signals" (Simulink Coder) (only if you have a Simulink Coder license) Select the following optimization check boxes in the Configuration Parameters dialog box: • "Block reduction" • "Conditional input branch execution" • "Implement logic signals as Boolean data (vs. double)" • "Use memset to initialize floats and doubles to 0.0" (Simulink Coder) • "Remove code from floating-point to integer conversions that wraps out-of-range values" (Simulink Coder) (only if you have a Simulink Coder license) • "Signal storage reuse" • "Enable local block outputs" (Simulink Coder) • "Reuse local block outputs" (Simulink Coder) • "Eliminate superfluous local variables (Expression folding)" (Simulink Coder) Select the following optimization check boxes on the **Optimization** pane in the Configuration Parameters dialog box: **Note** Model Advisor checks these parameters only if there is a Stateflow chart in the model.

Condition	Recommended Action
	• “Use bitsets for storing state configuration” (Simulink Coder) • “Use bitsets for storing Boolean data” (Simulink Coder)
“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: • **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: • 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 **Pass reusable subsystem outputs as** to `Individual arguments`. For details, see “Pass reusable subsystem outputs as” (Simulink Coder).

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

- “Model Configuration Parameters: Code Generation Optimization” (Simulink Coder)

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

“Create Tunable Calibration Parameter 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 9-56.

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 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

“Convert to Variant Subsystem” on page 1-216

Identify Variant Model blocks and convert those to Variant Subsystem containing Model block choices

Check ID: `mathworks.design.ConvertMdlrefVarToVSS`

Search the model to identify Variant Model blocks.

Results and Recommended Actions

Condition	Recommended Action
Variant Model blocks available in the model are listed.	Convert these blocks to Variant Subsystem blocks.

See Also

“Convert to Variants”

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 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.

Condition	Recommended Action
The data store variable names are not strongly typed in one of the following: • **Signal Attributes** pane of the **Block Parameters** dialog for the Date Store Memory block • Global data store name	Specify a data type other than auto by taking one of the following actions: • Choose a data type other than `Inherit: auto` 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 `Simulink.Signal` 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 9-23
- “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 `Simulink.Bus.createMATLABStructure` 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 `Simulink.Parameter` object, and creating a `Simulink.Bus` object to use as the data type of the bus signal and of the parameter object. To control numeric data types, use the `Simulink.BusElement` 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.	The check identifies these instances and allows you to upgrade the model. 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.CallslDataTypeAndScale`

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 `slDataTypeAndScale`. Cases listed require you to replace calls to `slDataTypeAndScale`.
Recommended Replacement Cases	For the listed cases, it is recommended that you manually or automatically replace calls to `slDataTypeAndScale`.
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 `Simulink.BlockDiagram.addBusToVector` 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 `error`.

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 Outport 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 9-5
- `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. • 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 9-5.

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.	**1** In the Configuration Parameters dialog box, set “Detect multiple driving blocks executing at the same time step” to `error`. **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.	Replace both the Mux block used to produce the input signal and the Merge block with one multi-input Merge block. Single-input Merge blocks are not supported in the simplified initialization mode.
Check for root Merge blocks that have an unspecified **Initial output** value.	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. 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”.

Condition	Recommended Action
Check for Merge blocks with nonzero input port offsets.	Clear the **Allow unequal port widths** parameter of the Merge block. **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.
Check for Merge blocks that have unconnected inputs or that have inputs from non-conditionally executed subsystems.	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. 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.
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.	Verify that input signals to each Merge block have the same **Sample time**. Failure to do so could result in unpredictable behavior. Consequently, the simplified initialization mode does not allow inconsistent sample times.
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.	Verify that the following behavior is acceptable. 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. **Note** Simulink generates a warning that the initial value of the signal object has been ignored.

See Also

- "Migrating to Simplified Initialization Mode Overview" on page 9-5
- "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 9-5.

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.	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. 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 `sl_subsys_semantics`. A standard data signal may require an additional signal copy for one of the following reasons: • 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 `Simulink.Signal` 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
Check for Outport blocks that require an explicit signal copy.	An explicit copy of the bus signal driving the Outport block is required for the initialization of the output signal of the corresponding subsystem. Insert a Signal Conversion block before the Outport block, then set the Output parameter of the Signal Conversion block to `Bus copy`. A standard data signal may require an additional signal copy for one or more of the following reasons: • 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 `Simulink.Signal` object with an explicit initial value.
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.	When Outport blocks are driving a Merge block, do not set their **Output when disabled** parameters to `reset`.
Check for merged Outport blocks that are driven by nested conditionally executed subsystems.	Determine if the new behavior of the Outport blocks is acceptable. If it is not acceptable, modify the model to account for the new behavior before migrating to the simplified initialization mode.

Condition	Recommended Action
Check for merged Outport blocks that reset when the blocks are disabled.	Set the **Output when disabled** parameter of the Outport block to `held`. This setting is required because the Outport block connects to a Merge block. For more information, see Outport.
Check for Outport 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 (`[]`), it inherits the initial output from the source blocks. If at least one of the sources of the Outport 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 Outport 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.
Check Outport blocks that have automatic rate transitions.	Simulink has inserted a Rate Transition block at the input of the Outport block. Specify the **Initial output** parameter for each Outport block. Otherwise, perform the following procedure: **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.

Condition	Recommended Action
Check Outport blocks that have a special signal storage requirement and have an undefined **Initial output** value.	Verify that the following behavior is acceptable. 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.
Check the **Initial output** setting of Outport blocks that reset when they are disabled.	Specify the **Initial output** parameter of the Outport block. You must specify the **Initial output** value for blocks that are configured to reset when they become disabled.
Check the **Initial output** setting for Outport blocks that pass through a function-call data dependency signal.	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: **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.	Verify that the following behavior is acceptable. 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. **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.
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.	Verify that the following behavior is acceptable. 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.

Condition	Recommended Action
Check for Outport blocks that obtain the **Initial output** value from an input signal when they are migrated.	Verify that the following behavior is acceptable. 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. If this behavior is not acceptable, modify your model before you migrate to the simplified initialization mode.
Check for outer Outport blocks that have an explicit **Initial output**.	Verify that the following behavior is acceptable. In classic initialization mode, the **Initial output** and **Output when disabled** parameters of the Outport block must match those of their source Outport blocks. In simplified initialization mode, Simulink sets the **Initial output** parameter of outer Outport blocks to `[]` (empty matrix) and **Output when disabled** parameter to `held`.
Check for conditionally executed subsystems that propagate execution context across the output boundary.	Verify that the following behavior is acceptable. 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.

Condition	Recommended Action
Check for blocks that read input from conditionally executed subsystems during initialization.	Verify that the following behavior is acceptable. 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. 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.
Check for a migration conflict for Outport blocks that use a `Dialog` as the **Source of initial output value**.	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. The Outport block will maintain its current settings and use its specified **Initial output** value.

Condition	Recommended Action
Check for a migration conflict for Outport blocks that use `Input signal` as the **Source of initial output** value.	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. The Outport block currently specifies an **Initial output** of `[]` (empty matrix), and **the Output when disabled** as `held`. This means that each outport does not perform initialization, but implicitly relies on source blocks to initialize its input signal. After migration, the parameter **Source of initial output value** will be set to `Input signal` to reflect this behavior.
Check for a migration conflict for Outport blocks that have SimEvents semantics.	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. The Outport blocks will continue to use an **Initial output** value of `[]` (empty matrix) and an **Output when disabled** setting of `held`. Simulink will maintain these settings because their parent conditionally executed subsystems are connected to SimEvents blocks.

Condition	Recommended Action
Check for a migration conflict for innermost Outport blocks with variable-size input and unspecified **Initial output**.	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. 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.
Check for a migration conflict for Outport blocks that use a default ground value as the Initial output.	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.
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 9-5
- “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 9-5.

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.

Condition	Recommended Action
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.	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.

See Also

- "Migrating to Simplified Initialization Mode Overview" on page 9-5
- "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 9-5.

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 9-5
- "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 `mdlOutputs` method.	Modify Continuous States at a major time step and use `ssSetSolverNeedsReset` function in S-function code.
Continuous states are modified in the `mdlUpdate` method.	Modify Continuous States only at a major time step and use `ssSetSolverNeedsReset` function in S-function code.
S-function discrete states are modified in the `mdlOutputs` at a minor step.	Modify the discrete states only at a major step guarded by `ssIsMajorTimeStep` function.
S-function mode vector is modified in the `mdlOutputs` method at a minor step.	Modify the mode vector only at a major step guarded by `sslsMajorTimeStep` function.
S-function is using static or global variables to declare internal states.	Declare the states explicitly using `ssSetNumDiscStates` function or "Model Global Data by Creating Data Stores".

Condition	Recommended Action
S-function has continuous states but sample time is not declared continuous.	Specify continuous sample time using `ssSetSampleTime` function.
S-function has discrete states but the `mdlOutputs` and `mdlUpdate` methods are combined.	Define the `mdlOutputs` and `mdlUpdate` methods separately and modify discrete states only in `mdlUpdate` method.
S-function sets the `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` 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 **Diagnostic 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.	• 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 `slCharacterEncoding`. 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.

Condition	Recommended Action
Model contains Unit Delay blocks with **Sample time** set to `-1` that inherit a continuous sample time.	Replace Unit Delay blocks with Memory blocks.

Check Data Store Memory blocks for multitasking

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 model for Aerospace Blockset navigation blocks

Check ID: `mathworks.design.Aeroblks.CheckNAV`

Searches for Three-Axis Inertial Measurement Unit, Three-Axis Gyroscope, and Three-Axis Accelerometer blocks prior to 3.21 (R2018a).

Description

This check searches for Three-Axis Inertial Measurement Unit, Three-Axis Gyroscope, and Three-Axis Accelerometer blocks that have been updated in R2018a.

Results and Recommended Actions

Condition	Recommended Action
Three-Axis Inertial Measurement Unit, Three-Axis Gyroscope, and Three-Axis Accelerometer blocks prior to R2018a.	In R2018a or later, if you did not previously solve for steady state conditions, save the model now. If you previously solved for steady state conditions for the model, solve for these steady state conditions again, and then save the model.

See Also

- Three-Axis Accelerometer
- Three-Axis Gyroscope
- Three-Axis Inertial Measurement Unit

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 `imread()` 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 `MaskTabNames` 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: • Block input and output data types are set correctly. • In the Configuration Parameters dialog box, **Default for underspecified data type** is set to `single`.
Model uses a library standard that is not optimal for strict-single designs.	Verify that: • All target-specific math libraries used by the model support single-precision implementations. Set **Configuration Parameters > Standard math library** to `C99 (ISO)`.

Condition	Recommended Action
Logic signals are not implemented as Boolean data.	Verify that: • 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
```

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 `get_param(block,CompiledSampleTime)`	None
Some MATLAB files call `get_param(block,CompiledSampleTime)`	If files use the block `CompiledSampleTime` 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 `set_param(control_port,'DisallowConstTsAndPrmTs'`
The model has an invalid control port value in a conditionally executed subsystem.	At the command prompt, run `set_param(control_port,'DisallowConstTsAndPrmTs'`

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 Outport blocks — Enable the **Output as nonvirtual bus in parent model** parameter.
- For Model blocks — For ports whose Outport 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 Outport 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 Outport 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 Outport block is driven by a nonvirtual bus
- Have Model blocks that reference models containing Outport blocks that have been fixed — Clicking **Upgrade Model** updates Model blocks referencing the models that had Outport 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 model for S-function upgrade issues

Check ID: `'mathworks.design.CheckForSFcnUpgradeIssues'`

Use this check on your model to identify your S-function's upgrade compatibility issues. These issues may include the use of 32-bit APIs, compilation with incompatible options, or use of deprecated separate complex APIs. Some common issues and information related to the fixes are described in results and recommendations section below.

Description

When upgrading your S-functions to use the features in the latest release, this check scans your model to warn against S-function upgrade incompatibility issues. If the result of this check gives a warning or error, fix your C MEX S-functions according to the description.

Results and Recommended Actions

Condition	Recommended Action
Custom-built S-functions are not supported.	Recompile your S-function with available compatible options. See "Custom-built MEX File Not Supported In Current Release" (MATLAB) for more information.
S-function is not compiled with the latest API (`mex -R2018a`).	Recompile using the latest flag (`mex -R2018a`). See "MEX File Is Compiled With Outdated Option" (MATLAB) for more information.
S-function uses 32-bit functions.	Modify your code according to the instructions in "MEX File Calls A 32-bit Function" (MATLAB).

Condition	Recommended Action
S-function is using deprecated separate complex APIs (mexGetPi, mexSetPi, mexGetImagData, mexSetImagData).	Use interleaved complex APIs and recompile your code with the latest flag (`mex -R2018a`). See “Upgrade MEX Files to Use Interleaved Complex API” (MATLAB) for more information.
S-function is using deprecated type-unsafe data API (mxGetData, mexSetData).	Use type-safe data APIs and recompile your code. See “MEX File Calls An Untyped Data Access Function” (MATLAB) for more information.
S-function is compiled with a future release and not supported in current release.	See “MEX File Built In MATLAB Release Not Supported In Current Release” (MATLAB) to recompile your files.

See Also

- “MATLAB Data in C S-Functions”

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.

Condition	Recommended Action
Simulation mode is rapid accelerator. Upgrade Advisor did not find signals with signal logging enabled.	None. The 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: • 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 `Inherit: auto`	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 `Bus: <BusObject>`.
Signal Conversion block **Output** parameter matches input bus type	A Signal Conversion block whose **Output** parameter is set to `Nonvirtual bus` requires a virtual bus input. A Signal Conversion block whose **Output** parameter is set to `Virtual bus` requires a nonvirtual bus input.	To create a copy of the input signal, set **Output** to `Signal copy`.
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 `1` or `-1 (inherit)`.

Modeling Pattern	Issue	Solution
Mux block with nonvirtual bus inputs	A Mux block cannot accept nonvirtual bus signals.	To treat the output as an array, replace the Mux block with a Vector Concatenate block. If you want a virtual bus output, use a Bus Creator block to combine the signals.
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 `Y0` port bus input to a vector.	Add a Bus to Vector block before the Assignment block.
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 Outport block with a single-element bus object data type	The input to the Outport block must be a bus if it specifies a bus object as its data type.	In the Outport 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 9-5
- `Simulink.BlockDiagram.addBusToVector`

Check for root outports with constant sample time

Check ID: `mathworks.design.CheckConstRootOutportWithInterfaceUpgrade`

Use this check to identify root outports 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 Outport 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: • 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.

Condition	Recommended Action
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 ^. 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: • Add a data dependency between the blocks. • Set block priority. • 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	The advisor provides a model name that is based on the Subsystem block name and is unique in the MATLAB path. The model name cannot exceed 59 characters. **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.
Conversion data file name	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 `_conversion_data.mat`. For example, if the subsystem is in a model named `myModel`, the conversion file name is `myModel_conversion_data.mat`. You can save the conversion data in a MAT-file (default) or a MATLAB file. If you use a `.m` file extension, the advisor serializes all variables to a MATLAB file. **Note** If the top model uses a data dictionary, you cannot select this option.
Fix errors automatically	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. If you enable this parameter, the advisor fixes all conversion errors that it can, without displaying the **Fix** button.

Input Parameter	Description
Replace content of a subsystem with a Model block	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. • 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. 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.

Input Parameter	Description
Check simulation results after conversion	Compare the results of simulating the top model for the referenced model to the results of simulating the baseline model that has the subsystem. 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: • **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 `inf`, the advisor stops after 10 seconds. • **Absolute tolerance** — Specify a value if you do not want to use the default of `'1e-06'`. • **Relative tolerance** — Specify a value if you do not want to use the default of `'1e-03'`. 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”.
Stop time	By default, the advisor uses the stop time of the top model, unless the stop time of the top model is `inf`. If the stop time of the top model is `inf`, 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”. To use this option, select **Check simulation results after conversion**.

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. • `Normal` (default) • `Accelerator`

After you configure the advisor, to start the conversion checks, click **Run this task**.

11

Performance Advisor Checks

Simulink Performance Advisor Checks

In this section...

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

- “Model Configuration Parameters: Code Generation Optimization” (Simulink Coder)
- “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 `(mmo)`, 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

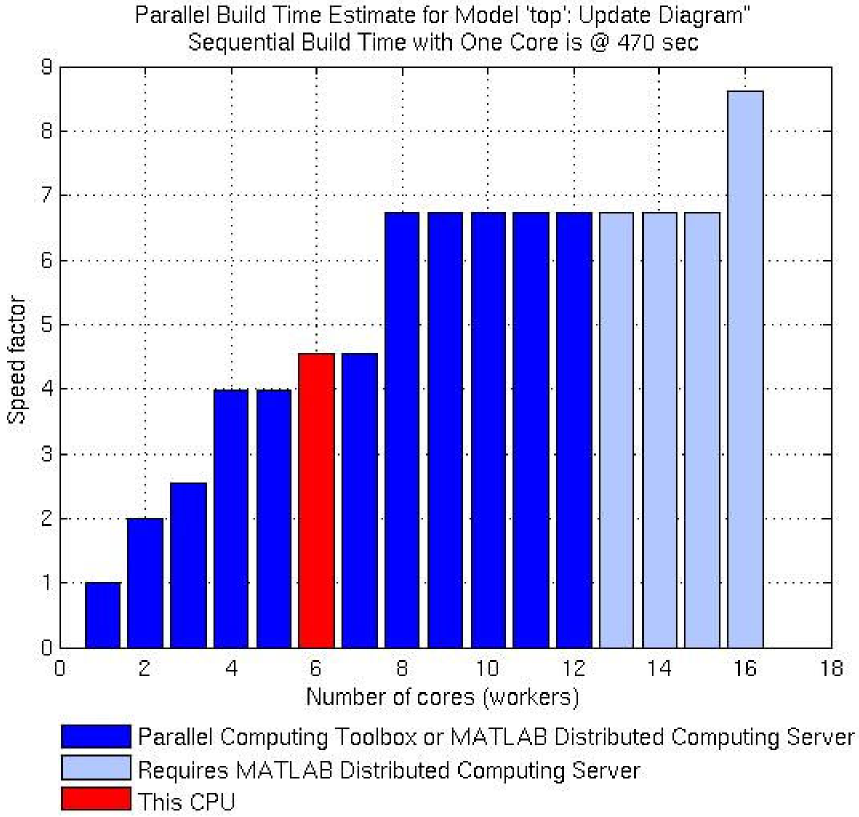
- Fixed-Point Instrumentation and Data Type Override (Fixed-Point Designer)

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 + Parallel Build Overhead Time Estimation Factor)*(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

- “Solver Types”

- “Speed Up Simulation”
- “Improve Simulation Performance Using Performance Advisor”

Check zero-crossing impact on continuous integration

If your model contains zero-crossings which do not impact the continuous integration, the simulation might slow down when all the following conditions are satisfied:

- The model uses a variable-step solver.
- The model contains blocks that have continuous states and zero-crossings.
- Some of the zero-crossings do not affect the integration of the continuous states.

Setting the `MinimalZcImpactIntegration` parameter to `On` might speed up simulation.

See Also

- “Speed Up Simulation”
- “Solver Types”
- “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: • 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	• 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

- “Solver Types”
- “Speed Up Simulation”
- “Improve Simulation Performance Using Performance Advisor”

Select multi-thread co-simulation setting on or off

Adjust co-simulation settings for better performance and accuracy.

- Validate and revert changes if simulation time increases — Performance Advisor reverts previous co-simulation settings when the simulation time increases.
- Validate and revert changes if degree of accuracy is greater than tolerance — Performance Advisor reverts previous co-simulation settings if the degree of accuracy is greater than tolerance.

Tip You can use the `tic` and `toc` functions to measure the simulation time.

See Also

- `tic`
- `toc`
- `sim`

Identify co-simulation signals for numerical compensation

Identify co-simulation signals that may need explicit numerical compensation.

- Validate and revert changes if time of simulation increases — Performance Advisor reverts previous co-simulation settings the simulation time increases.
- Validate and revert changes if degree of accuracy is greater than tolerance — Performance Advisor reverts co-simulations if the degree of accuracy is greater than tolerance.

Tip You can use the `tic` and `toc` functions to measure the simulation time.

See Also

- `tic`
- `toc`
- `sim`

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
Maximum length of string data type	32,766 characters

Simulink Terminology

Simulink Terminology and Definitions

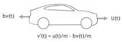
Systems and Models

system, physical system – Physical object or process with observable and measurable characteristics that change over time. For example, a vehicle is a *system* with multiple *system components*

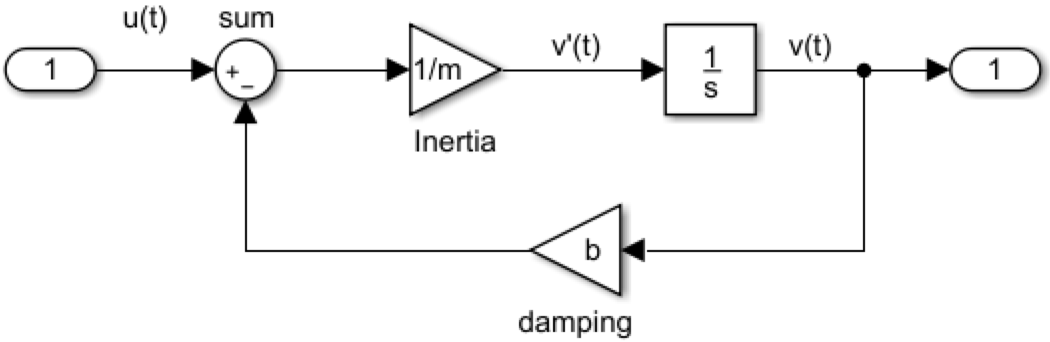
system component – Physical or functional part of a *system* that interacts with the other parts. The interactions define the structure and behavior of the *system*. For example, a cruise control module is a *system component* in a vehicle *system*.

Model, System Model, Dynamic System Model – Mathematical description of a *system* derived either from physical laws or experimental data. The description typically uses differential and difference equations.

In the following example for a vehicle, u(t) is the force (N) moving the vehicle forward, v(t) is the velocity (m/s), b is a damping coefficient (N·s/m), and m the mass of the vehicle (kg).

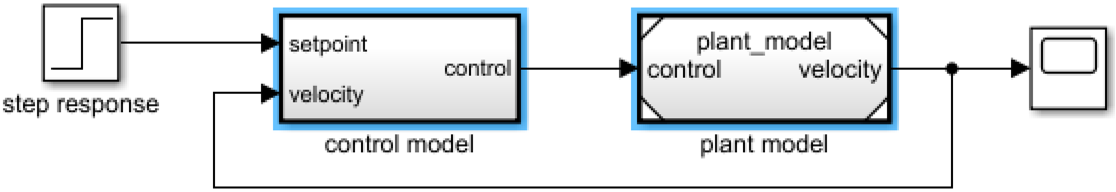


Simulink uses block diagrams to visually implement the mathematics of a *model*.



model component – Part of a *model* that interacts with the other parts through an interface of inputs and outputs.

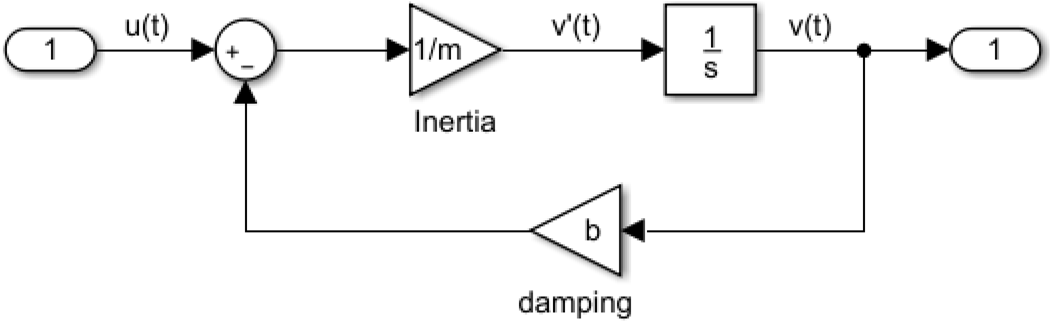
Simulink implements *model components* using Subsystem blocks and Model blocks that refer to other Simulink models. In the following example, **control model** is a Subsystem block and **plant model** is a Model block that references the file `plant_model.slx`.



differential equation (DE) – Mathematical equation containing derivatives of a variable. For continuous systems, differential equations describe the rate of change for variables with the equations defined for all values of time. For example, the velocity of a vehicle `v(t)` is defined with the following first order differential equation.

```
mv'(t) + bv(t) = u(t)
```

`u(t)` is the force moving the vehicle, `m` is mass, `b` is a damping coefficient, and the derivative `v'(t)` is acceleration. The following model implements the differential equation.

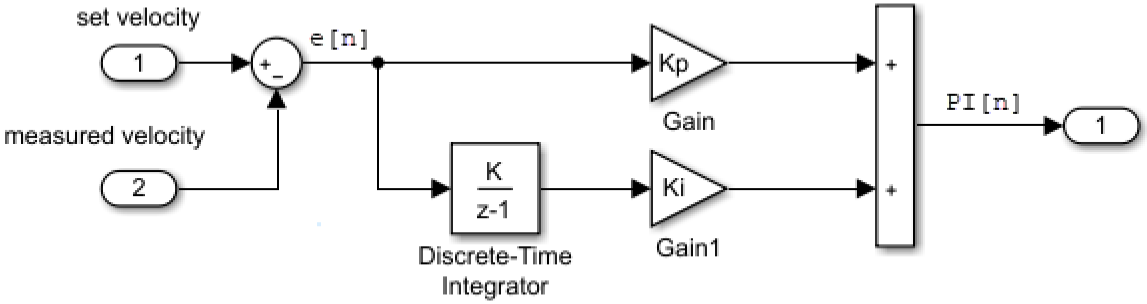


difference equation – Mathematical equation containing terms with present and past values. For discrete systems, difference equations describe the rate of change for variables defined only at specific times. For example, the control signal for a simple discrete PI (proportional–integral) controller is defined with the following difference equation.

```
PI[n] = e[n]Kp + (e[n]+e[n-1])Ki
```

`e[n]` is the error between the control signal and the set point value, `Kp` is the proportion constant while `Ki` is the integration constant, and `n` is the time step.

The following Simulink model implements the difference equation.



algebraic equation – Mathematical equation that is the sum of powers for one or more variables.

differential algebraic equations (DAE) – Set of differential and algebraic equations with a least one dependent variable that is not defined with a differential equation. Typically, an algebraic equation defines a variable constraint in the system model.

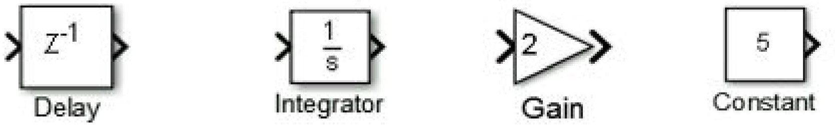
$$A' = -0.04A + 1 \cdot 10^4 BC$$

$$B' = 0.04A - 1 \cdot 10^4 BC - 3 \cdot 10^7 B^2$$

$$C = 1 - A - B$$

Simulink Models

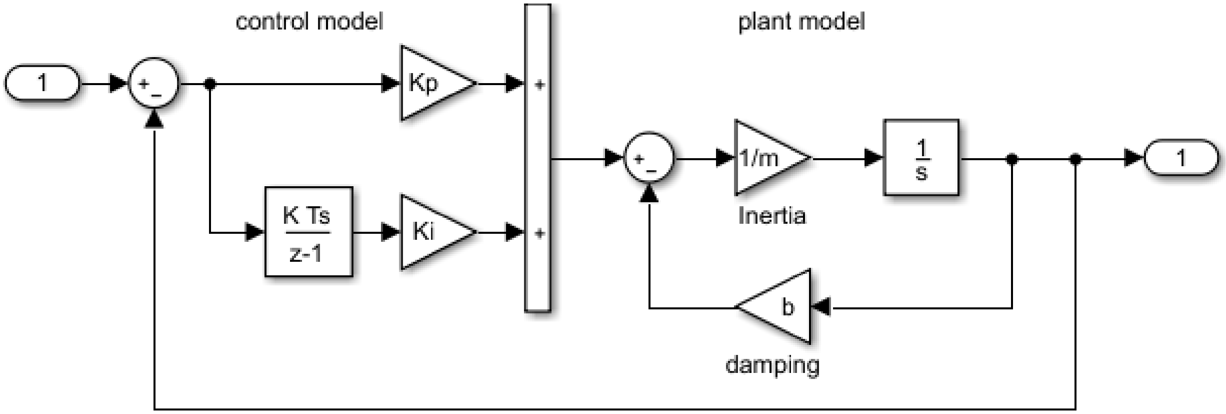
block, **model block**, **library block** – Basic modeling construct of Simulink. Some blocks have input signals, output signals, and state. Most blocks have parameters that specify block behavior. A library block is a block prototype while a model block is an instance of the library block.



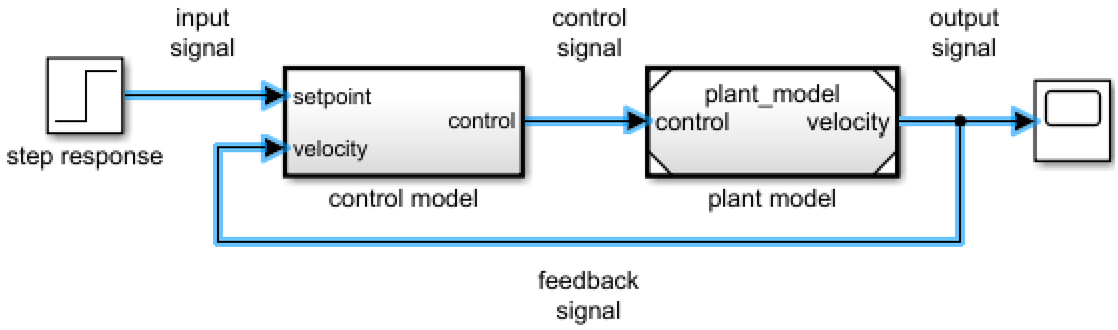
Each block represents a system of equations for the Simulink engine. The blocks show above have the following block equations.

$$\begin{cases} x[n+1] = u[n] \\ y[n] = x[n] \end{cases} \quad \begin{cases} \dfrac{dx}{dt} = u(t) \\ y(t) = x(t) \end{cases} \quad y(t) = 2 * x(t) \quad y(t) = 5$$

block diagram – Visual representation of a dynamic system model. Blocks are the variables and coefficients from the model equations while lines are the relationships between blocks.

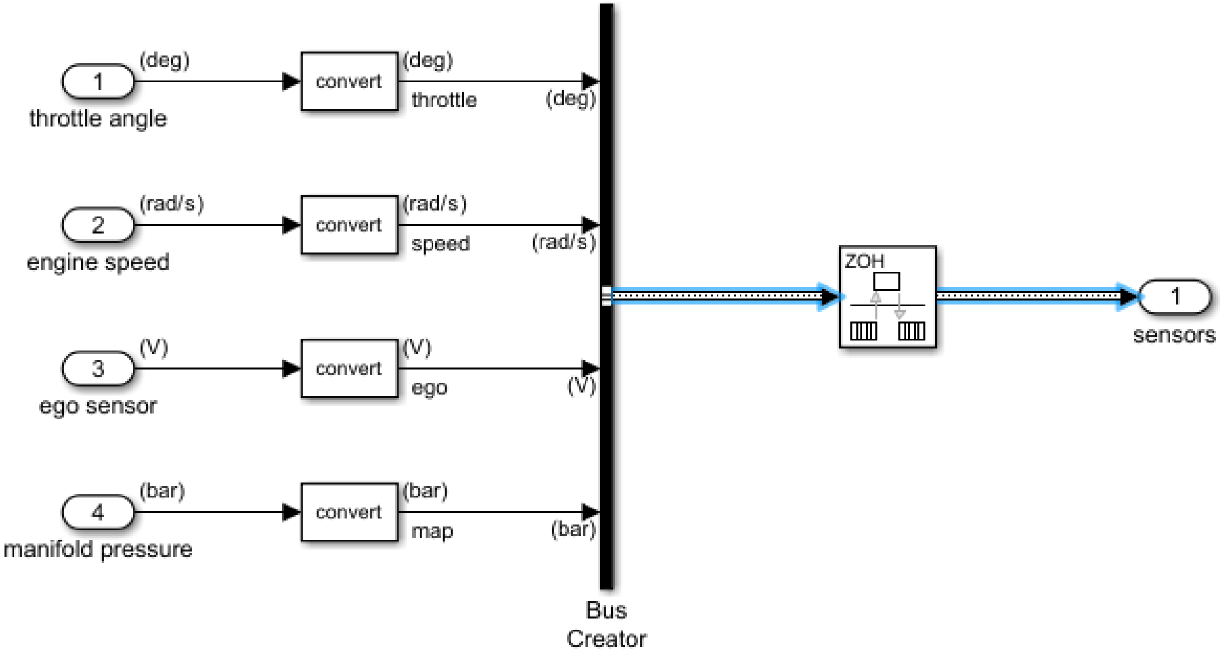


signal – Calculated output from a simulation that transfers data from one block to another block. Signals appear in a model as lines that connect the outputs of blocks to the inputs of other blocks. You can specify signal attributes, including signal name, data type, and dimension.



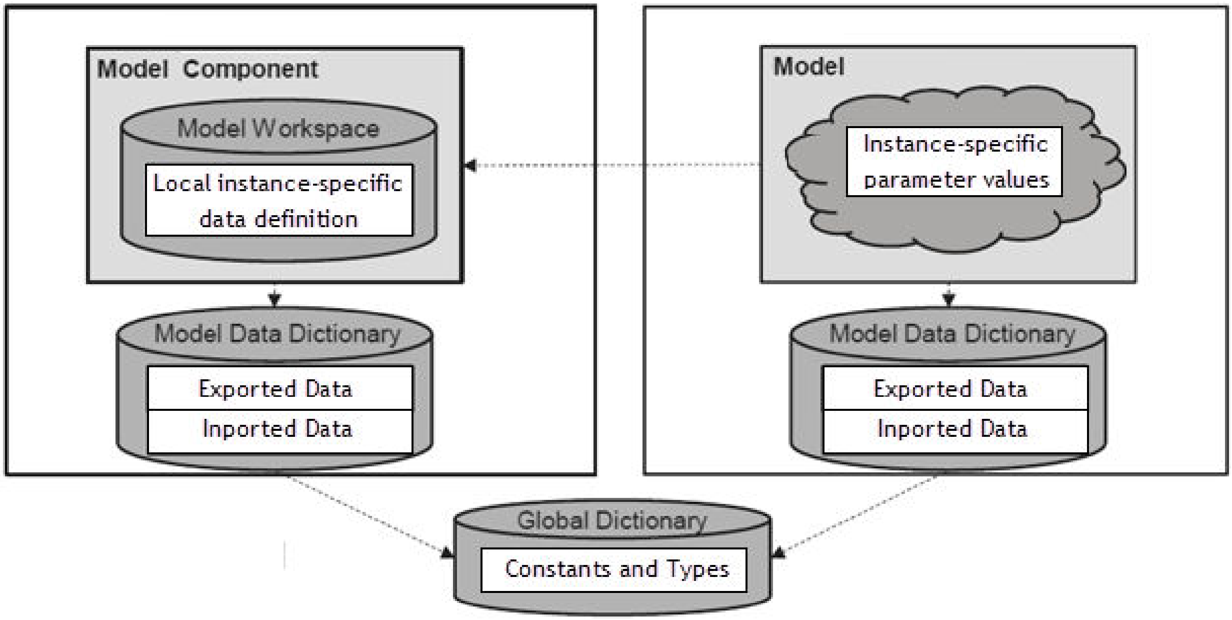
bus, **composit signal** – Group of signals used to manager and reduce visual complexity in a block diagram. Access the bus as a whole or select specific signals from the bus.

In the following model, individual sensor signals are combined into one bus signal.



data – Values for signals, block states, parameters, and other data a model needs to simulate and produce outputs from a simulation.

Simulink organizes model data into categories such as root-level I/O, global parameters, local parameters, and exported data.



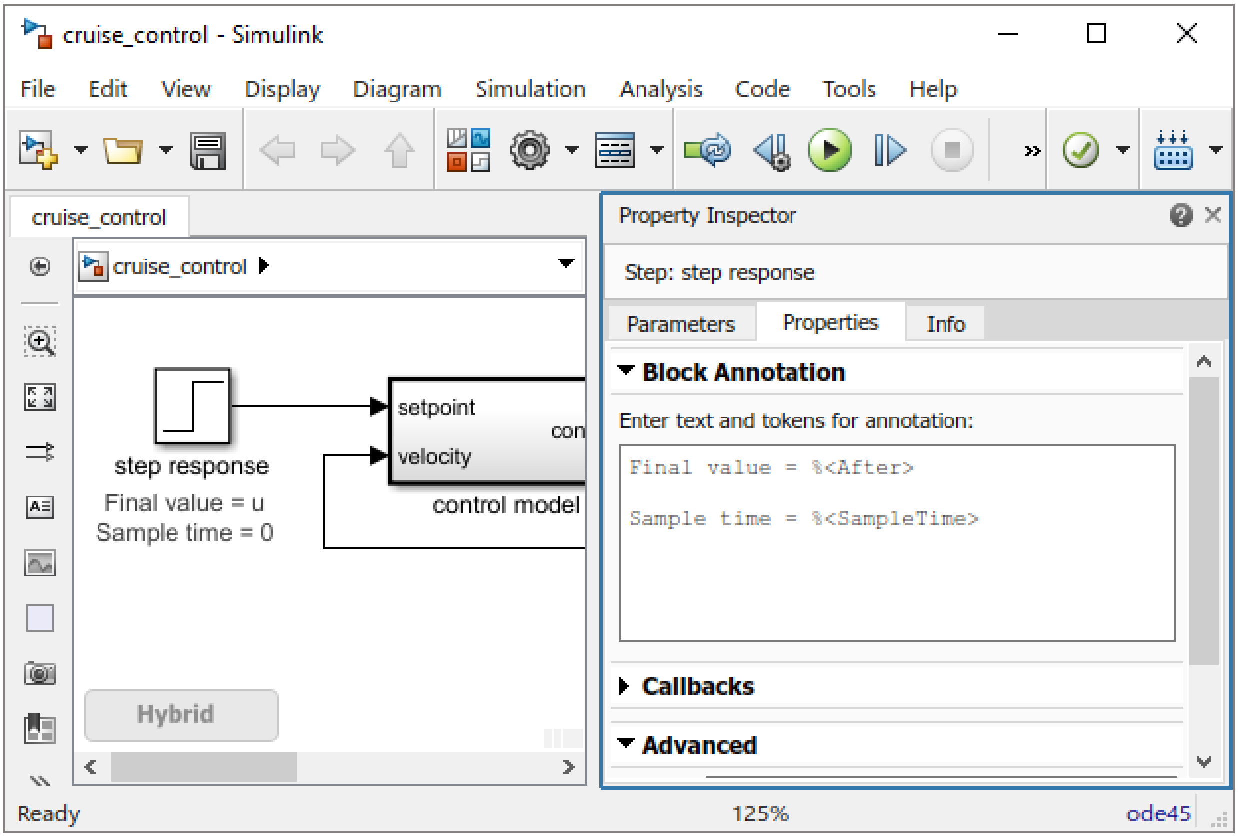
parameter – Name and value of a Simulink model characteristic:

- Model parameters – Control model behavior during compilation, simulation, and code generation. To set model parameters using the Configuration Parameters dialog box, select **Simulation > Model Configuration Parameters**.
- Block parameters – Define model dynamics and mathematics. Set block parameters using the Block Parameters dialog box or the Property Inspector.

property – Model and block characteristics that include:

- Block Annotation – Values of selected block parameters displayed below the block.
- Callbacks – Commands that execute when a specific model or block event occurs.
- Tags – Block identifiers that are programmatically searchable .

To set properties, select **View > Property Inspection**, and then select the **Properties** tab.



Step: step response

Enter text and tokens for annotation:

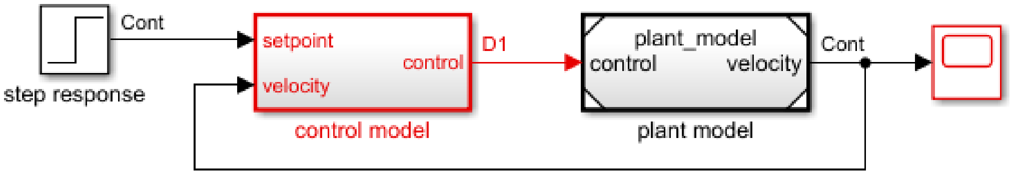
```
Final value = %<After>

Sample time = %<SampleTime>
```

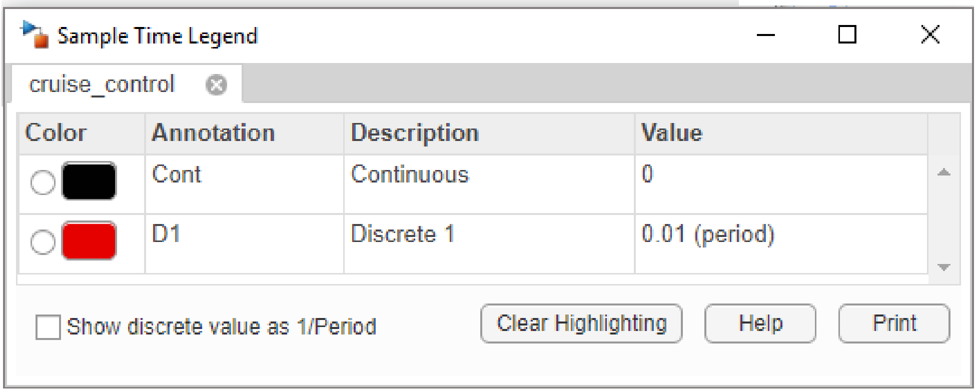
Final value = u
Sample time = 0

sample time – Time interval that defined the rate (1 / sample time) for calling the block methods to produce outputs and update internal state. Sample time is specified as:

- `-1` – Simulink determines the sample time.
- `0` – Continuous rate where blocks run at variable times based on solver settings. Blocks with continuous state, always have a sample time of `0`.
- Positive non-zero number – Discrete rate where blocks run at times that are multiples of their sample time.

In the following model, the controller runs at a discrete rate of `0.01` seconds while the plant was determined by Simulink to have a continuous sample time.



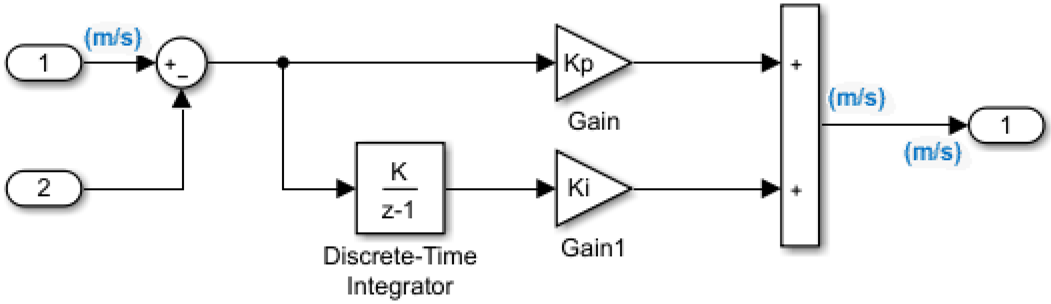


Color	Annotation	Description	Value
○ ⬛	Cont	Continuous	0
○ 🟥	D1	Discrete 1	0.01 (period)

□ Show discrete value as 1/Period

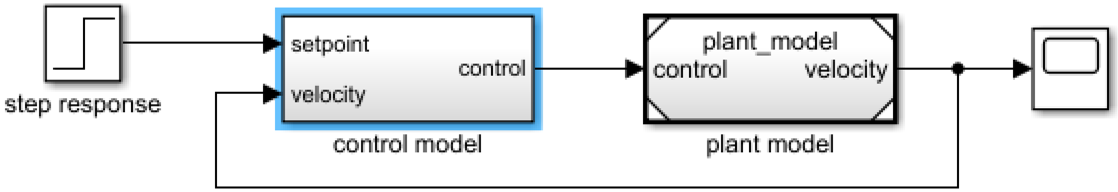
unit – Agreed upon amount of a quantity used to measure the total amount of a quantity.

Simulink *units* are specified as a Inport block or Outport block parameter at the boundaries of a Simulink *model component*. Simulink *model components* include Subsystem blocks, Model blocks, Stateflow charts, and Simulink to Simscape converter blocks. To display units on a model, select **Display** > **Signals & Ports** > **Port Units**.



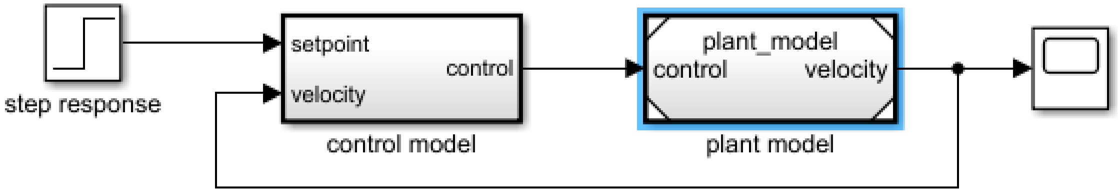
Subsystem block – Simulink block that groups together blocks that execute together as a unit (atomic subsystem) or execute separately within a *sorted order* (virtual subsystem).

In the following model, the control model is an atomic Subsystem block containing blocks that model a PI controller.



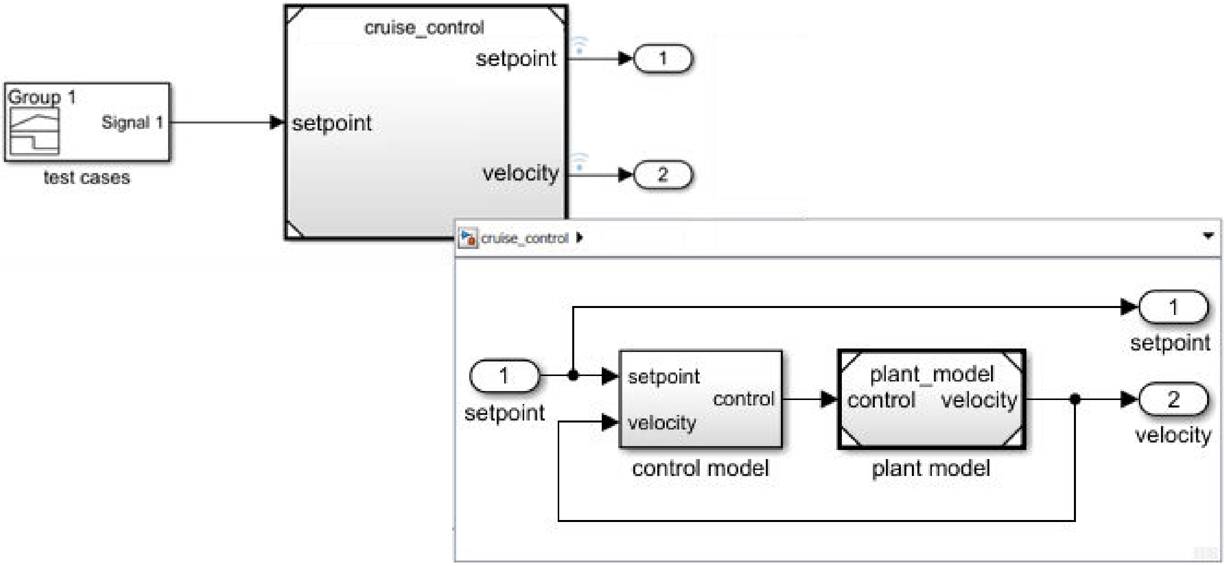
Model block, referenced model -- Child model included in a parent model using a Model block. For Simulation, blocks within a referenced model execute together as a unit.

In the following example, the plant model is saved in the file `plant_model.slx`, and then it is added to the `cruise_control` model using a Model block.



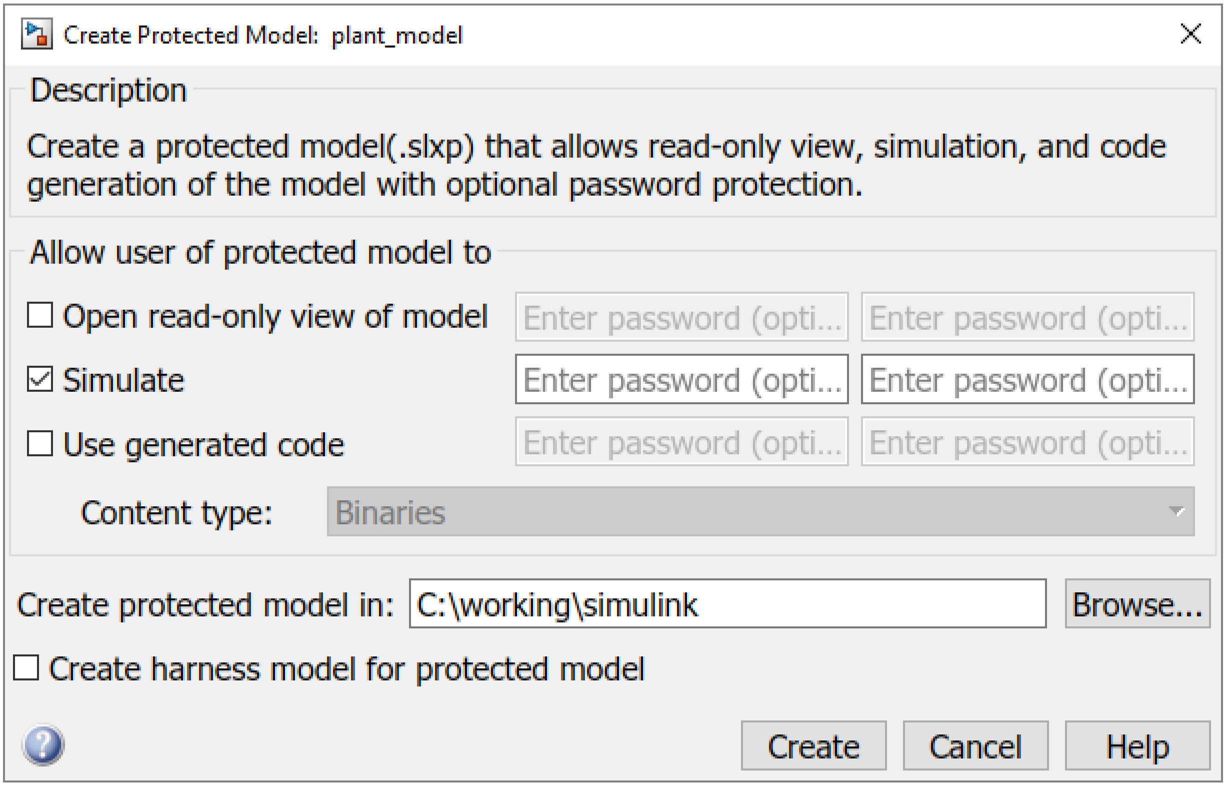
test-harness, test model – Simulink model that provides a framework for testing the simulation behavior and outputs of another model or model component. Objectives of the test-harness are to automate the testing process, execute a test suite with multiple test cases, and save results.

In the following model, a Signal Builder block provides test signals while outputs are logged for analysis in Simulation Data Inspector or MATLAB.



protected model - Simulink referenced model that cannot be edited and possibly hides the block diagram. A protected model provides the ability to deliver a model for simulation and code generation in a larger system model without revealing intellectual property.

To protect a referenced model, right-click a Model block, select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**. Select the allowed model actions with optional passwords.



Create Protected Model: plant_model

Description

Create a protected model(.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.

Allow user of protected model to

□ Open read-only view of model	Enter password (opti...	Enter password (opti...
☑ Simulate	Enter password (opti...	Enter password (opti...
□ Use generated code	Enter password (opti...	Enter password (opti...

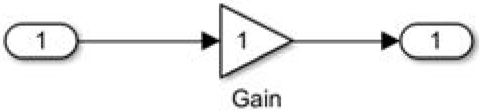
Content type: Binaries

Create protected model in: C:\working\simulink

□ Create harness model for protected model

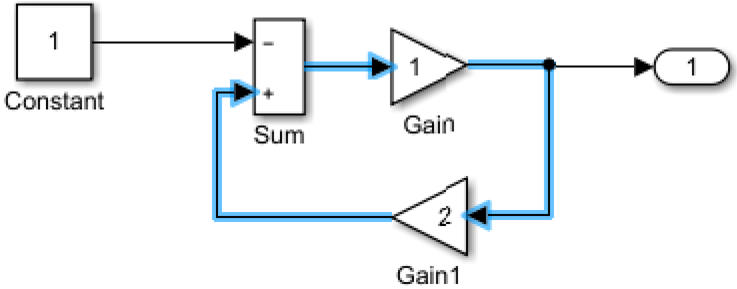
direct feedthrough – Simulink block characteristic where the output port signal of a block is computed from the values of its input port signals. The output signal value is a function of the input signal values.

Blocks with direct feedthrough include the Gain, Product, Sum, Transfer Fcn, State-Space, and Math Function blocks.

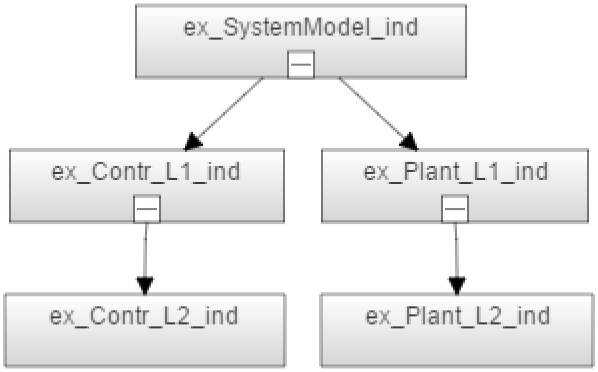


algebraic loop – *Signal* loop between blocks with *direct feedthrough*. An algebraic loop generally occurs when an input port of a block with *direct feedthrough* is driven directly by the output port of the same block or indirectly through other blocks with *direct feedthrough*.

In the following model, the two Gain blocks with *direct feedthrough* create an *algebraic loop*.



model dependencies – Files a Simulink model needs to run a simulation. Files include referenced models, data files, and S-function files. Use the *Model Dependency Viewer* to identify dependent model files.



Simulink Tools

Simulink Processes

compilation – Simulink process where the block diagram is translated to an internal representation that interacts with the Simulink engine. The model-level set of equations are composed from the block-level equations.

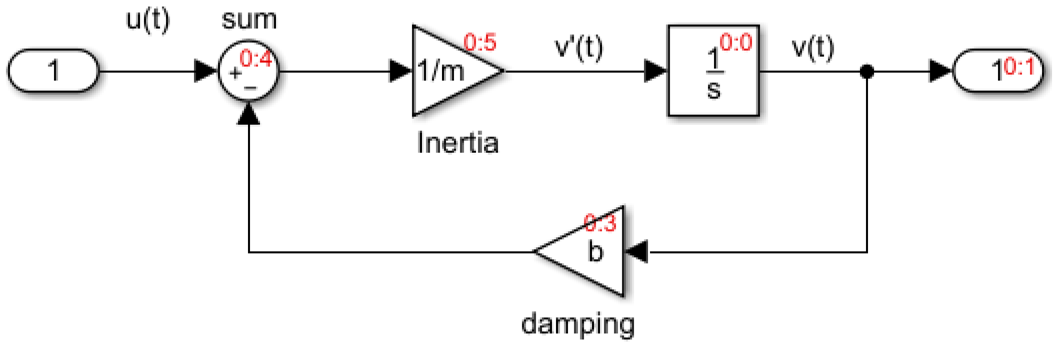
$$\begin{cases} X = [x_c, x_d] & \text{Model states} \\ \dot{x}_c = f_c(X, U, t) & \text{Derivative method} \\ x_d[n+1] = f_d(X, U, t) & \text{Update method} \\ Y = [y_1, y_2] & \\ Y = g(X, U, t) & \text{Output method} \end{cases}$$

$$X(t_0) = X_0 \qquad \text{Initial model states}$$

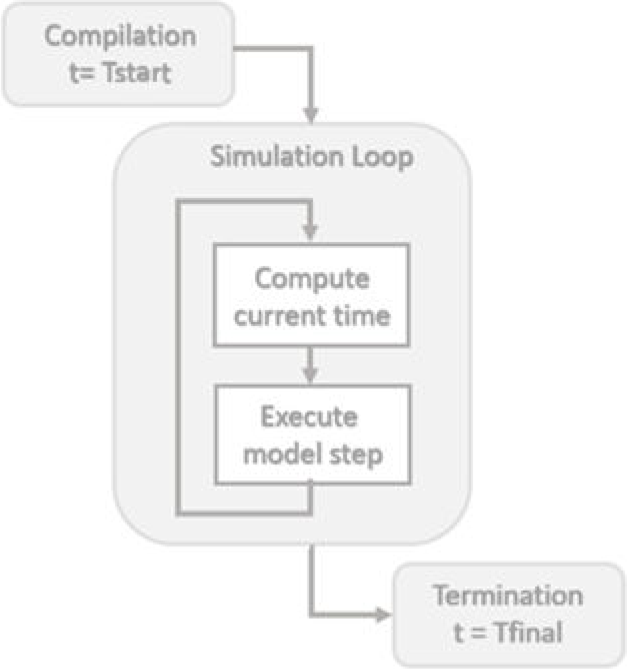
There are no model-level equations that contain terms for the blocks. Instead, model-level equations correspond to calling the methods on each block individually and in a specific order. This is equivalent to having a model-level system of equations.

sorted order – Order in which block output methods are called after evaluating direct feedthrough of each input port. To display sorted order, select **Display** > **Blocks** > **Sorted Execution Order**.

In the following model, the Integrator block output runs first, and then the loop of blocks connected to the Integrator block input.

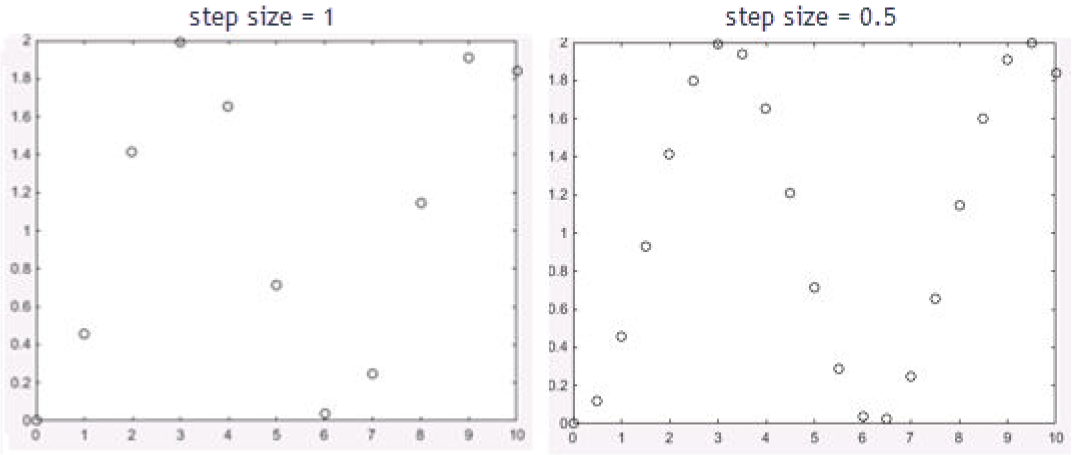


simulation – Simulink process after model *compilation* where block outputs and states are computed at successive time steps over a specified time range. During each simulation loop, Simulink calculates a `Δt` to determine the current time step `t(k+1) = t(k) + Δt`. At the end of a simulation, results are given as vectors `[t, X, Y]` for time, state and output at each time step.



solver, **numerical solver** – Algorithm that interprets the graphical model for a dynamic system to construct the underlying equations, and then solves the equations numerically. Types of solver:

- Fixed step – Time step `T(k+1) = T(k) + Δt` where `Δt` is constant. If step size is too large, simulation results can have a large error.



- Variable step – Time step `T(k+1) = T(k) + t`Δ_k were `t`Δ_k changes from one simulation step to the next. Variable step solvers iterate to reach a solution based on an error tolerance.

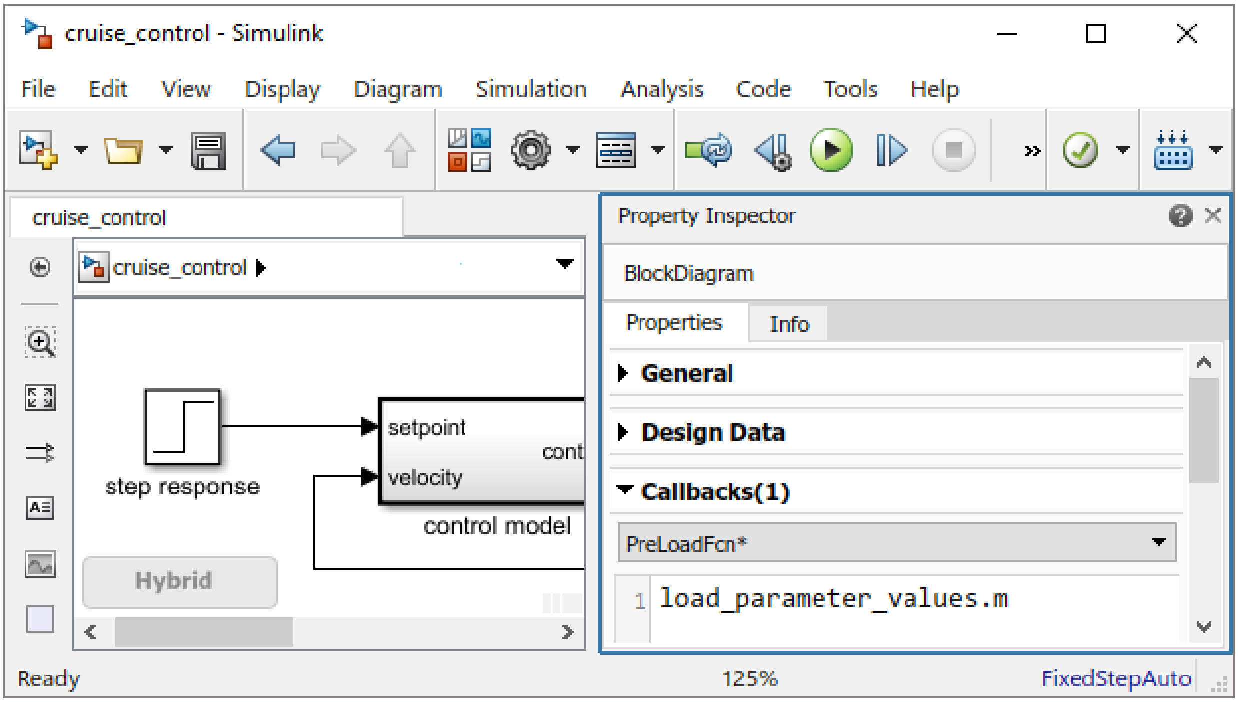
Programming Constructs in Simulink

Simulink command – MATLAB command that is specific to Simulinkmodeling or simulation. Enter Simulink commands in the MATLAB Command Window or use in MATLAB scripts for testing a model programmatically. The following commands set simulation parameters, run a simulation, and saves the simulation results. In the first statement, `sim` is a *Simulink command* for running a simulation.

```
simOut = sim('cruise_control','SimulationMode','normal',...
             'AbsTol','1e-5','SaveState','on',...
             'StateSaveName','xout','SaveOutput','on',...
             'OutputSaveName','yout','SaveFormat', 'Dataset');
outputs = simOut.get('yout')
```

callback, **model callback** – MATLAB® code that executes in response to a specific model or block action. To add a model callback, select **View** > **Property Inspector**, select the **Properties** tab, select a function from the **Callbacks** list, and then enter MATLAB code or the name of a MATALB script.

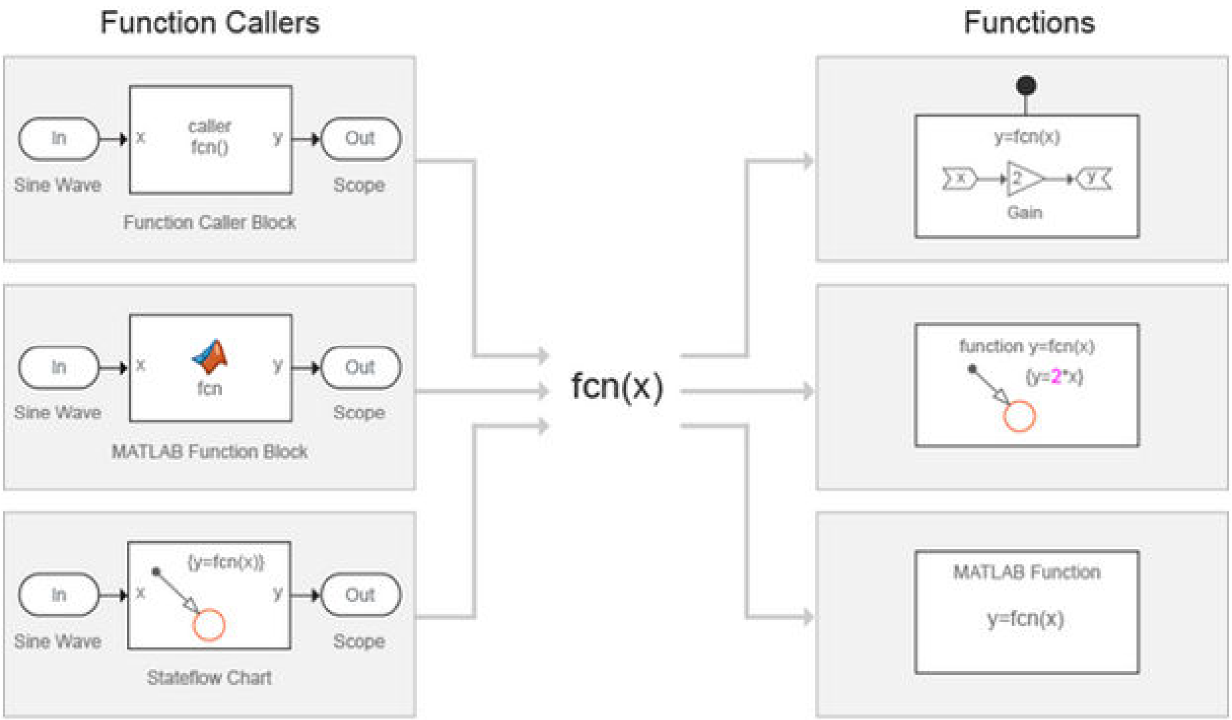
In the following example, when Simulink loads a model it also loads a file into the MATLAB workspace with parameters values for the model.



PreLoadFcn*

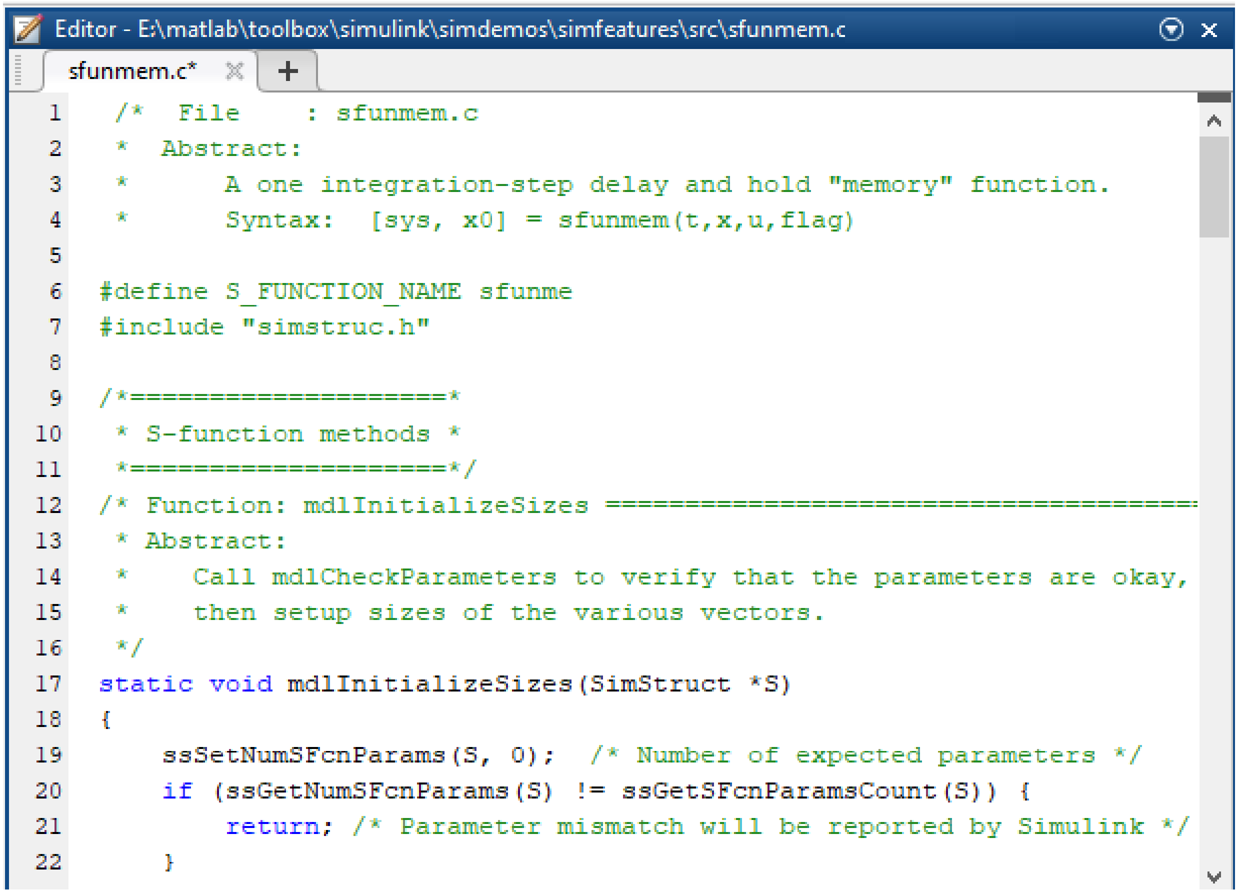
```
load_parameter_values.m
```

Simulink function – Computational unit that calculates a set of outputs when provided with a set of inputs. A common text interface between function caller and function definition allows various definition formats using a Simulink Function block, exported Stateflow graphical function, or exported Stateflow MATLAB function.



MATLAB Function –

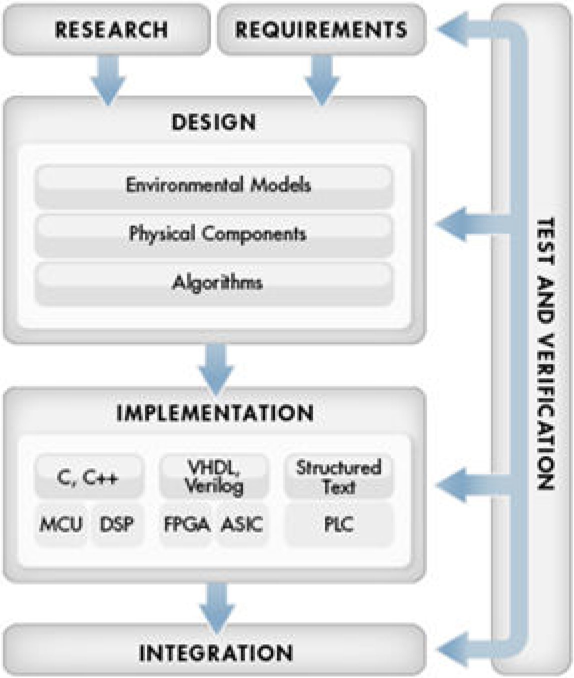
S-function, **system-function** – Computer language description of a Simulink S-Function block written in MATLAB code, C, C++, or Fortran. C, C++, and Fortran S-functions are compiled as MEX files using the MATLAB `mex` utility.


Editor - E:\matlab\toolbox\simulink\simdemos\simfeatures\src\sfunmem.c
sfunmem.c*
+

```
 /*  File    : sfunmem.c
  *  Abstract:
  *      A one integration-step delay and hold "memory" function.
  *      Syntax:  [sys, x0] = sfunmem(t,x,u,flag)

#define S_FUNCTION_NAME sfunme
#include "simstruc.h"

/*====================*
 * S-function methods *
 *====================*/
/* Function: mdlInitializeSizes =====================================
 * Abstract:
 *    Call mdlCheckParameters to verify that the parameters are okay,
 *    then setup sizes of the various vectors.
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }
```

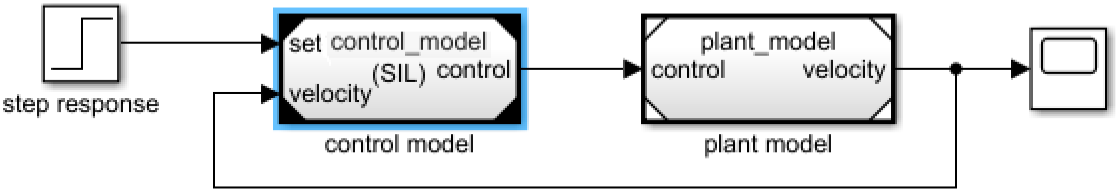
Model Development Processes

model-based design – Development process that uses a system model as an executable specification throughout development. The process supports model and model component design, model simulation of dynamic behavior, code generation from the model, and continuous test and verification.



software-in-the-loop (SIL) simulation – Development process where compiled source code on a development computer executes as a separate process from the rest of the Simulink model. Typical goals include initial source code testing and verification by comparing software-in-the-loop results with model simulation results or system requirements.

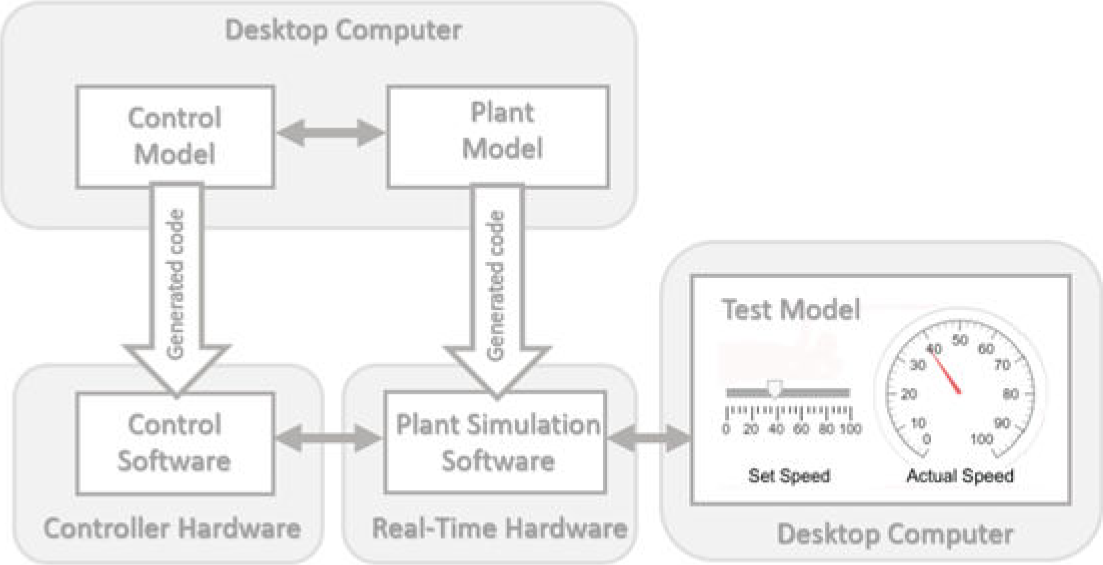
To run a software-in-the-loop simulation, right-click a Model block and select **Block parameters**. From the **Simulation mode** list, select `Software-in-the-loop (SIL)`.



processor-in-the-loop (PIL) simulation – Development process that cross-compiles source code and runs the resulting object code on a target processor using hardware-specific data attributes and sample time attributes. Typical goals include object code verification by comparing processor-in-the-loop results with model simulation or *software-in-the-loop (SIL)* results. Also, collect execution time profiling data.

hardware-in-the-loop (HIL) simulation – Simulation process that pairs physical components, such as the hardware and software for a controller, with a virtual real-time implementation of a physical component, such as a plant.

In the following example, code for the controller model is generated and downloaded to the production controller hardware. Code is generated for the plant model and downloaded to a real-time hardware computer (e.g. Simulink Realtime).



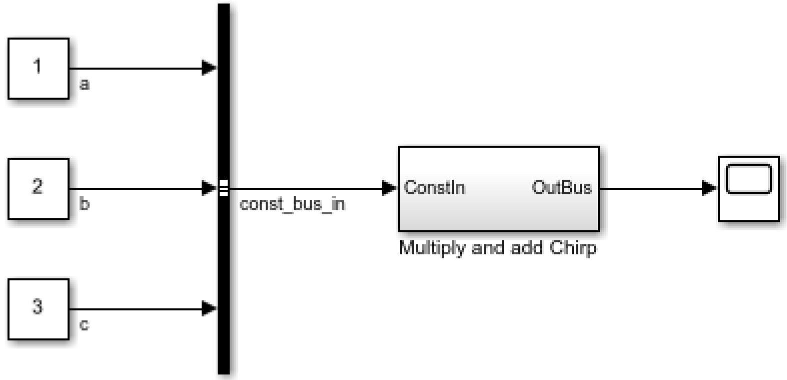
Block Reference Page Examples

Create Bus Ports in a Subsystem

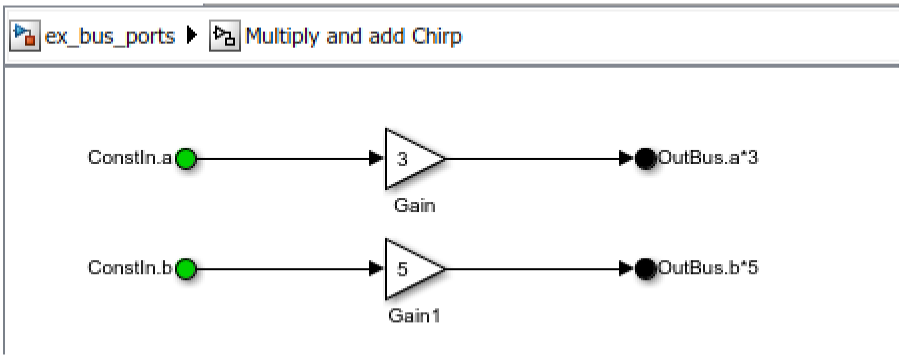
This example shows adding In Bus Element and Out Bus Element blocks to create bus element ports in a subsystem for selecting signals from an input bus and creating an output bus signal.

Model Structure

Open the model. The top model has three constant signals combined into a bus that feeds a subsystem input port and outputs a bus signal to a Scope block.

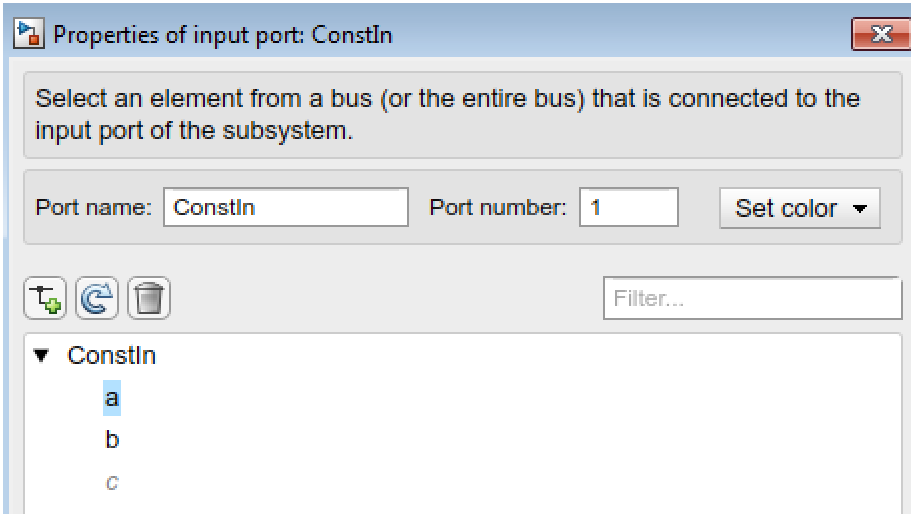


The subsystem includes two In Bus Element and Out Bus Element blocks.



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 port name from the default InBus. This block selects the `a` signal. The block color is set to green instead of the default black.



Properties of input port: ConstIn

Select an element from a bus (or the entire bus) that is connected to the input port of the subsystem.

Port name: ConstIn   Port number: 1   Set color ▾

Filter...

▼ ConstIn
- a
- b
- c

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

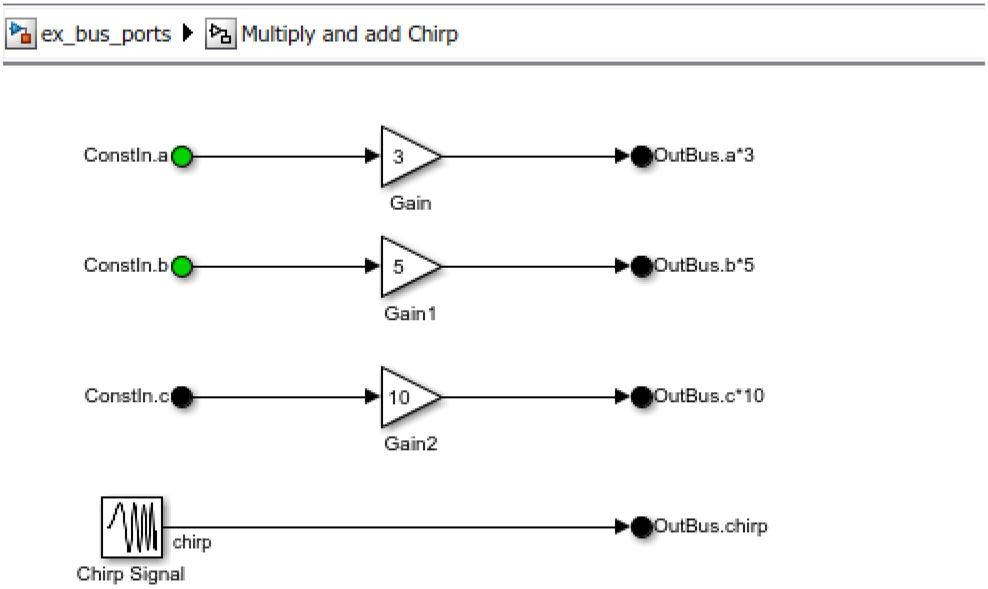
Create a third In Bus Element block for the `c` signal. In the Simulink Editor, right-click and drag the ConstIn.b block to make a copy of the block. Specify to use the existing port. Edit the block icon text to say `ConstIn.c`. Select the block and override the green block color by specifying black. 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.

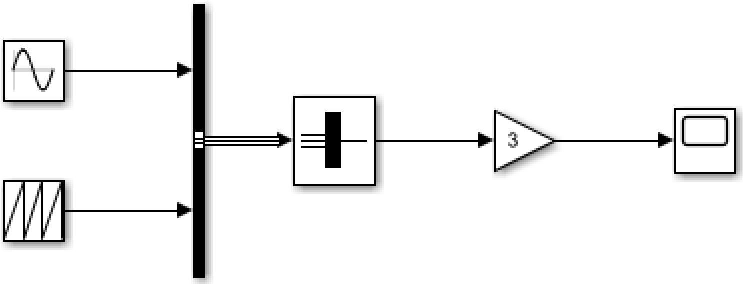


Convert Bus Signal to a Vector

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)
```

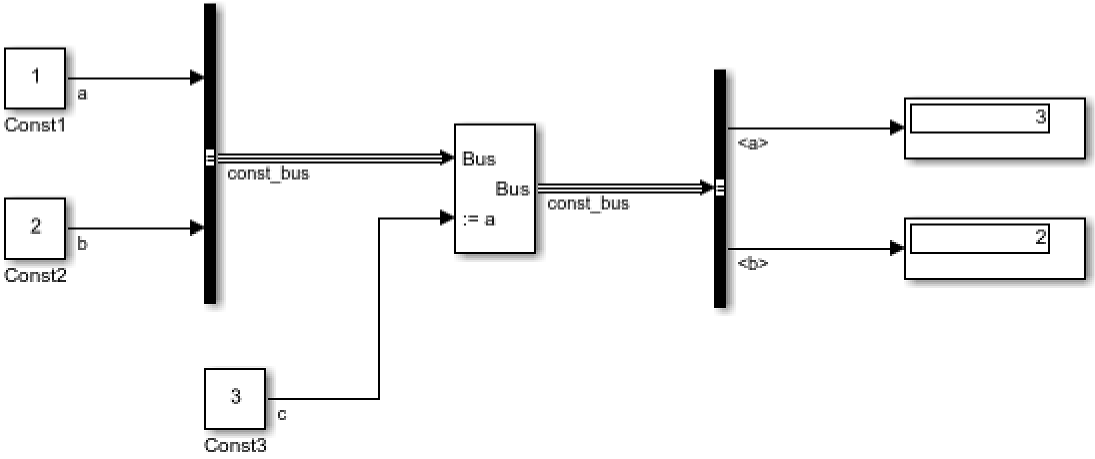


The Gain block cannot accept a bus signal. Inserting a Bus to Vector block provides the type of input signal (a vector) that the Gain block can accept. The Bus to Vector block has no user-accessible parameters.

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.



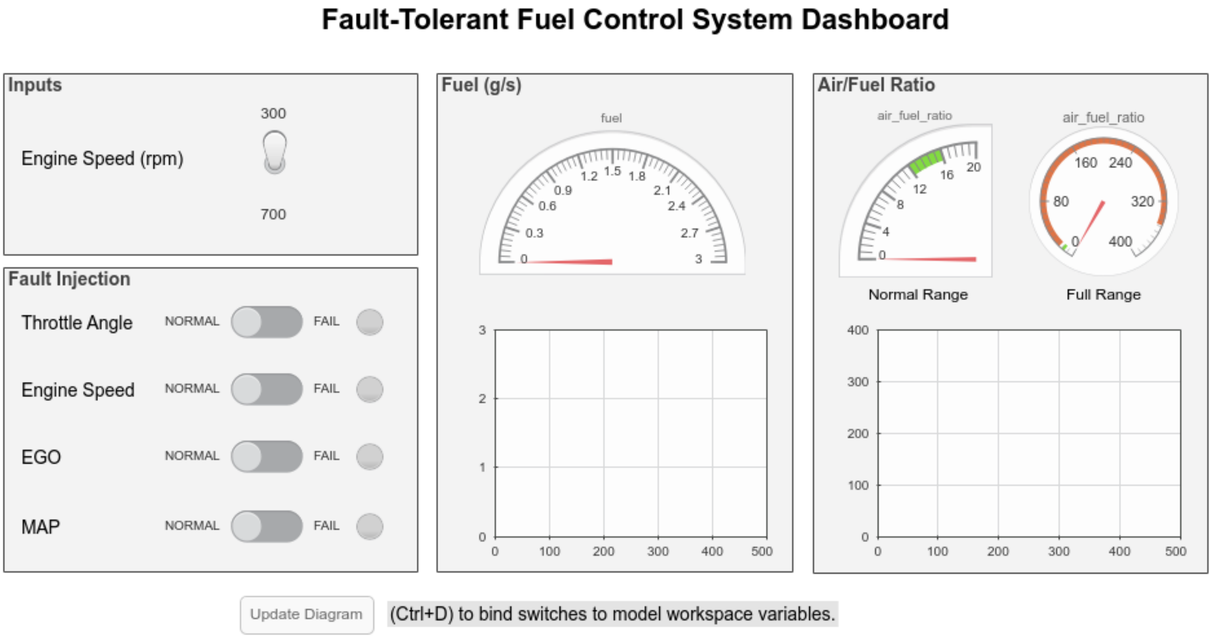
Initially, the value of signal a is 1. However, the Bus Assignment block replaces that initial value of signal a with the value of signal c, which is 3. The const_bus output signal has a value of 3 for signal a, as the Display block shows.

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.



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

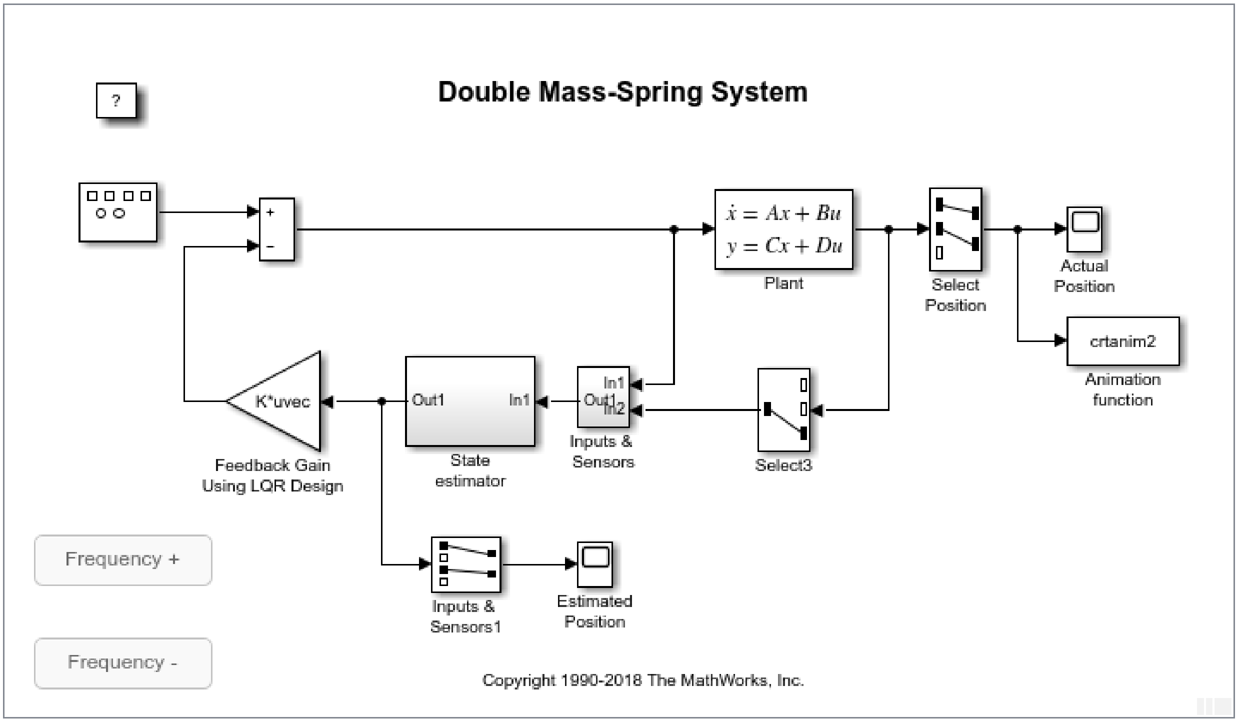
“Control a Parameter Value with Callback Button Blocks” on page 14-13

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

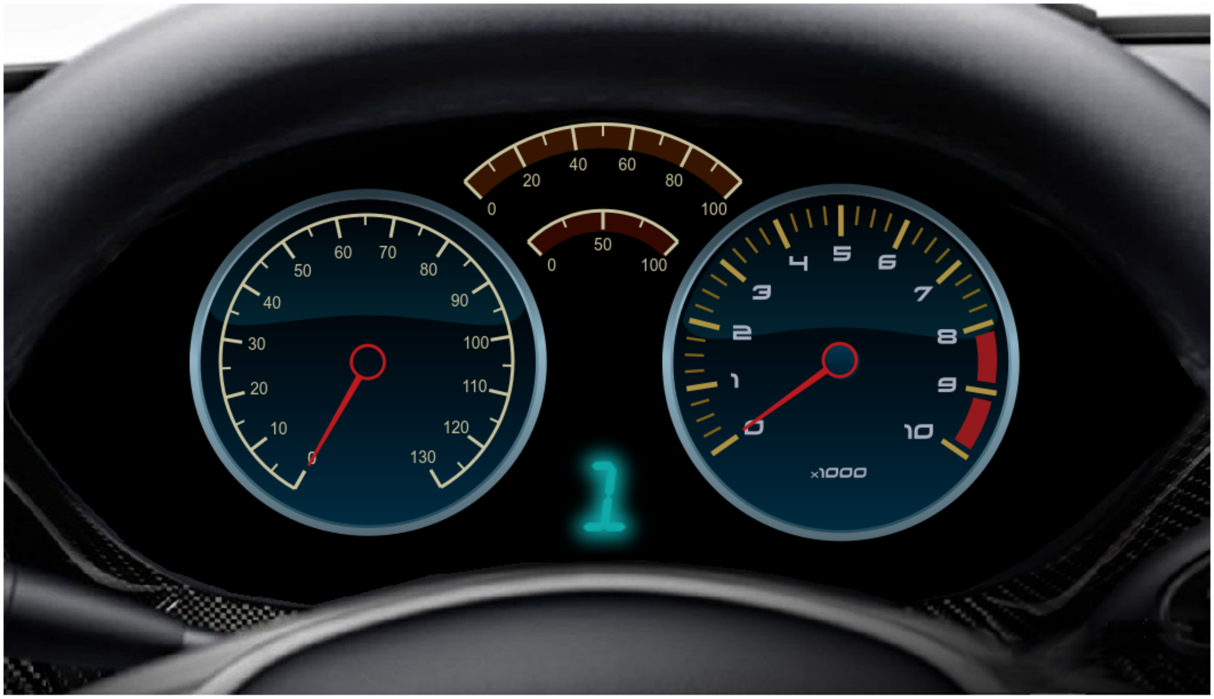
“Initialize Your Model Using the Callback Button Block” on page 14-11

Create a Realistic Dashboard Using the Custom Gauge Block

You can use the Custom Gauge block to create a dashboard of controls and indicators for your model that looks how it would in a real system.

Model Overview

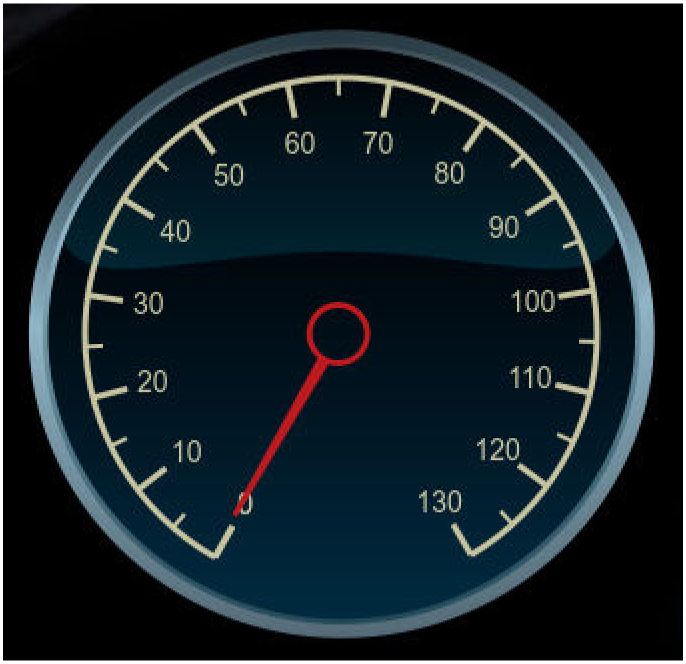
This example model uses four Custom Gauge blocks and a MultiStateImage block to create a dashboard for the `sf_car` model like one you might see in a real car.



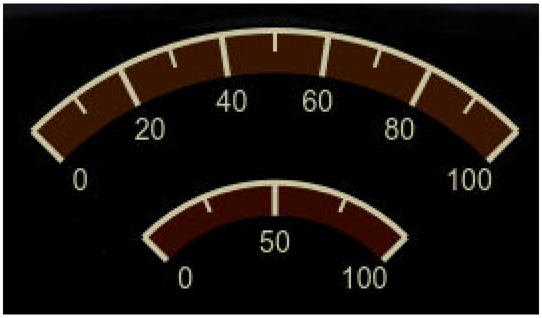
Explore the Model

To understand the connections between the Dashboard and model, you can select each block and jump to the signal it displays in the model. To jump to the connected signal, hover on the ellipsis that appears over the block when you select it. Then, click the arrow button.

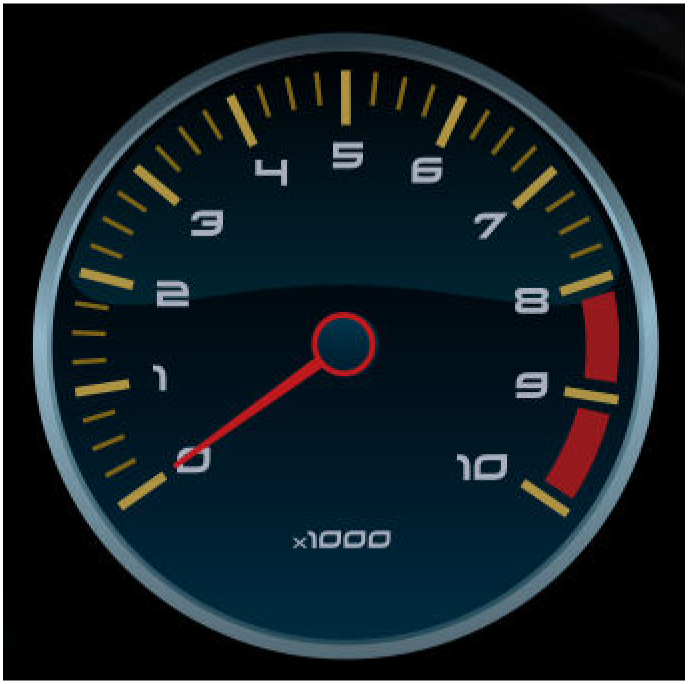
The Custom Gauge block on the left represents the `vehicle speed` signal, acting as a speedometer on the dashboard and using the default Custom Gauge appearance and a value arc you can see when you simulate the model.



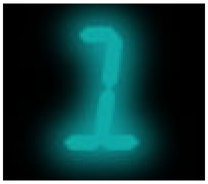
The Custom Gauge blocks at the top of the model display the `throttle` and `brake` signals. You can create this type of gauge by deleting the background image and needle image from the default Custom Gauge. Then, draw an arc and select colors for the tick marks, arc, and value arc.



The Custom Gauge block on the right displays the `engine RPM` signal on a custom gauge face. You can create a gauge like this by deleting the background image in the default Custom Gauge and uploading your own.



The MultiStateImage block on the bottom displays the `gear` signal using seven segment display style number images that correspond to the value of the `gear` signal.



Observe System During Simulation

You can use the dashboard to monitor the system response during simulation. The User Inputs subsystem includes several simulation inputs to model different vehicle maneuvers. To change the simulation input, navigate to the top level of the model and double-click the User Inputs subsystem. You can use the drop-down menu at the top of the gui to choose the vehicle action for the model to simulate. Choose between a passing maneuver, gradual acceleration, hard braking, and coasting. To monitor the system response during simulation, navigate back to the Dashboard subsystem and then press the play button to simulate the model.

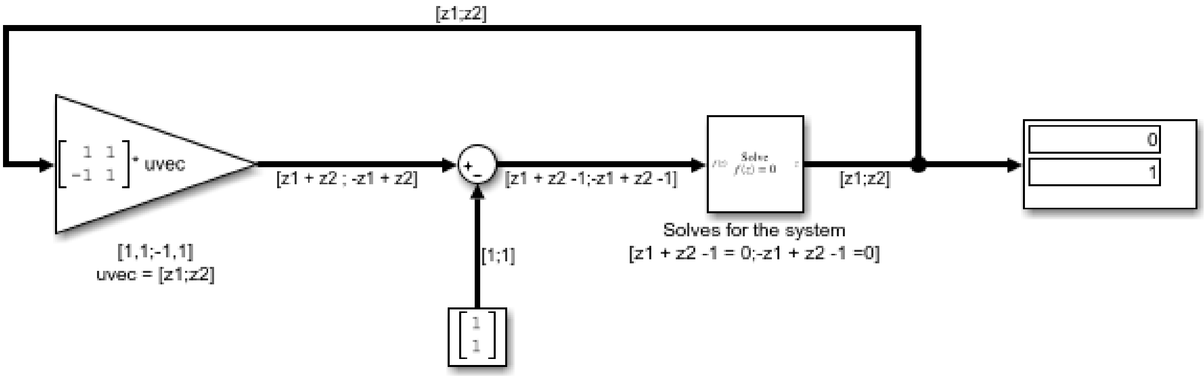
Solve a Linear System of Algebraic Equations

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

$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$



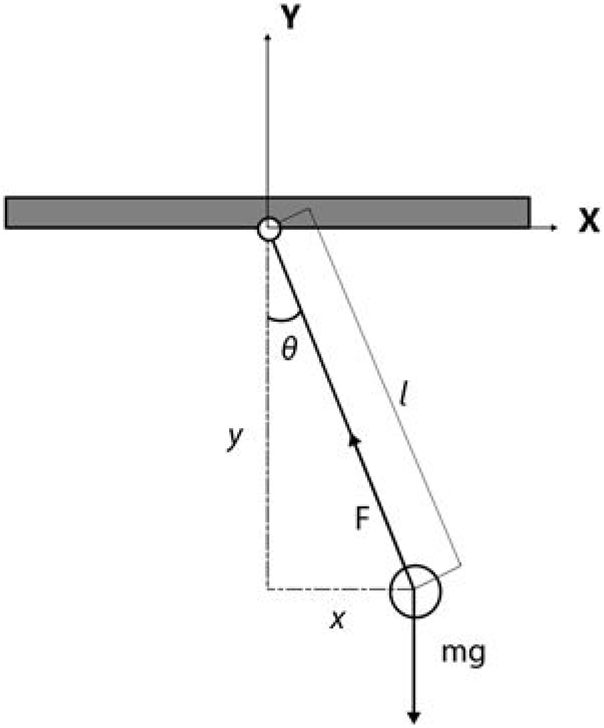
The signal fed to the Algebraic Constraint block $f(z)$ is a 2×1 vector of the form

$$\begin{bmatrix} z_1 + z_2 - 1 \\ -z - 1 + z_2 - 1 \end{bmatrix}$$

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 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.

$$\begin{aligned} m\ddot{x} &= F\sin\theta \qquad (1) \\ m\ddot{y} + F\cos\theta &= -mg \qquad (2) \end{aligned}$$

Let *(u, v)* be the velocities in *(x, y)* respectively. The system can be rewritten as a system of first order ODEs

$$
\begin{aligned}
\dot{x} &= u && (3)\\
\dot{u} &= -F\tfrac{x}{ml} && (4)\\
\dot{y} &= v && (5)\\
\dot{v} &= -F\tfrac{y}{ml} - g && (6)
\end{aligned}
$$

where *F* is the tension in the rod. The system also possesses the geometric constraint

$$x^2 + y^2 \quad = \quad l^2 \qquad (7)$$

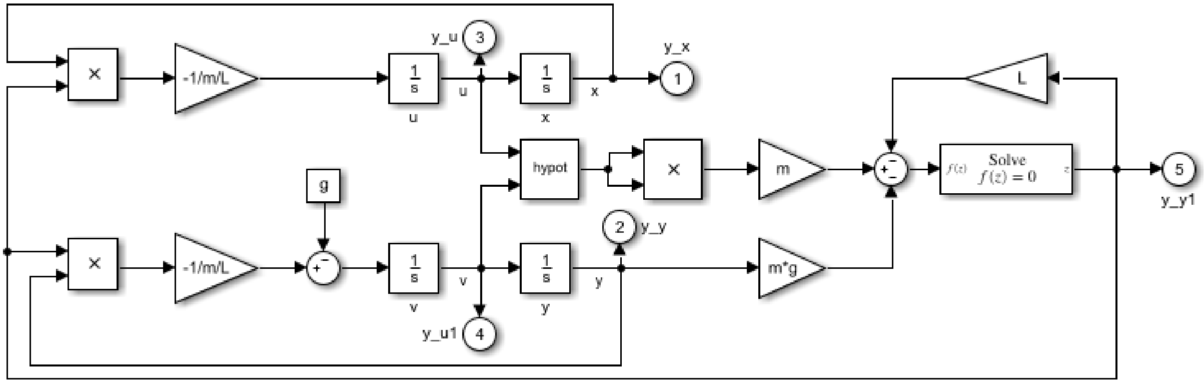
Differentiate (7) twice with respect to time *t* to arrive at

$$m(u^2 + v^2) - Fl - mgy \quad = \quad 0 \qquad (8)$$

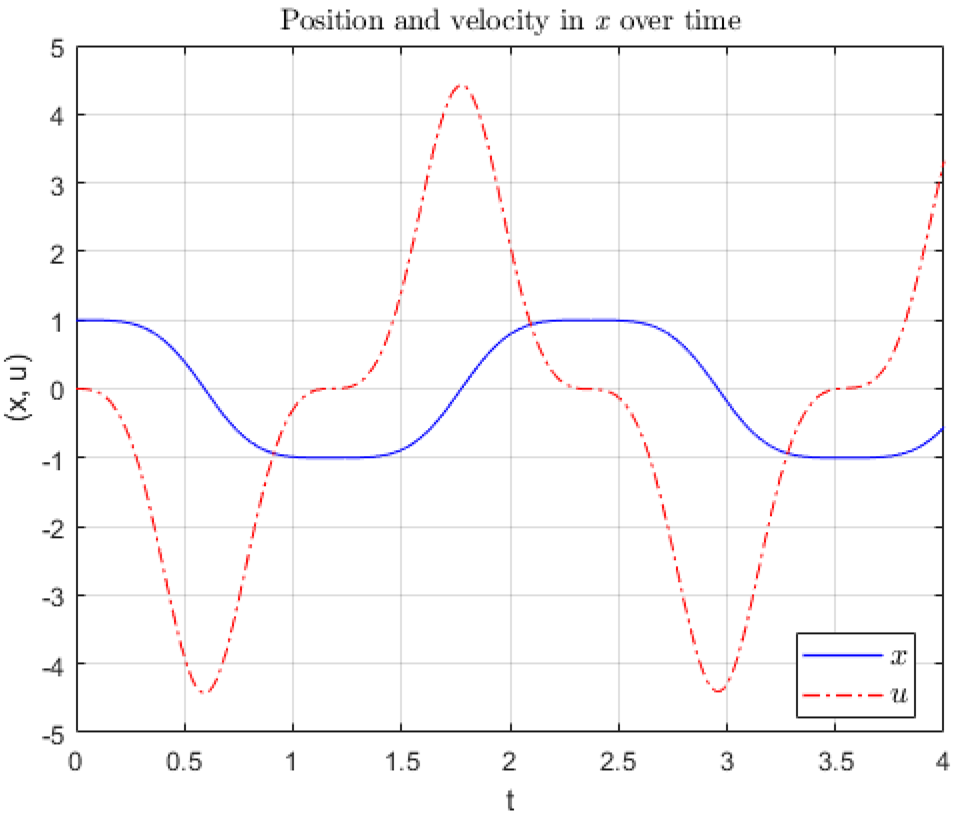
This relationship is useful since it allows *F* to determined at every step for use in modeling the kinematics of the system.

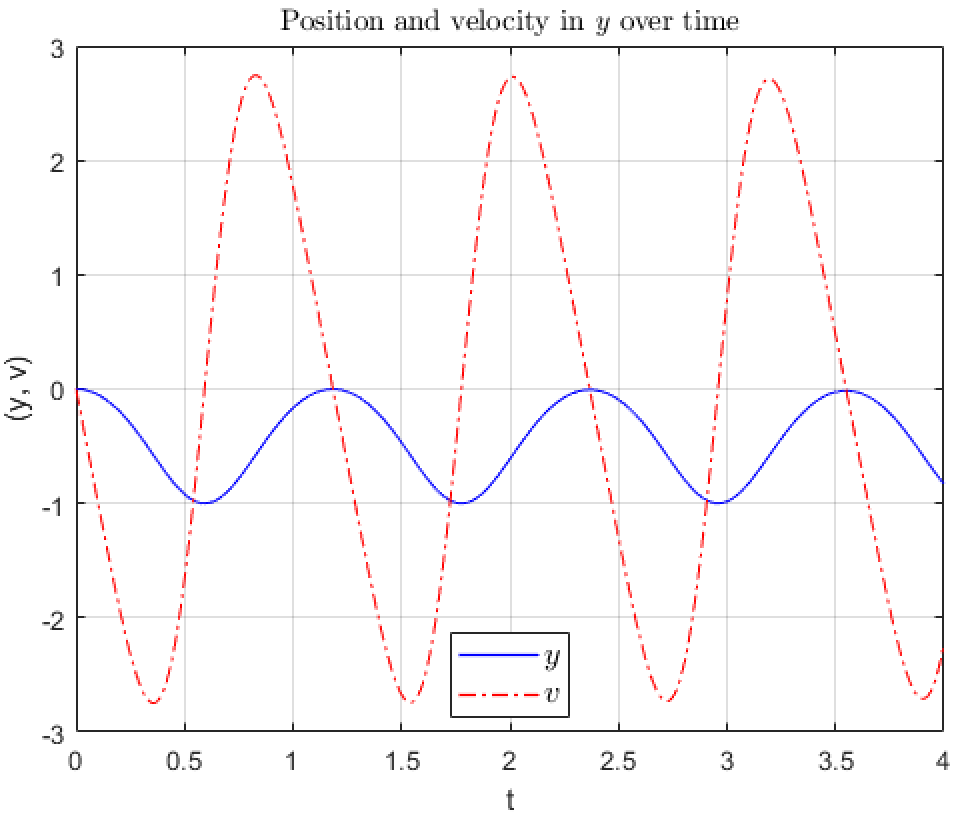
Simulating the System

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).


Position and velocity in x over time
5
4
3
2
1
0
-1
-2
-3
-4
-5
(x, u)
0
0.5
1
1.5
2
2.5
3
3.5
4
t
x
u



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.

Improved Linearization with Transfer Fcn Blocks

The Laplace domain transfer function for the operation of differentiation is:

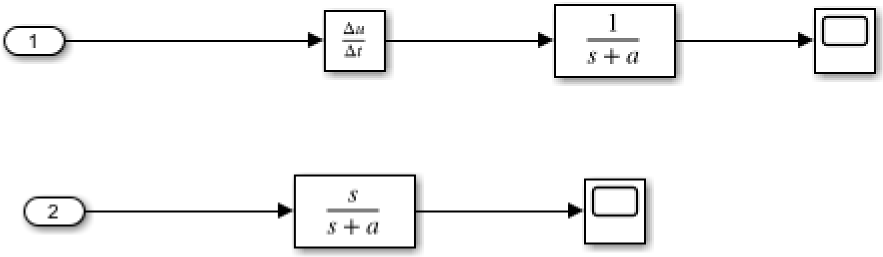
$$\frac{Y(s)}{X(s)} = s$$

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.

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

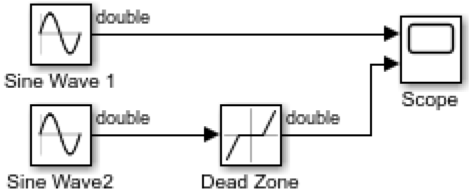
$$\frac{s}{s+a}$$

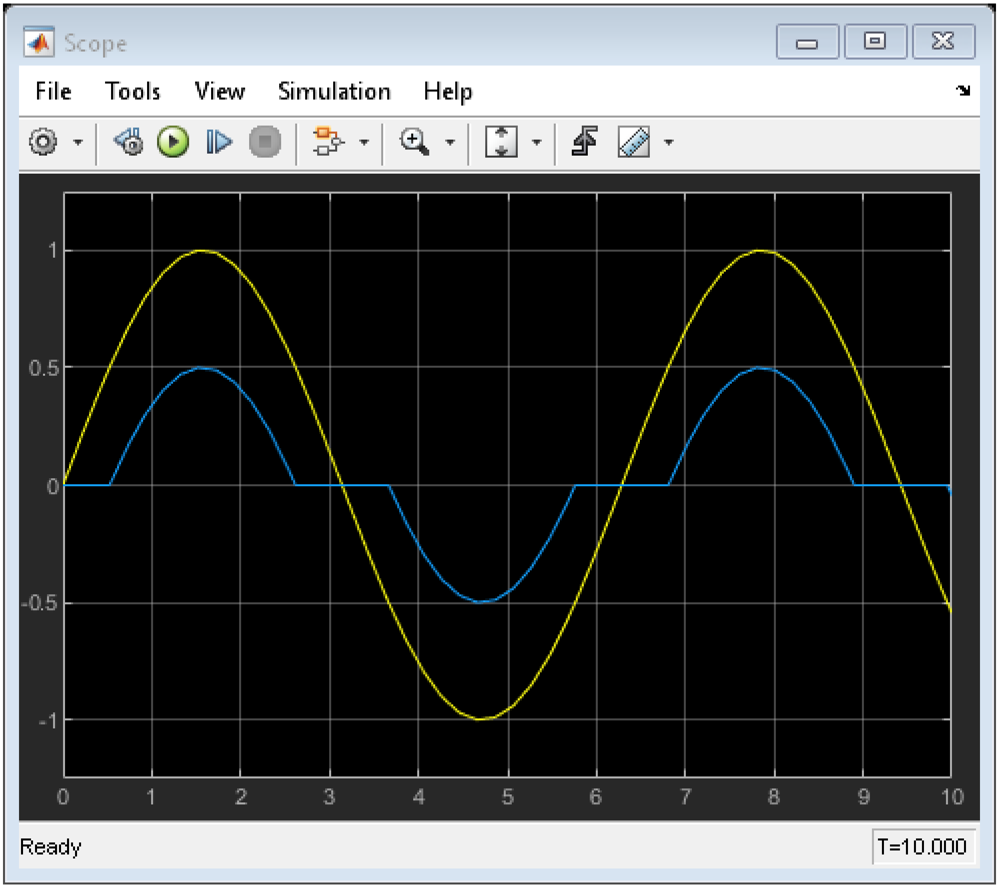
For example, you can replace the first set of blocks in this figure with the blocks below them:



View Dead Zone Output on Sine Wave

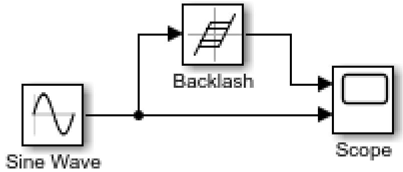
This example shows the effect of the Dead Zone block on a sine wave. The model uses a dead zone lower limit of `-0.5` and an upper limit as `0.5`. Set these values through the parameters **Start of Dead Zone** and **End of Dead Zone** .

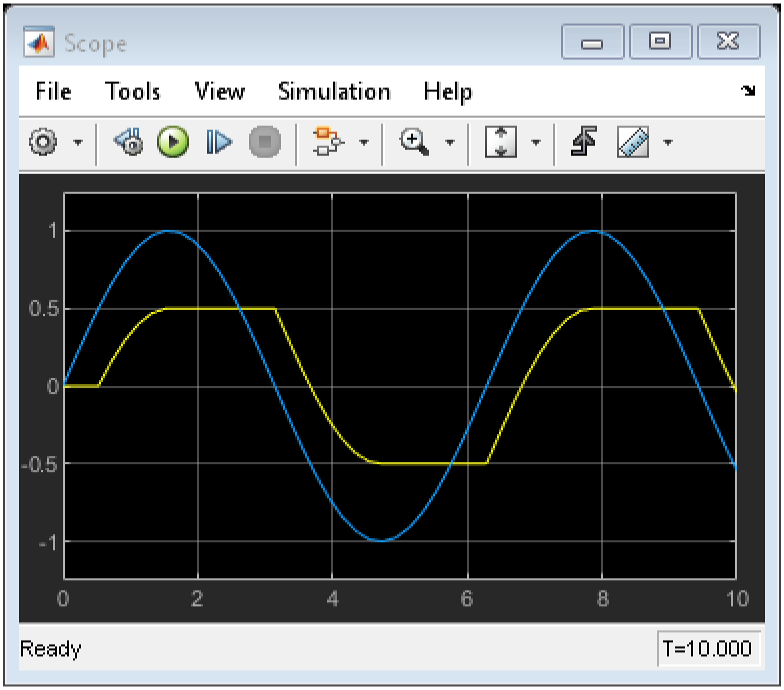



Scope
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

View Backlash Output on Sine Wave

This example shows the effect of the Backlash block on a sine wave using default parameters. The initial **Deadband width** is 1 and the **Initial output** is 0.

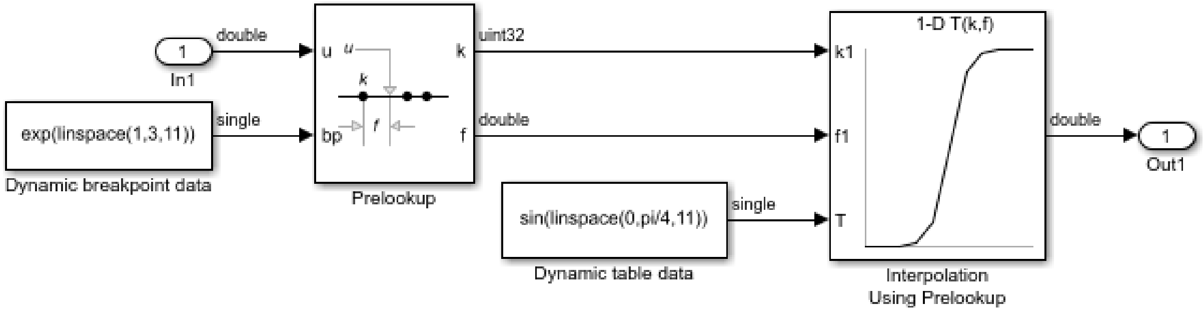




The initial deadband is centered around 0 and has a width of 1, which extends .5 in each direction. The output from the Backlash block begins at 0 and does not change until the input reaches the edge of the deadzone at .5. Then the output engages in a positive direction and changes an equal amount as the input. After the input reaches a value of 1, it starts moving in a negative direction. At this point the output disengages and stays flat until the input passes through the deadband width of 1. Once the input reaches the end of the deadband zone at 0, then the output engages and starts moving in a negative direction with the input.

Prelookup With External Breakpoint Specification

This example shows how to feed a breakpoint dataset from a Constant block to the bp input port of the Prelookup block.



The Prelookup block inherits the following breakpoint attributes from the bp input port:

- **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 attributes:

- **Minimum**: `Inf`
- **Maximum**: `Inf`
- **Data type**: `single`

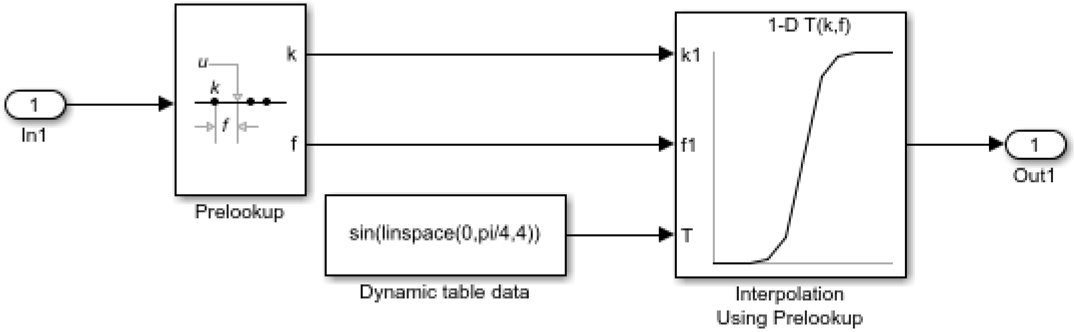
Simulink® uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision, floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.

Prelookup with Evenly Spaced Breakpoints

This example shows how to specify evenly spaced breakpoint data in the Prelookup block.

In the **Breakpoints data** section, the **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]`.

An alternative way to specify evenly spaced breakpoints is to set **Specification** to `Explicit values` and set **Value** to `[25:12:61]`.

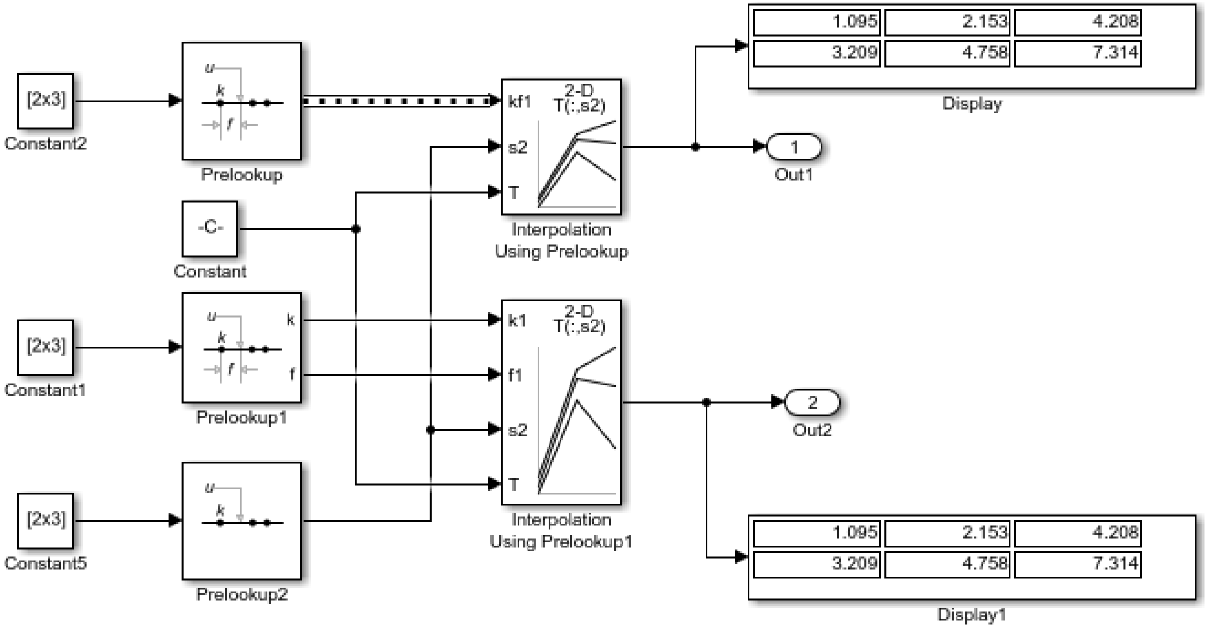


Simulink® uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoints and table data as double.

Configure the Prelookup Block to Output Index and Fraction as a Bus

This example shows how to output a bus containing the index (k) and fraction (f) from the Prelookup block. The bus object can then be used as an input to the Interpolation Using Prelookup block. The example also shows how to get the same results without using a bus object.

Open and simulate the model.



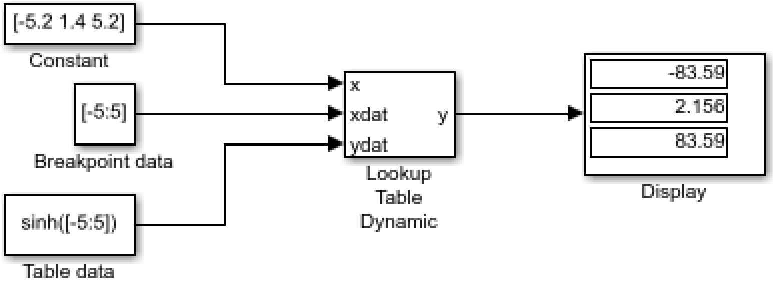
At the top of the model, open the dialog box for the Prelookup block. In the **Main** tab, note that **Output selection** is set to `Index and fraction as bus`. In the **Data Types** tab, note that **Output** is set to `Bus: myBus`. In the Simulink® Editor, select **File>Model Properties** and open the **Callbacks** tab. In the model's `PreLoadFcn`, the code defines the bus object `myBus`, which specifies the index as the first bus element and the fraction as the second element.

Open the dialog box for the Interpolation Using Prelookup block. 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.

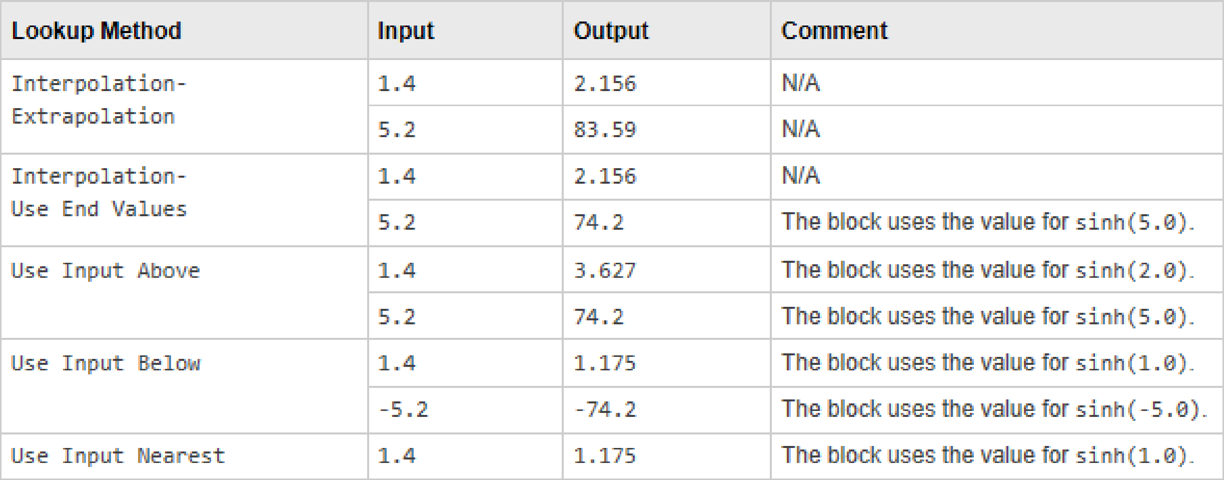
Approximating the sinh Function Using the Lookup Table Dynamic Block

This example shows how to use the Lookup Table Dynamic block to approximate the `sinh` function. The breakpoint data is given by the vector `[-5:5]` and the table data is given by the vector `sinh([-5:5])`. The input `x` is provided by the Constant block as a 1-by-3 vector containing values that are below, within, and above the breakpoint data values.

To see how each lookup method handles input values that are below, within, and above the breakpoint data values, change the value of the **Lookup Method** parameter on the Lookup Table Dynamic block.



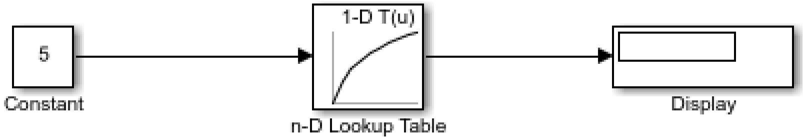
The Lookup Table Dynamic block outputs the following values when using the specified lookup methods and inputs.



Lookup Method	Input	Output	Comment
`Interpolation-Extrapolation`	`1.4`	`2.156`	N/A
	`5.2`	`83.59`	N/A
`Interpolation-Use End Values`	`1.4`	`2.156`	N/A
	`5.2`	`74.2`	The block uses the value for `sinh(5.0)`.
`Use Input Above`	`1.4`	`3.627`	The block uses the value for `sinh(2.0)`.
	`5.2`	`74.2`	The block uses the value for `sinh(5.0)`.
`Use Input Below`	`1.4`	`1.175`	The block uses the value for `sinh(1.0)`.
	`-5.2`	`-74.2`	The block uses the value for `sinh(-5.0)`.
`Use Input Nearest`	`1.4`	`1.175`	The block uses the value for `sinh(1.0)`.

Create a Logarithm Lookup Table

This example shows how to use the n-D Lookup Table block to create a logarithm lookup table. The lookup table allows you to approximate the common logarithm (base 10) over the input range [1,10] without performing an expensive computation.

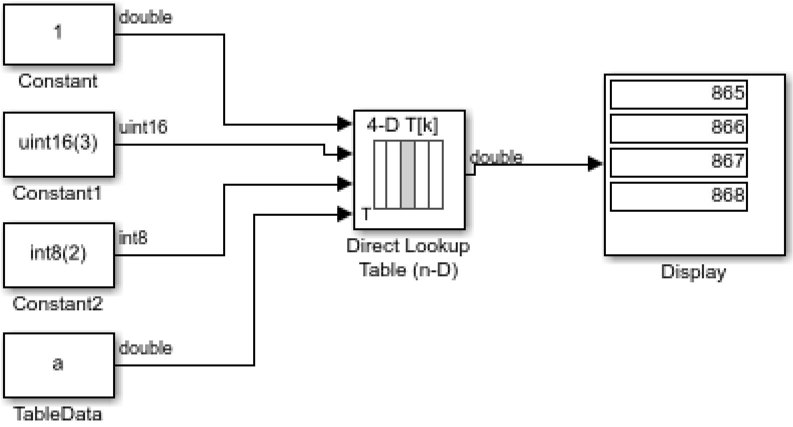


Providing Table Data as an Input to the Direct Lookup Table Block

This example shows how to provide table data as an input to the Direct Lookup Table block. In the following model, `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 block labeled TableData feeds a 4-D array to the Direct Lookup Table (n-D) block, with a data type of double. Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```
a(:,2,4,3)
```

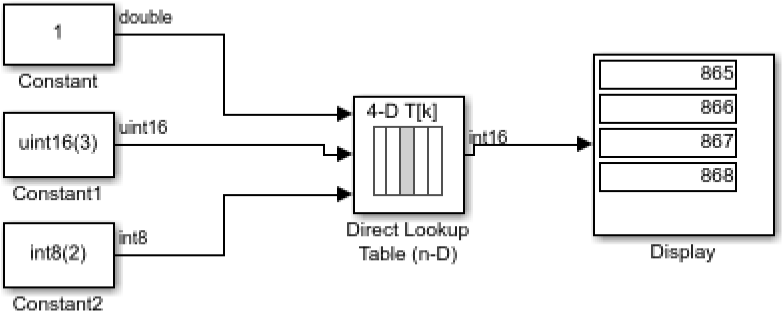
The output has the same data type as the table data input signal: double.

Specifying Table Data in the Direct Lookup Table Block Dialog Box

This example shows how to specify table data on the dialog box of the Direct Lookup Table (n-D) block. In the following model, the table data is 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:

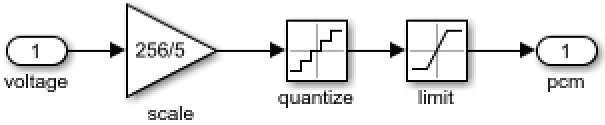
```
a(:,2,4,3)
```

The output data type matches the Direct Lookup Table block's **Table data type**, which is set to `int16`.

Using the Quantizer and Saturation blocks in sldemo_boiler

This example shows how the Quantizer and Saturation blocks are used in the model ex_sldemo_boiler. The ADC subsystem digitizes the input analog voltage by:

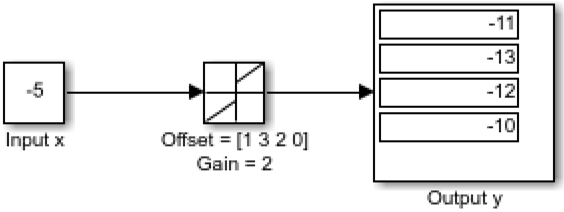
- Multiplying the analog voltage by 256/5 with the Gain block
- Rounding the value to integer floor with the Quantizer block
- Limiting the output to a maximum of 255 (the largest unsigned 8-bit integer value) with the Saturation block



This subsystem models a typical 8-bit analog-to-digital converter having an input range of 0-5 volts

Scalar Expansion with the Coulomb and Viscous Friction Block

This example shows a model with a scalar input to a Coulomb & Viscous Friction block that uses scalar expansion to output a vector.



Double click the friction block to see the parameters. **Coefficient of viscous friction (Gain)** is a scalar value `2`, but **Coulomb friction value (Offset)** is a vector value `[1 3 2 0]`. Therefore, the block uses element-wise scalar expansion to compute the output.

Each output is calculated using this formula.

$$y = sign(x).*(Gain.*abs(x)+Offset)$$

For example, the first offset 1 is calculated as follows.

$$y = - * ((2*5)+1)$$

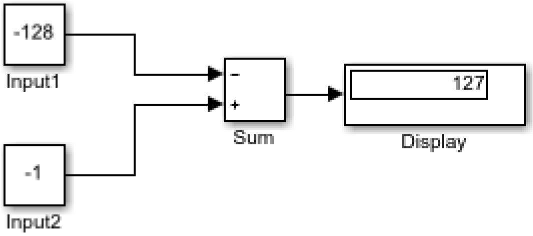
$$y = -11$$

If the dimensions for the input and `Offset` are the same, then no expansion is necessary.

Sum Block Reorders Inputs

This example shows how the Sum block reorders inputs. If you use a `-` sign as the first operation, the block reorders the inputs, if possible, to use a + operation. For example, in the expression `output = -a-b+c`, the Sum block reorders the input so that `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` .



Both the constant inputs use `int8` data types The Sum block also uses `int8` for the accumulator and output data types and has **Saturate on integer overflow** turned on. The Sum block reorders the inputs to give the ideal result of `127`.

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(-12) = 127`
4. Caclulates the block output. `Output = int8(127) = 127`

If the Sum block does not reorder the inputs, then you get the nonideal result of `126`.

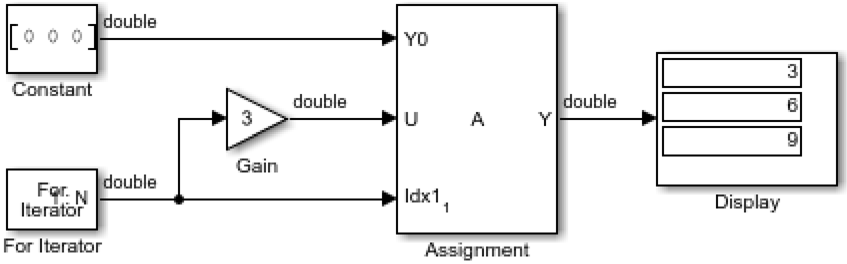
1. Initializes the accumulator by using the first input port. `Accumulator = int8(-(-128)) = 127`
2. Because saturation is on, the initial value of the accumulator saturates at `127` and does not wrap.
3. Continues to accumulate values. `Accumulator = Accumulator + int8(-1) = 126`

4 Calculates the block output. `Output = int8(126) = 126`

To explicitly specify a unary minus operation for `output = -a-b+c`, you can use the Unary Minus block in the Math Operations library.

Iterated Assignment with the Assignment Block

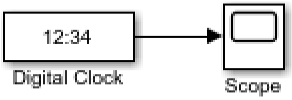
This example shows using the Assignment block to assign values computed in a For or While Iterator loop to successive elements. You can use vector, matrix or multidemensional signals and do the assignment in a single time step. In this model, the For Iterator block creates a vector signal each of whose elements equals $3 * i$ where i is the index of the element.

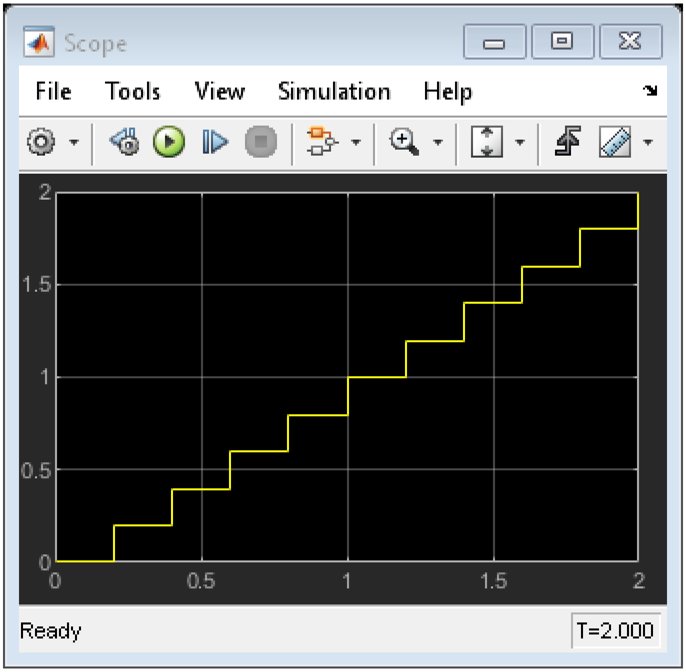


The iterator generates indices for the Assignment block. On the first iteration, the Assignment block copies the first input (Y0) to the output (Y) and assigns the second input (U) to the output Y(E1). On successive iterations, the Assignment block assigns the current value of U to Y(Ei), that is, without first copying Y0 to Y. These actions occur in a single time step.

View Sample Time Using the Digital Clock Block

This example shows how to view the simulation sample time at a specified sampling interval using the Digital Clock block. In this model, the Scope shows the output of a Digital Clock block with the **Sample time** set to `0.2`.

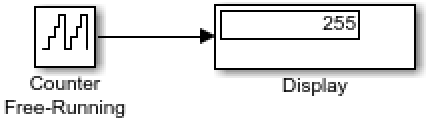




In this configuration, the Digital Clock block outputs the simulation time every 0.2 seconds. Otherwise, the block holds the output at the previous value.

Bit Specification Using a Positive Integer

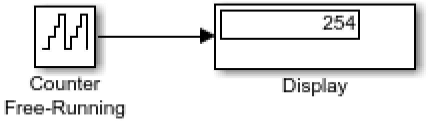
This example shows how to specify the **Number of bits** in the Counter Free-Running block as a positive integer.



At t = 255, the counter reaches the maximum value of `(2^8)-1`. If you increase the stop time of the simulation to 256, the counter wraps to zero.

Bit Specification Using an Unsigned Integer Expression

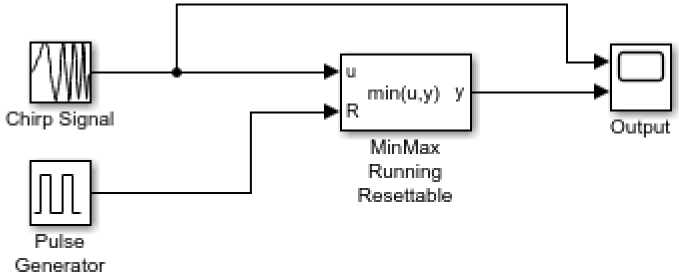
This example shows how to specify the **Number of bits** in the Counter Free-Running block as an unsigned integer expression.

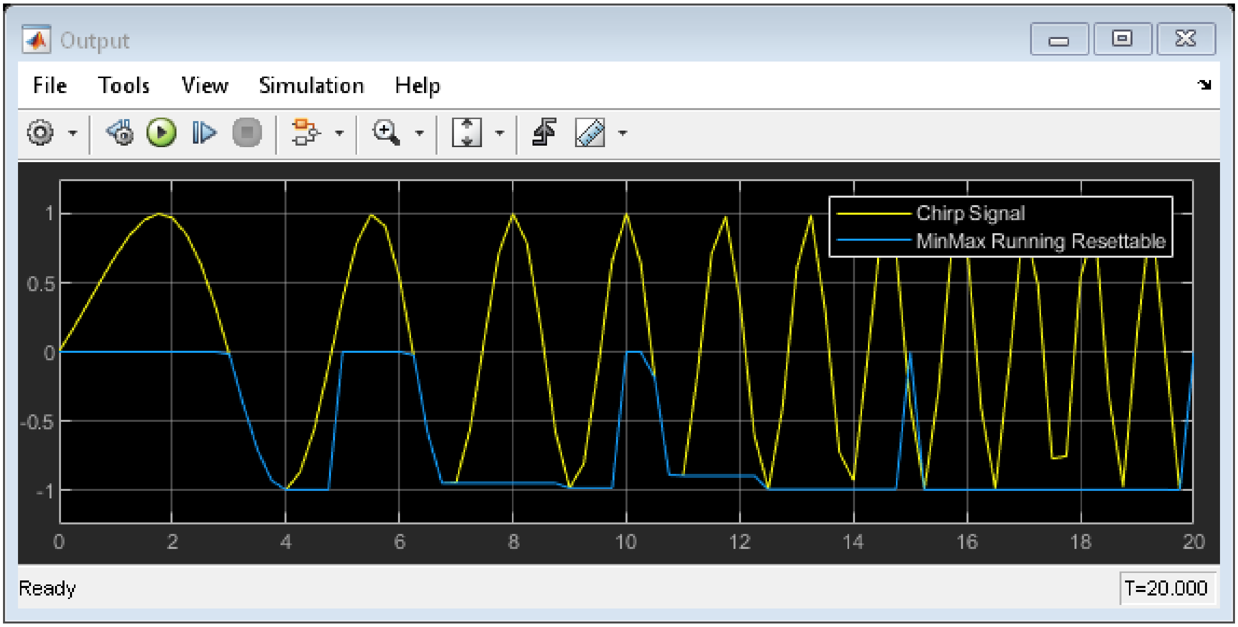


At t = 254, the counter reaches the maximum value of `uint8(2^(uint8(8))-1)`. If you increase the stop time of the simulation to 255, the counter wraps to zero.

Track Running Minimum Value of Chirp Signal

This example shows how to track the running minimum value of a signal generated by the Chirp Signal block.

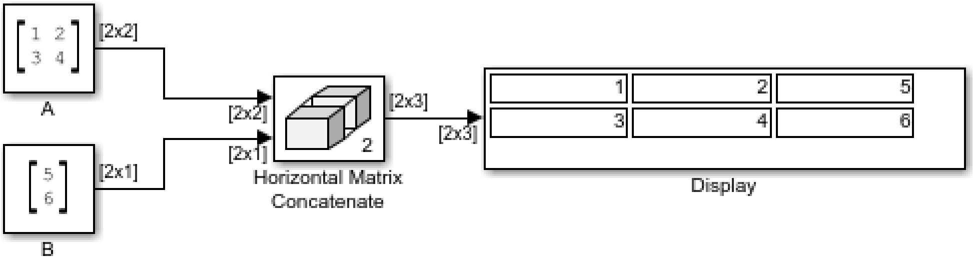




The Chirp Signal block generates a sine wave whose frequency increases at a linear rate with time. The MinMax Running Resettable block tracks the minimum value of that chirp signal over time. The running minimum value is reset every 5 seconds by the Pulse Generator block.

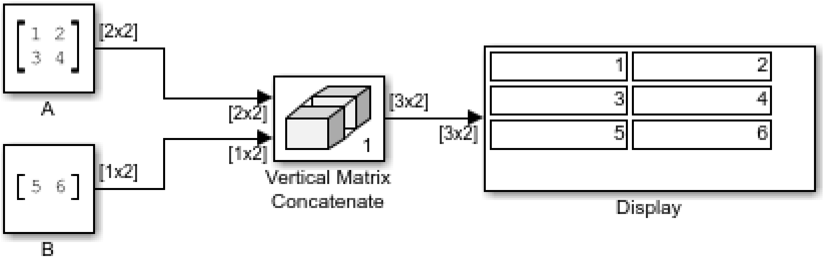
Horizontal Matrix Concatenation

This example shows how to perform a horizontal matrix concatenation with the Matrix Concatenate block. When you set the **Concatenate dimension** parameter to 2 and the inputs are 2-D matrices, the block performs horizontal matrix concatenation and places the input matrices side-by-side to create the output matrix.



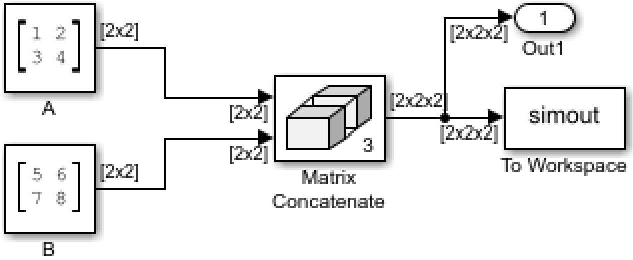
Vertical Matrix Concatenation

This example shows how to perform a vertical matrix concatenation with the Matrix Concatenate block. When you set the **Concatenate dimension** parameter to 1 and the inputs are 2-D matrices, the block performs vertical matrix concatenation and places the input matrices on top of each other to create the output matrix.



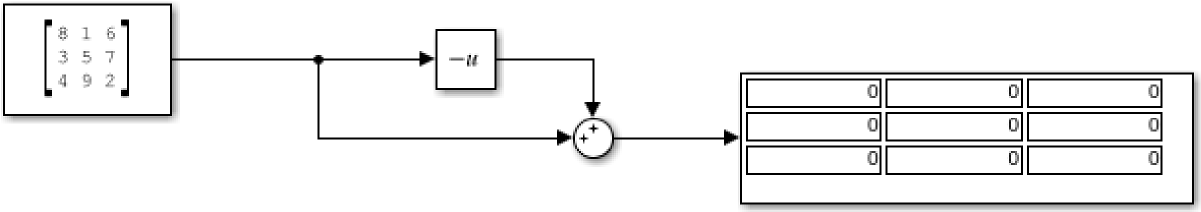
Multidimensional Matrix Concatenation

This example shows how to perform multidimensional matrix concatenation with the Matrix Concatenate block. When you set the **Concatenate dimension** parameter to 3 and the inputs are 2-D matrices, the block performs multidimensional matrix concatenation.



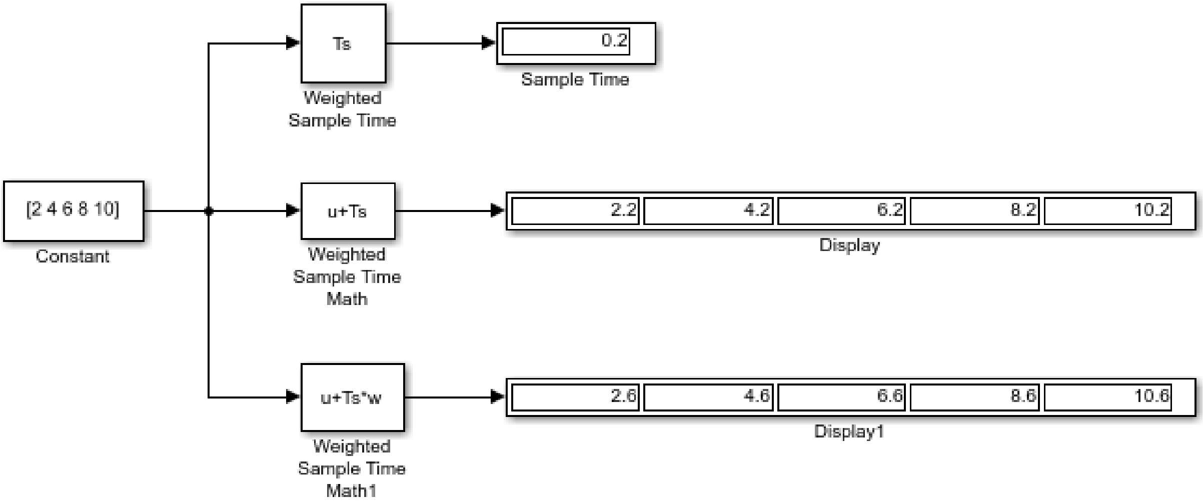
Unary Minus of Matrix Input

This example shows how to compute the unary minus of a matrix input.



Sample Time Math Operations Using the Weighted Sample Time Math Block

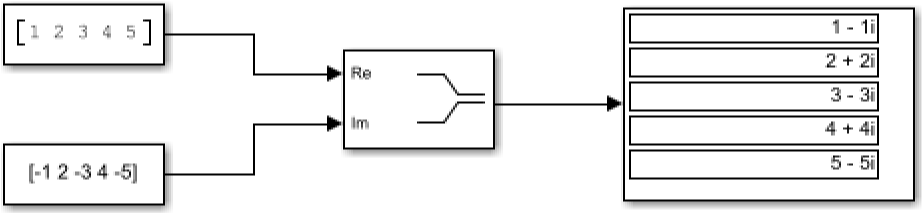
This example shows how to add the sample time value to a signal using the Weighted Sample Time Math block.

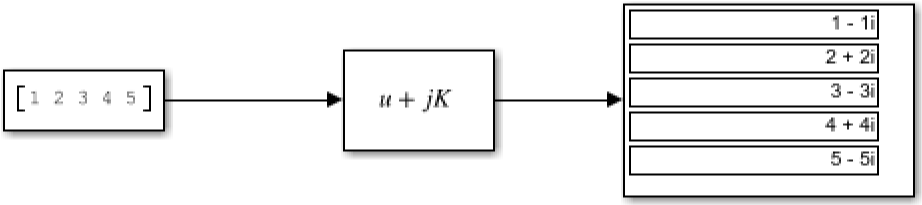


Using the Weighted Sample Time block, you can see the sample time of this model is `0.2`. When you set the **Operation** parameter to `+` and the **Weight value** to `1` on the Weighted Sample Time Math block, the block adds the sample time value of `0.2` to the input signal. When you set the **Weight value** to `3` in the Weighted Sample Time Math1 block, the block adds `Ts*3` to the input signal, thus increasing each value by `0.6`.

Construct Complex Signal from Real and Imaginary Parts

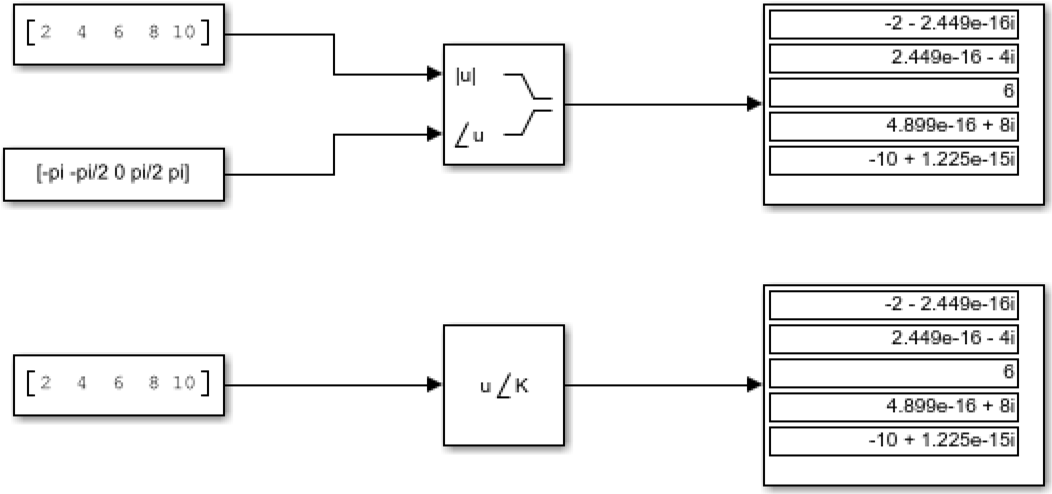
This example shows how to use the Real-Imag to Complex block to construct a complex-valued signal from real and imaginary parts. You can provide both the real and imaginary parts as block inputs, or provide one value as an input, and the other on the block dialog box.





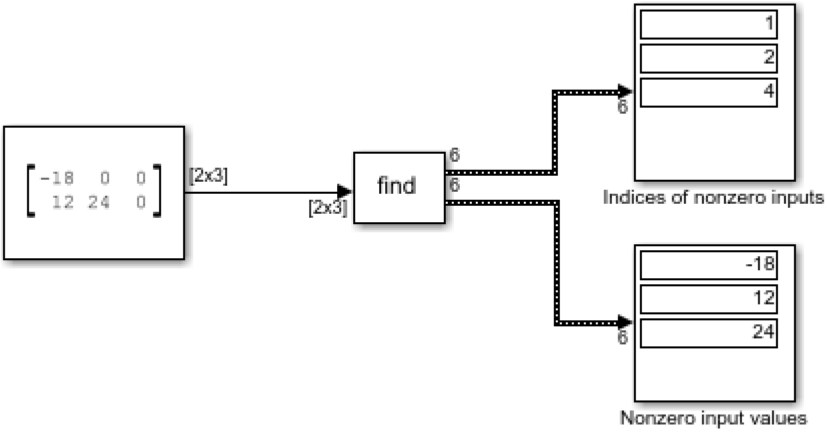
Construct Complex Signal from Magnitude and Phase Angle

This example shows how to use the Magnitude-Angle to Complex block to construct a complex-valued signal. You can provide both the magnitude and phase angle as block inputs, or provide one value as an input, and the other on the block dialog box.



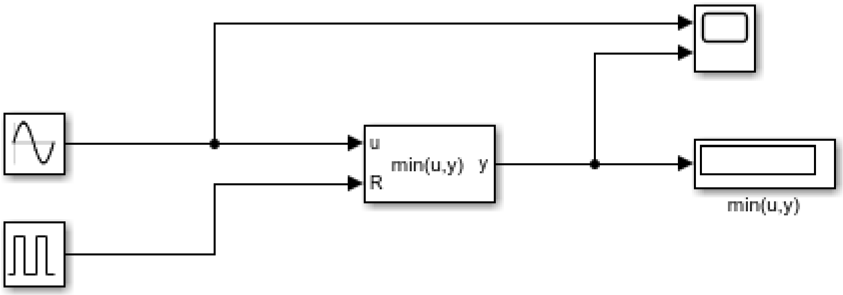
Find Nonzero Elements in an Array

This example shows how to use the Find block to find nonzero elements in an array. In the following model, the block is configured to output both the one-based linear index and the value of each nonzero element.

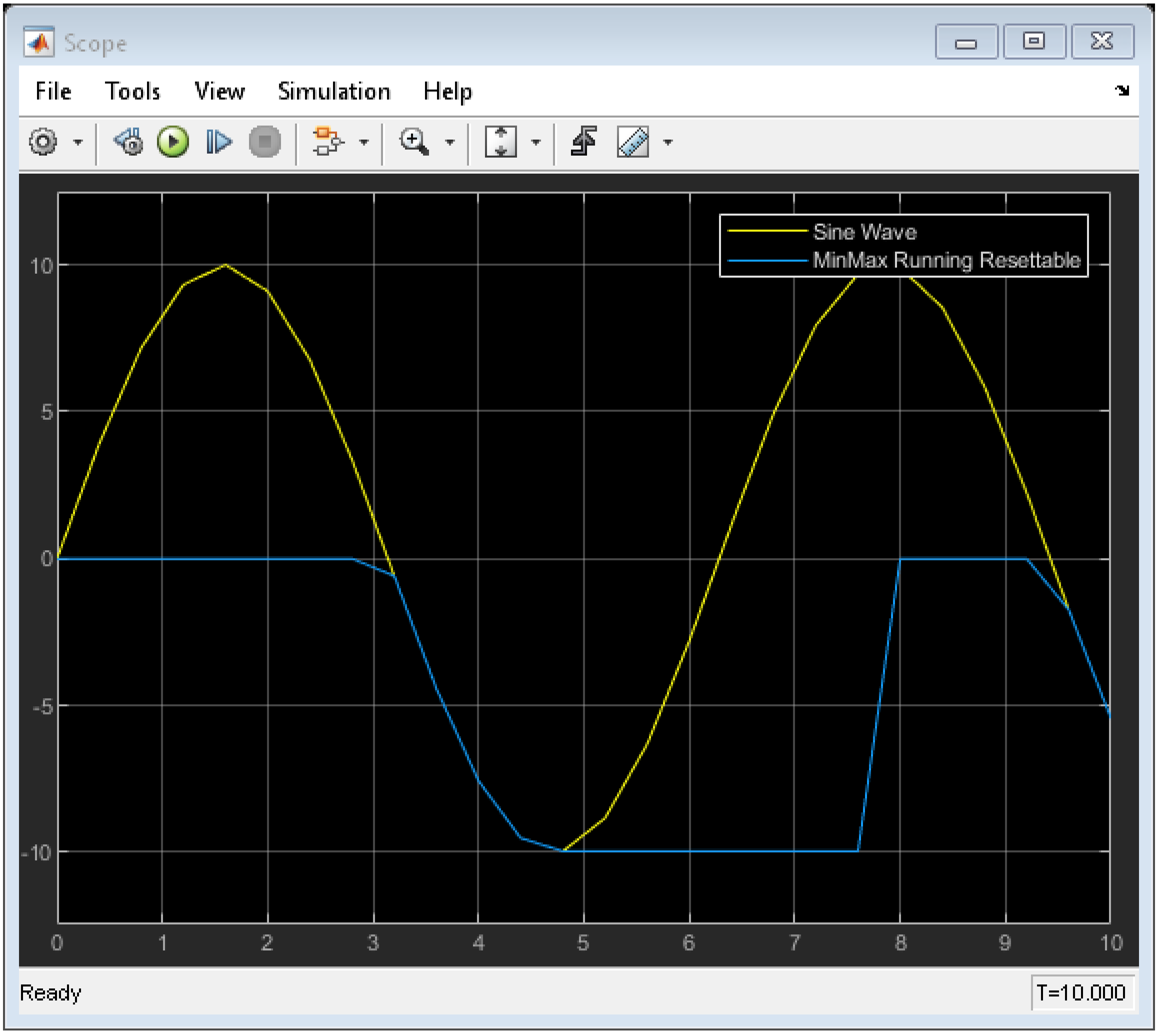


Calculate the Running Minimum Value with the MinMax Running Resettable Block

This example shows how to use the MinMax Running Resettable block to calculate the running minimum value. To watch how the running minimum value changes at each time step, you can use the Step Forward button to advance the simulation one step at a time.

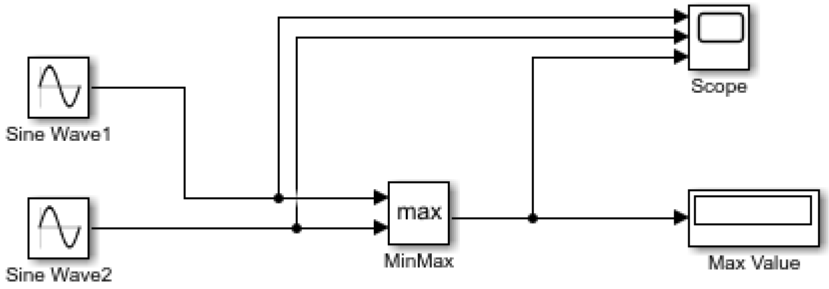


After running the full simulation, you can view the results in the Scope. The initial value of the running minimum is `0`. It begins tracking the Sine Wave signal when the sine wave values turn negative. When the MinMax Running Resettable block receives a reset signal at T=8, the block resets the running minimum value to 0. The running minimum value tracks at `0` for a few time steps, until the sine wave values turn negative again.

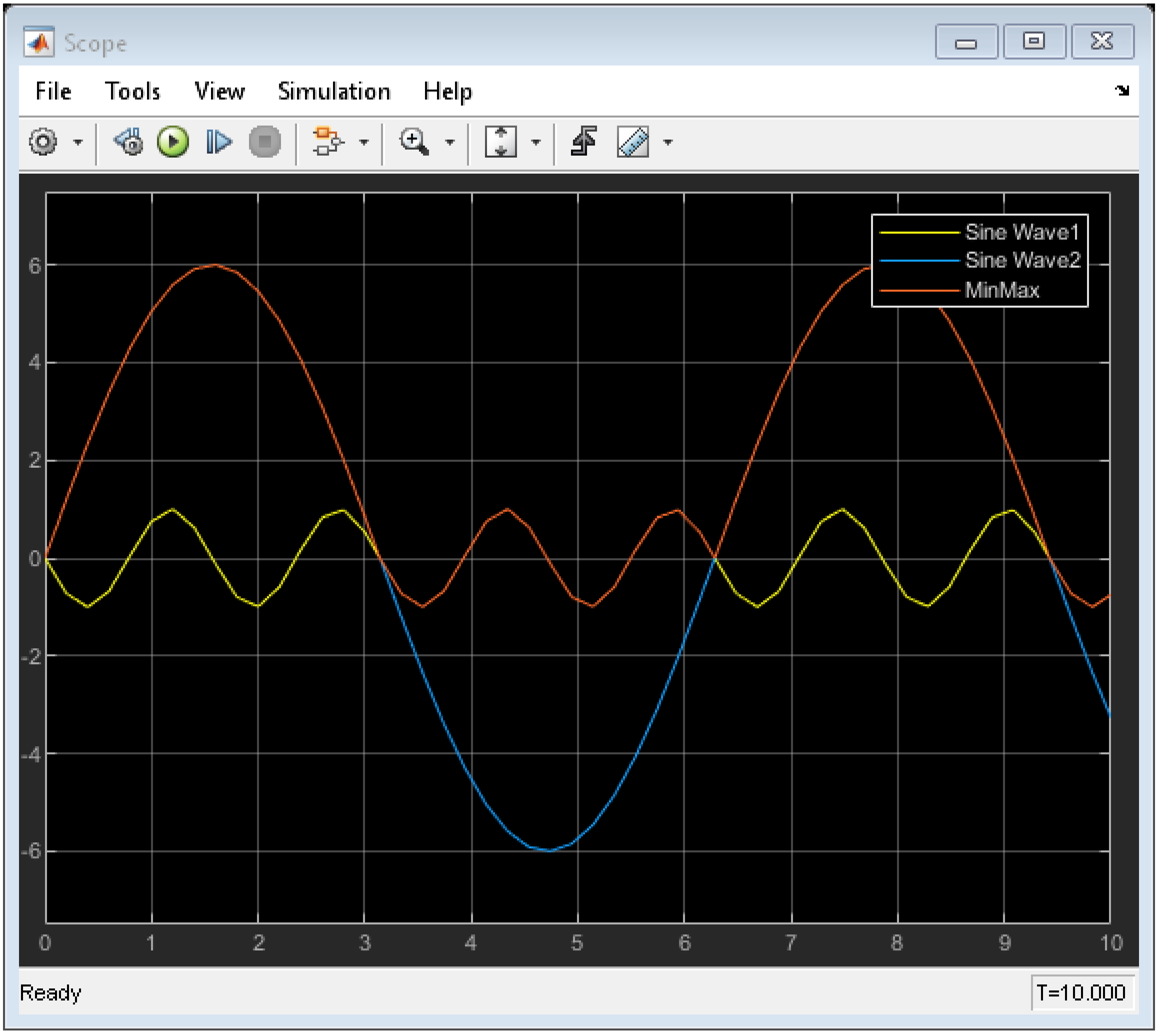

Scope
File
Tools
View
Simulation
Help
Sine Wave
MinMax Running Resettable
10
5
0
-5
-10
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

Find Maximum Value of Input

This example shows how to use the MinMax block to output the maximum value of two sine waves.

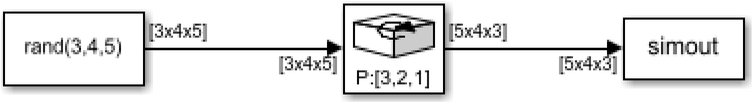


After running the full simulation, you can view the results in the Scope. Initially, the maximum value (orange line) tracks SineWave2. When the SineWave2 values turn negative, the maximum value begins tracking SineWave1. When the SineWave2 values become positive again, the maximum value resumes tracking SineWave2.


Scope
File
Tools
View
Simulation
Help
Sine Wave1
Sine Wave2
MinMax
6
4
2
0
-2
-4
-6
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

Permute Array Dimensions

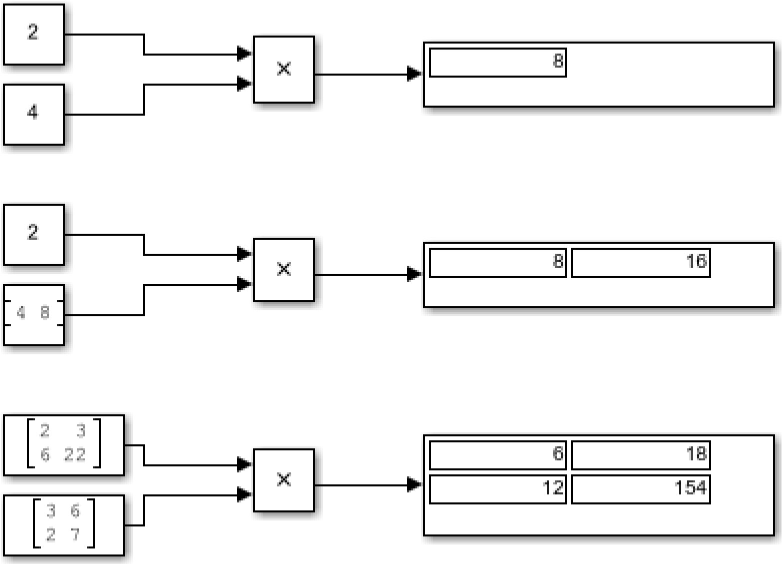
This example shows how to use the Permute Dimensions block to permute the first and third dimensions of a 3-by-4-by-5 input array.



When you set the **Order** parameter to `[3,2,1]`, the block permutes the first and third dimensions, and outputs a 5-by-4-by-3 array.

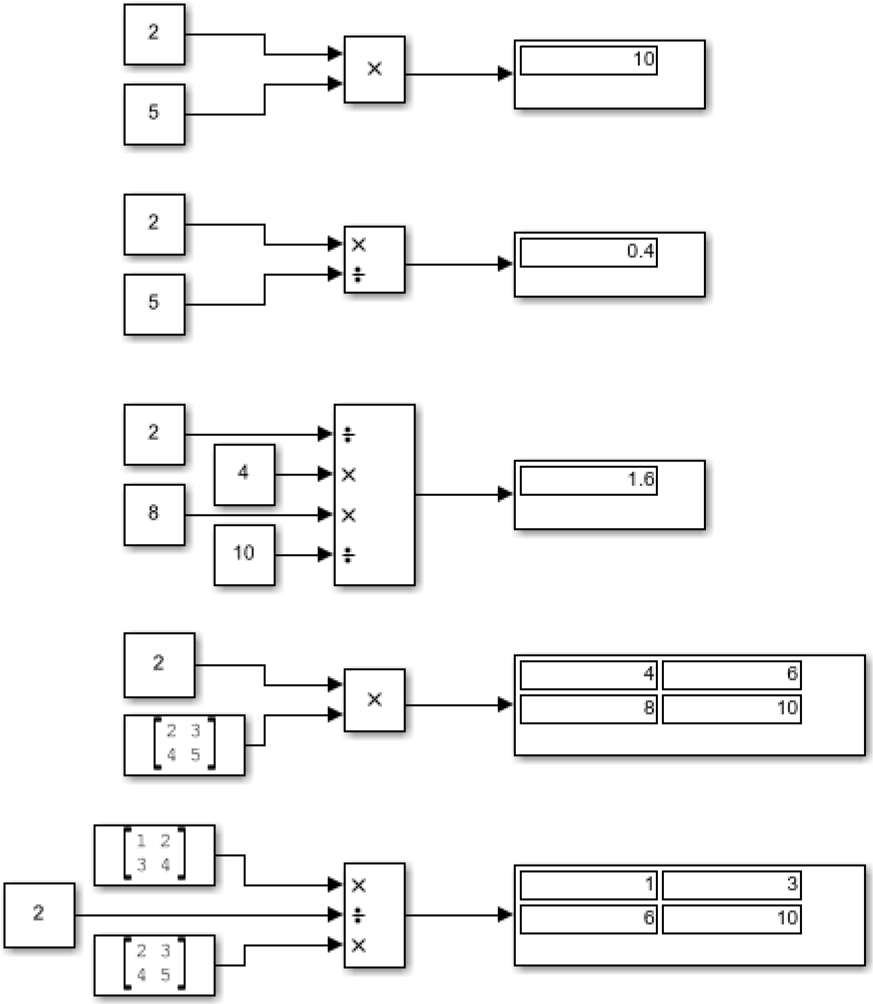
Multiply Inputs of Different Dimensions with the Product Block

This example shows how to perform element-wise `(.*)` multiplication of inputs using the Product block. In this example, the Product block multiplies two scalars, a scalar and a vector, and two 2x2 matrices.



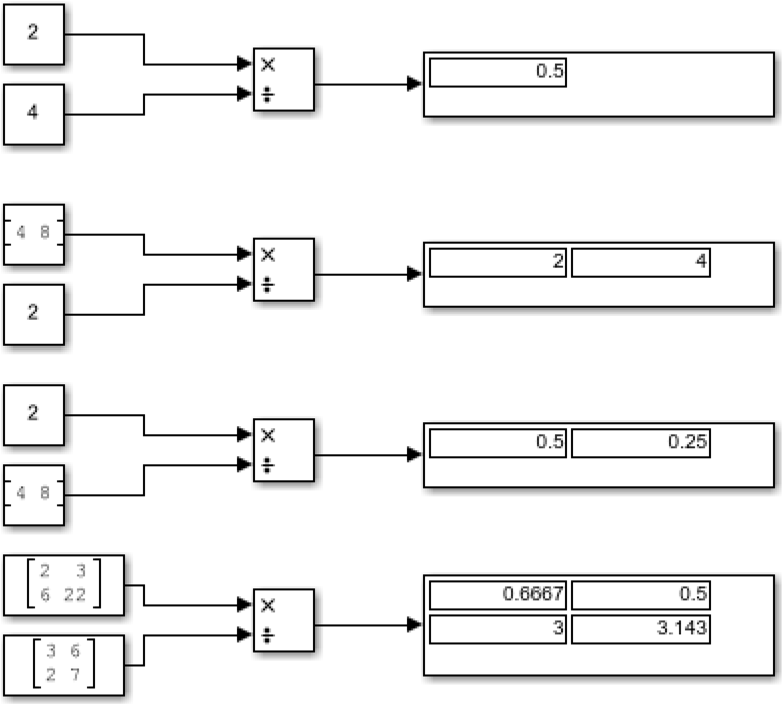
Multiply and Divide Inputs Using the Product Block

This example shows how to multiply and divide several input signals using the Product block.



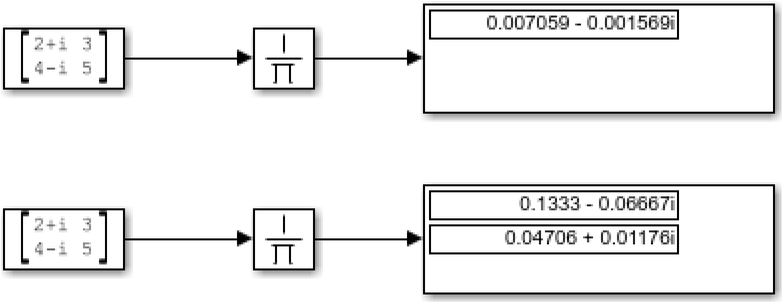
Divide Inputs of Different Dimensions Using the Divide Block

This example shows how to perform element-wise `(.*)` division of two inputs using the Divide block. In this example, the Divide block divides two scalars, a vector by a scalar, a scalar by a vector, and two matrices.



Complex Division Using the Product of Elements Block

This example shows how to perform element-wise complex division using the Product of Elements block.



The top Product of Elements block collapses the matrix input to a scalar by taking successive inverses of the four elements:

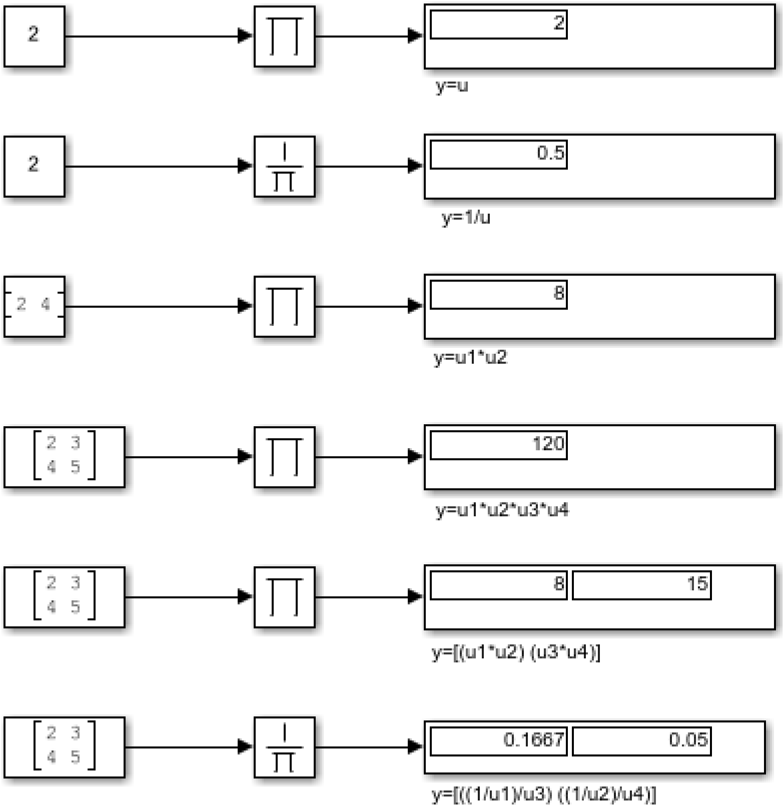
- `y = ((((1/2+i)/3)/4-i)/5)`

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)`

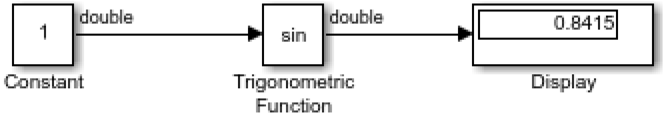
Element-Wise Multiplication and Division Using the Product of Elements Block

This example shows how to use the Product of Elements block to perform element-wise multiplication and division of inputs.



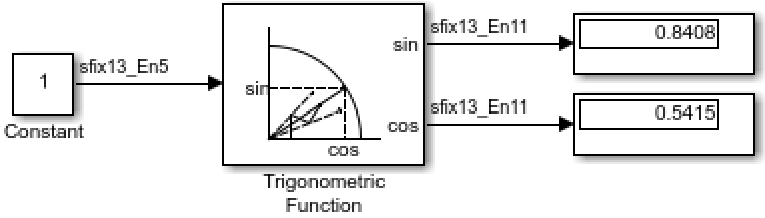
sin Function with Floating-Point Input

This example shows how to use the Trigonometric Function block to compute the sine of a floating-point input. The output of the Trigonometric Function block has the same data type as the input because the input data type is floating-point and the **Approximation method** is none.



sincos Function with Fixed-Point Input

This example shows how to use the Trigonometric Function block to compute the CORDIC approximation of `sincos` for a fixed-point input signal.



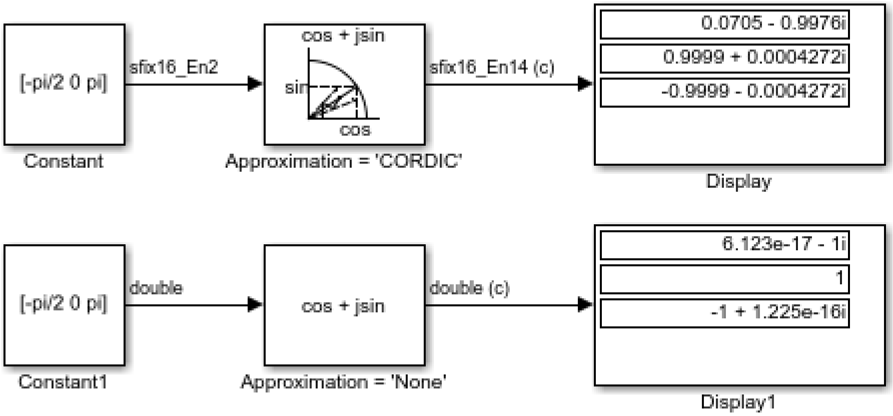
The Trigonometric Function block parameters are:

- **Function**: `sincos`
- **Approximation method**: `CORDIC`
- **Number of iterations**: `11`

When using the CORDIC approximation method, the input to the Trigonometric Function block must be in the range `[-2pi,2pi)`. The output type of the Trigonometric Function block is `fixdt(1,13,11)` because the input is a fixed-point signal and the **Approximation method** is set to `CORDIC`. The output fraction length equals the input word length minus two.

Trigonometric Function Block Behavior for Complex Exponential Output

This example compares the complex exponential output for two different configurations of the Trigonometric Function block.

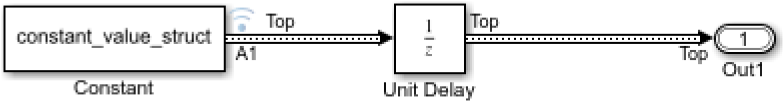


When the **Approximation method** is `CORDIC`, the input data type can be fixed point, in this case: `fixdt(1,16,2)`. The output data type is `fixdt(1,16,14)` because the output fraction length equals the input word length minus two.

When the **Approximation method** is `None`, the input data type must be floating point. The output data type is the same as the input data type.

Output a Bus Object from the Constant Block

This example shows how to output a bus object from the Constant block. The six Constant blocks used to create a bus object in the `ex_busic` example (see “Create Partial Structures for Initialization”) are replaced by a single constant block in this example.



To verify that the output from the Constant block reflects the values from the `constant_value_struct`, enter the following at the MATLAB command line:

```
constant_value_struct
```

```
constant_value_struct = 

  struct with fields:

    A: [1x1 struct]
    B: 5
    C: [1x1 struct]
```

Examine the logged data in the logsout variable, focusing on the A1 bus signal. The `constant_value_struct` structure sets the `B` element to `5`.

```
logsout.get('A1').Values.B.Data(1)
```

```
ans =

     5
```

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”.

Define Enumerated Type

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
```

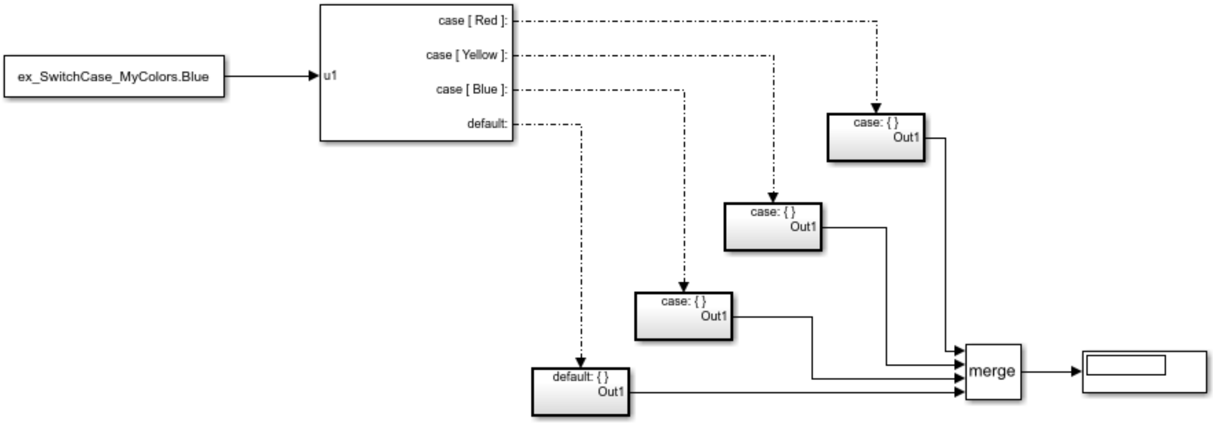
Alternatively, you can use the function Simulink.defineIntEnumType to define the type.

```
Simulink.defineIntEnumType('ex_SwitchCase_MyColors',...
{'Red','Yellow','Blue','Mauve'},[0;1;2;3])
```

Explore Example Model

Open the example model ex_enum_switch_case.

```
open_system('ex_enum_switch_case')
```



Open the Enumerated Constant block dialog box. The constant output value is `ex_SwitchCase_MyColors.Blue`.

Open the Switch Case block dialog box. The **Case conditions** box is specified as 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.

Open the Switch Case Action Subsystem blocks. The subsystems each contain a Constant block that uses a different constant value.

Control Execution During Simulation

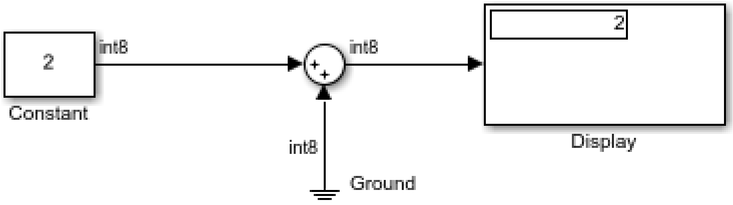
Simulate the model. The Display block shows the value `5`, which corresponds to the case `ex_SwitchCase_MyColors.Blue`.

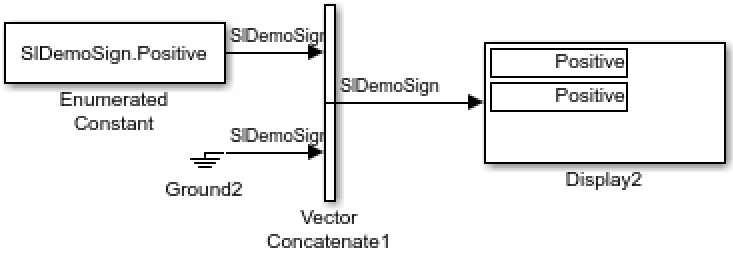
In the Enumerated Constant block dialog box, specify **Value** as `ex_SwitchCase_MyColors.Red` and click **Apply**. The Display block shows `19`.

Specify **Value** as `ex_SwitchCase_MyColors.Mauve` and click **Apply**. The Display block shows `3`, which corresponds to the default case.

Integer and Enumerated Data Type Support in the Ground Block

This example shows how to use the Ground block to ground block input ports that have integer and enumerated data types. In top row of 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, and the Ground block outputs a signal with zero value, and data type `int8`.





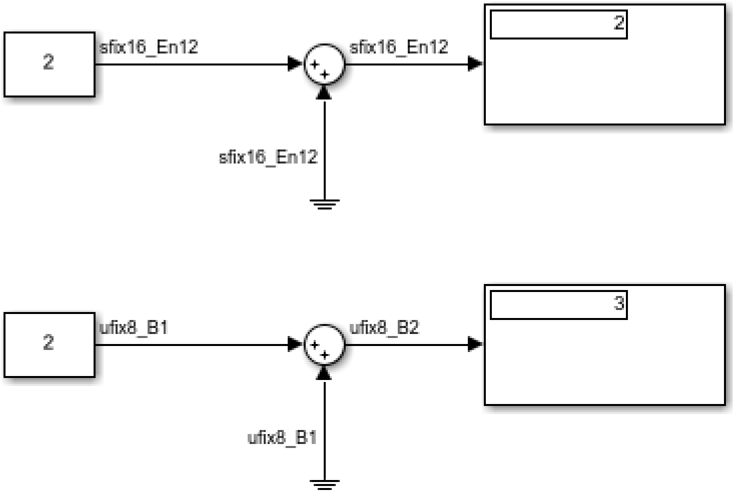
In the bottom row of this example, the Ground block is connected to a port with an enumerated data type. For enumerated data types, the Ground block outputs the default value of the enumeration. This behavior applies whether or not:

- 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.

Fixed-Point Data Type Support in the Ground Block

This example shows how to use the Ground block to ground block input ports that have fixed-point data types. The top row of this example illustrates the Ground block behavior when the fixed-point data type can represent zero. In that case, the Ground block outputs a signal with zero value, and the same fixed-point data type as the port it is connected to.



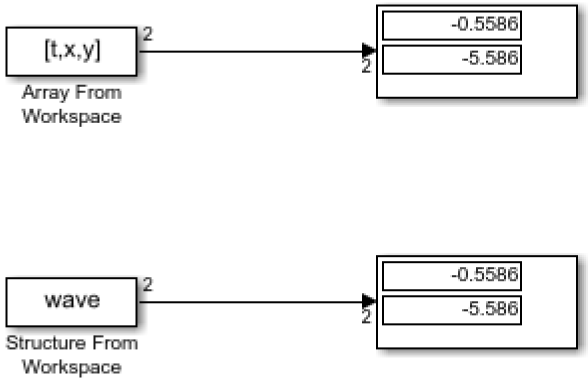
In the bottom row of this example, the output of the Constant block determines the data type of the port to which the Ground block is connected (`fixdt(0,8,1,1)`). Because zero cannot be represented exactly by the data type `fixdt(0,8,1,1)`, the Ground block outputs a nonzero value that is the closest possible value to zero (in this case, 1).

Read 1-D Array and Structure From Workspace

This example shows how to read 1-D signals from the MATLAB workspace. When you open the model, the following code is executed by a `PreLoadFcn` callback:

```
t = 0.2 * [0:49]';
x = sin(t);
y = 10*sin(t);
wave.time = t;
wave.signals.values = [x,y];
wave.signals.dimensions =2;
```

In the top row of the model, the From Workspace block reads the array `[t,x,y]` from the MATLAB workspace.



In the bottom row of the model, the From Workspace block reads the same values from the workspace, but this time they are read from a structure named `wave`.

Read Structure From Workspace Using Model Sample Time

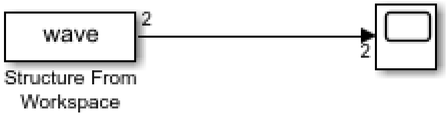
This example shows how to read a structure from the MATLAB workspace using a sample time specified in the From Workspace block. When you open the model, the following code is executed by a `PreLoadFcn` callback:

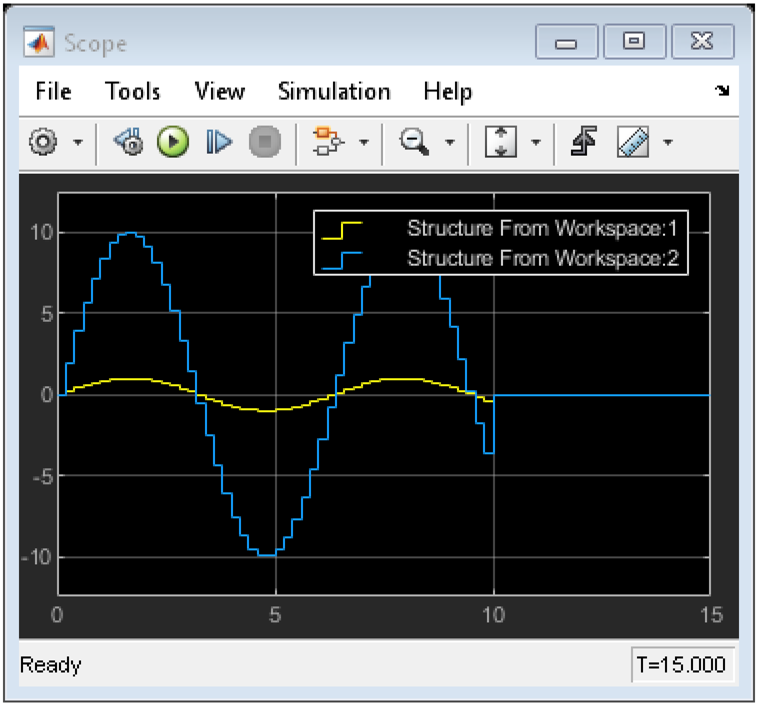
```
t = 0.2 * [0:49]';
x = sin(t);
y = 10*sin(t);
wave.time = [];
wave.signals.values = [x,y];
wave.signals.dimensions =2;
```

The From Workspace block is configured as follows:

- **Sample time**: `0.2`
- **Interpolate data**: off
- **Form output after final value by**: `Setting to zero`

When you run the model, the From Workspace block reads the structure `wave` from the workspace. After the last time hit for which workspace data is available, the block outputs `0`.




Scope
File
Tools
View
Simulation
Help
Structure From Workspace:1
Structure From Workspace:2
10
5
0
-5
-10
0
5
10
15
Ready
T=15.000

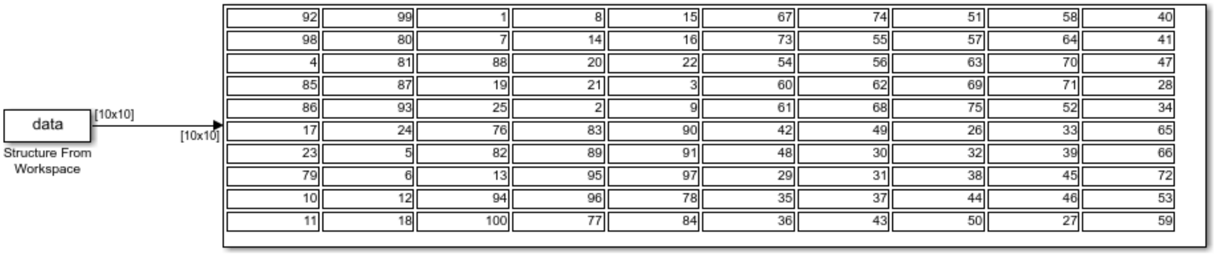
Read 2-D Signals in Structure Format From Workspace

This example shows how to read a 2-D structure from the MATLAB workspace. When you open the model, the following code is executed by a `PreLoadFcn` callback:

```
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];
```

This code creates 10-by-10 matrix (2-D signal) by using the `magic` function, and then creates a 3-D matrix by adding a time vector. 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.

When you run the model, the From Workspace block reads the structure `data` from the workspace.



92	99	1	8	15	67	74	51	58	40
98	80	7	14	16	73	55	57	64	41
4	81	88	20	22	54	56	63	70	47
85	87	19	21	3	60	62	69	71	28
86	93	25	2	9	61	68	75	52	34
17	24	76	83	90	42	49	26	33	65
23	5	82	89	91	48	30	32	39	66
79	6	13	95	97	29	31	38	45	72
10	12	94	96	78	35	37	44	46	53
11	18	100	77	84	36	43	50	27	59

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.

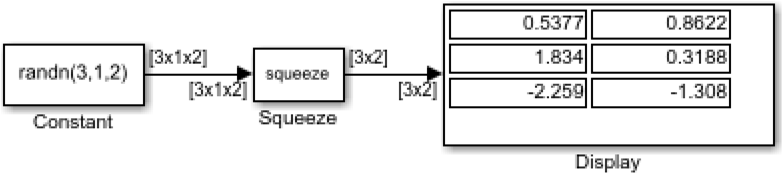
```
save('mySignals','ts','-v7.3')
```

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.

Eliminate Singleton Dimension with the Squeeze Block

This example shows a model using the Squeeze block to eliminate a dimension of size 1.



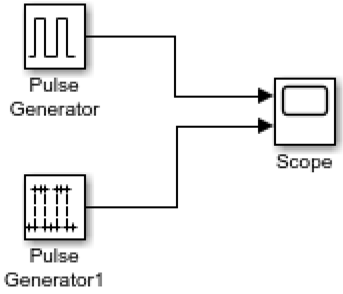
```
0.5377   0.8622
1.834    0.3188
-2.259   -1.308
```

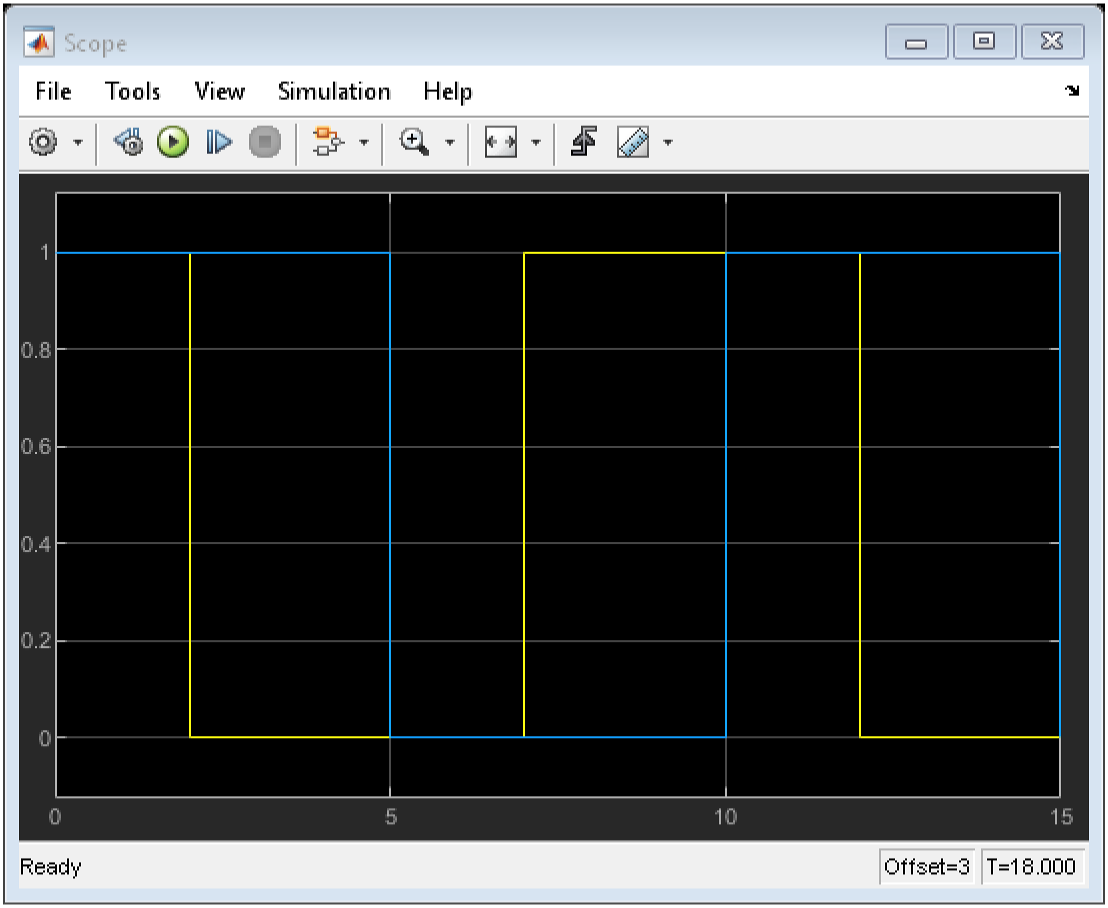
The Squeeze block converts a multidimensional array from the Constant block of size 3-by-1-by-2 into a 3-by-2 signal.

Difference Between Time- and Sample-Based Pulse Generation

This example shows the difference in behavior of the Pulse Generator block in time-based and sample-based modes.

Consider this model, with two Pulse Generator blocks. One block has the **Pulse type** parameter set to Time based, and the other to Sample based. Both blocks are configured to output a pulse with an amplitude of one that is `on` for five seconds, followed by `off` for five seconds. The simulation time runs from three seconds to a stop time of 18 seconds. Notice the time offset notice in the lower right corner.

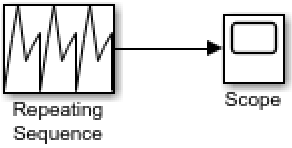


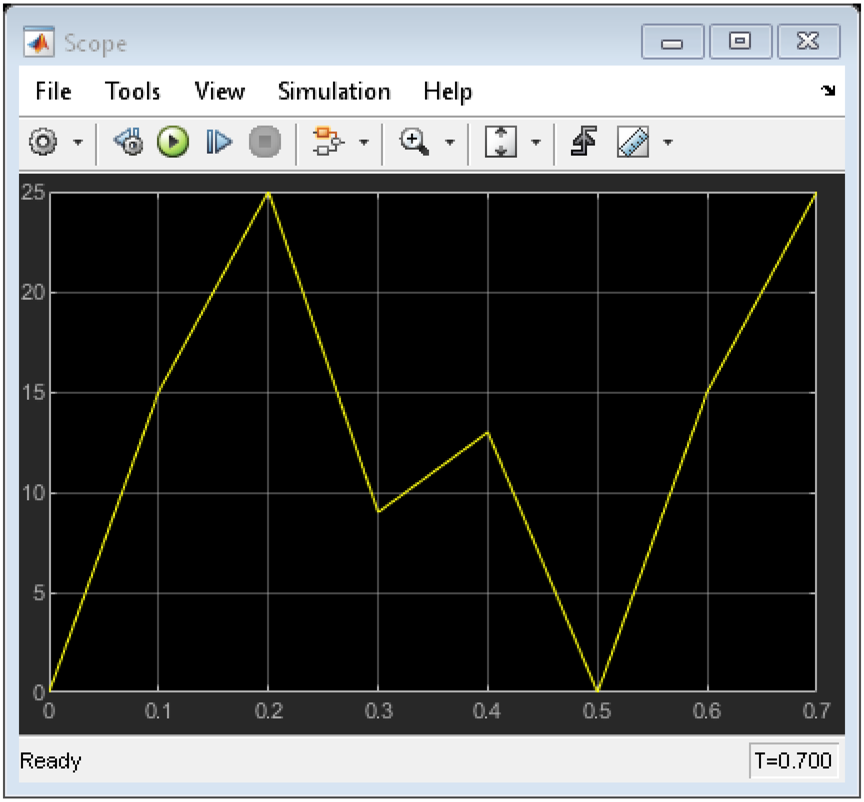


Notice that the time-based Pulse Generator produces an on signal for only two seconds and then switches to off. This is due to the block starting to compute the output from t=0 even though it does not output the simulation until t=3. The sample-based block outputs a pulse of five seconds on followed by five seconds off. In this case, the block output does not depend on simulation time and starts only when the simulation starts.

Specify a Waveform with the Repeating Sequence Block

This example shows how you specify a waveform with the Repeating Sequence block. In this model, the block defines the **Time values** parameter as `[0:0.1:0.5]` and the **Output values** parameter as `[0 15 25 09 13 17]`. The stop time of the simulation is `0.7` second.



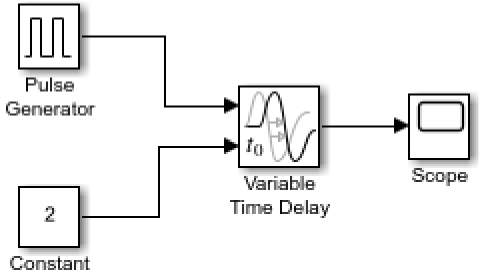


- Input period is `0.5`.
- Output at any time `t` is the output at time `t = t-0.5n`, where `n` = 0,1,2, and so on.
- Sequence repeats at `t = 0.5n`.

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** parameter vector `[0 15 25 09 13 17]` does not appear.

Tune Phase Delay on Pulse Generator During Simulation

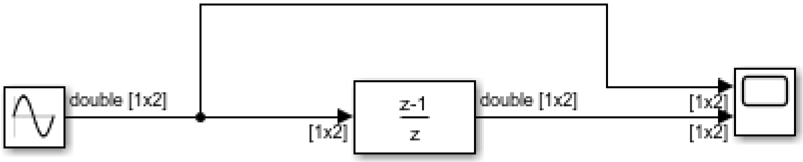
This example shows how to modify a model so that you can change a phase delay for a Pulse Generation block during simulation. You cannot tune the value of the **Phase delay** parameter during simulation. As a workaround, add a Constant block and a Variable Time Delay block.

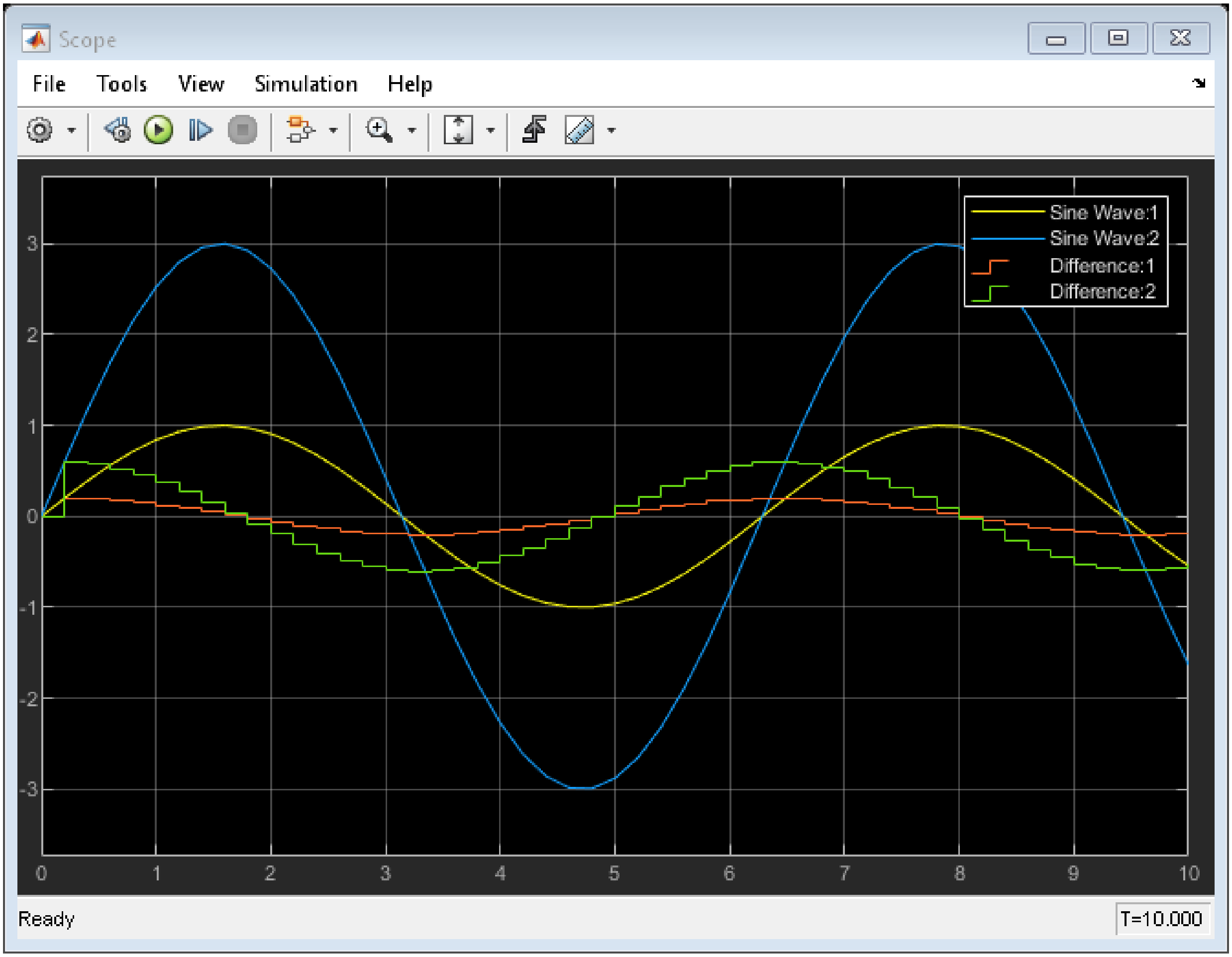


In the Pulse Generator block, set the value of the **Phase delay** parameter to zero. Use the Constant block to specify the delay time in seconds. To tune the delay time during simulation, change the value stored in the Constant block.

Difference Sine Wave Signal

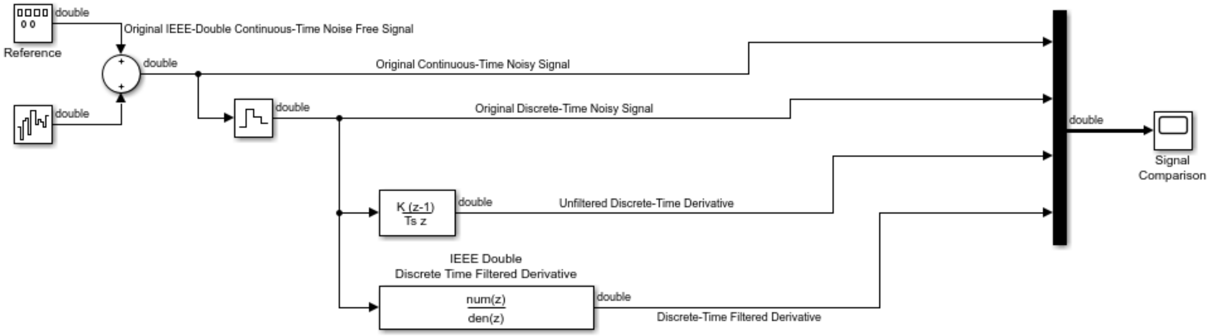
This example shows how to calculate the difference in a sine wave signal at each time step. The input is a 1-by-2 vector of sine waves, with amplitude 1 and 3. The difference block calculates the difference in each sine wave signal at every time step. The Scope block displays both the original sine waves and the output of the difference block.

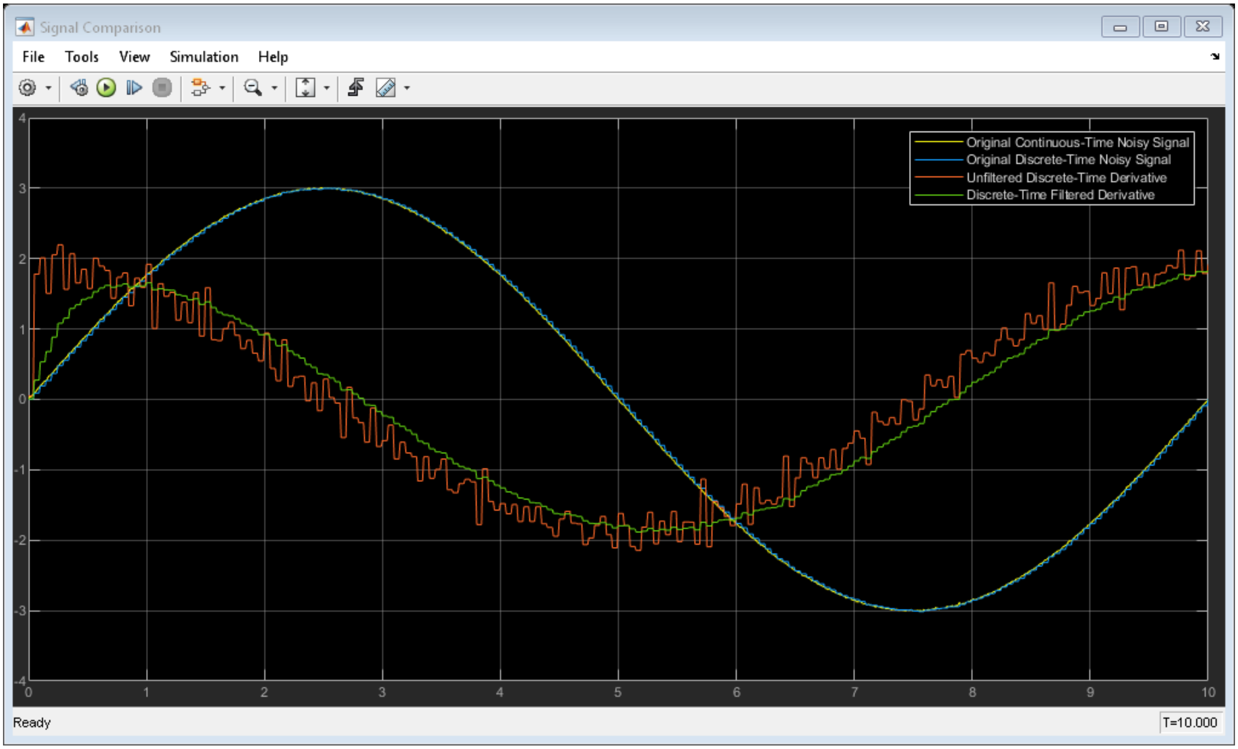



Scope
File
Tools
View
Simulation
Help
Sine Wave:1
Sine Wave:2
Difference:1
Difference:2
3
2
1
0
-1
-2
-3
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

Discrete-Time Derivative of Floating-Point Input

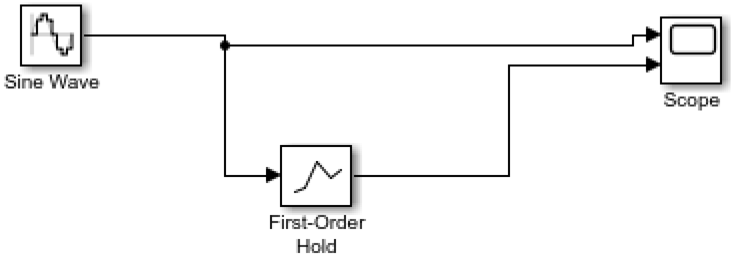
This example shows how to use the Discrete Derivative block to compute the discrete-time derivative of a floating-point input signal. The unfiltered discrete-time derivative is compared to a filtered discrete-time derivative that is computed by the Discrete Filter block.

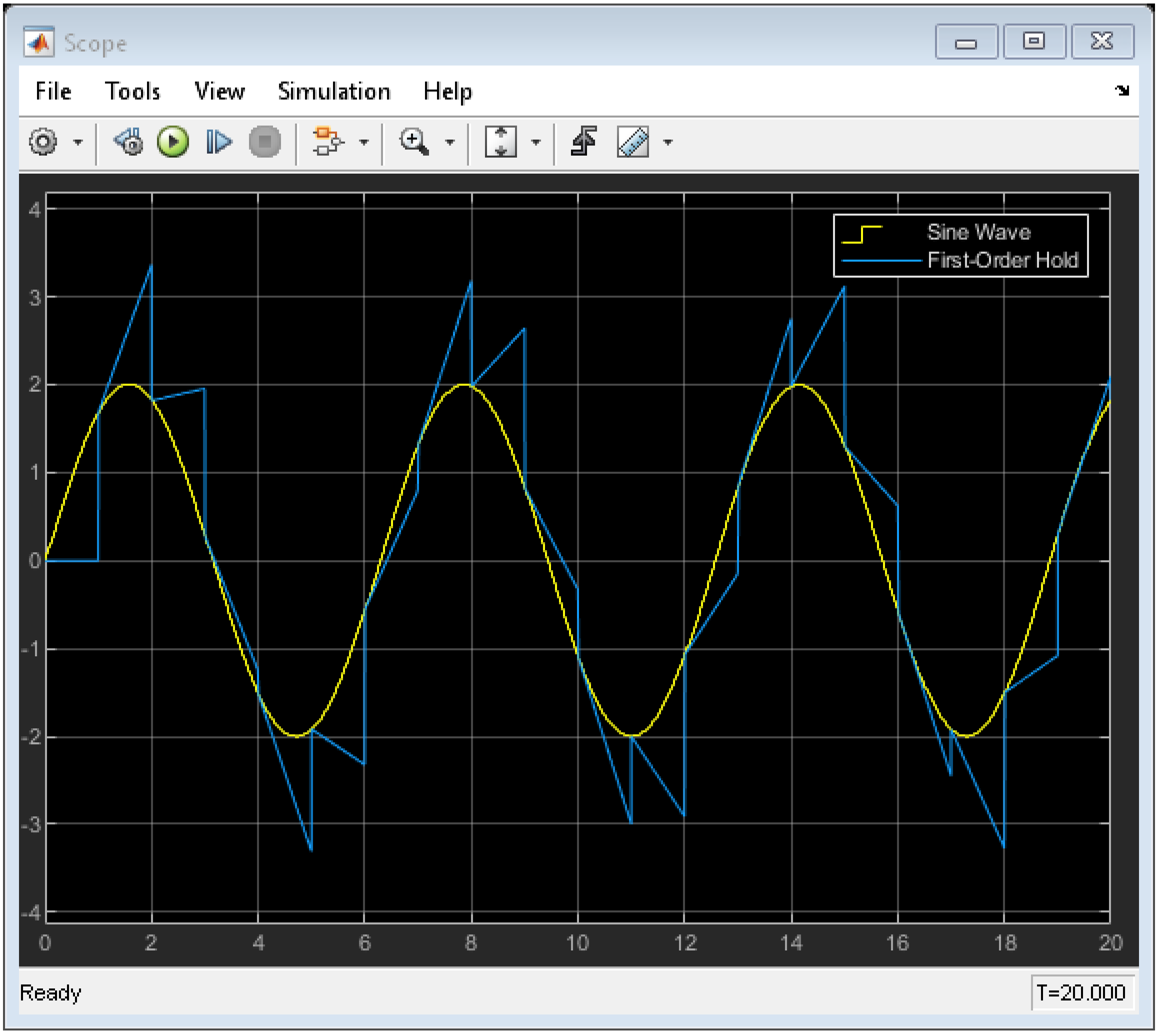

double
Reference
Original IEEE-Double Continuous-Time Noise Free Signal
+
+
double
Original Continuous-Time Noisy Signal
double
double
Original Discrete-Time Noisy Signal
double
Signal
Comparison
K (z-1)
Ts z
double
Unfiltered Discrete-Time Derivative
IEEE Double
Discrete Time Filtered Derivative
num(z)
den(z)
double
Discrete-Time Filtered Derivative


Signal Comparison
File
Tools
View
Simulation
Help
Original Continuous-Time Noisy Signal
Original Discrete-Time Noisy Signal
Unfiltered Discrete-Time Derivative
Discrete-Time Filtered Derivative
4
3
2
1
0
-1
-2
-3
-4
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

First-Order Sample-and-Hold of a Sine Wave

This example shows how to perform a first-order sample-and-hold of a sine wave signal using the First-Order Hold block.



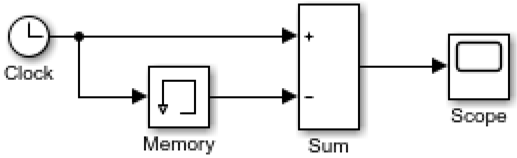


Ready

T=20.000

Calculate and Display Simulation Step Size using Memory and Clock Blocks

This example shows how to use the Memory and Clock blocks to calculate and display the step size in a simulation. The Sum block subtracts the time at the previous time step, which the Memory block generates, from the current time, which the Clock block generates.

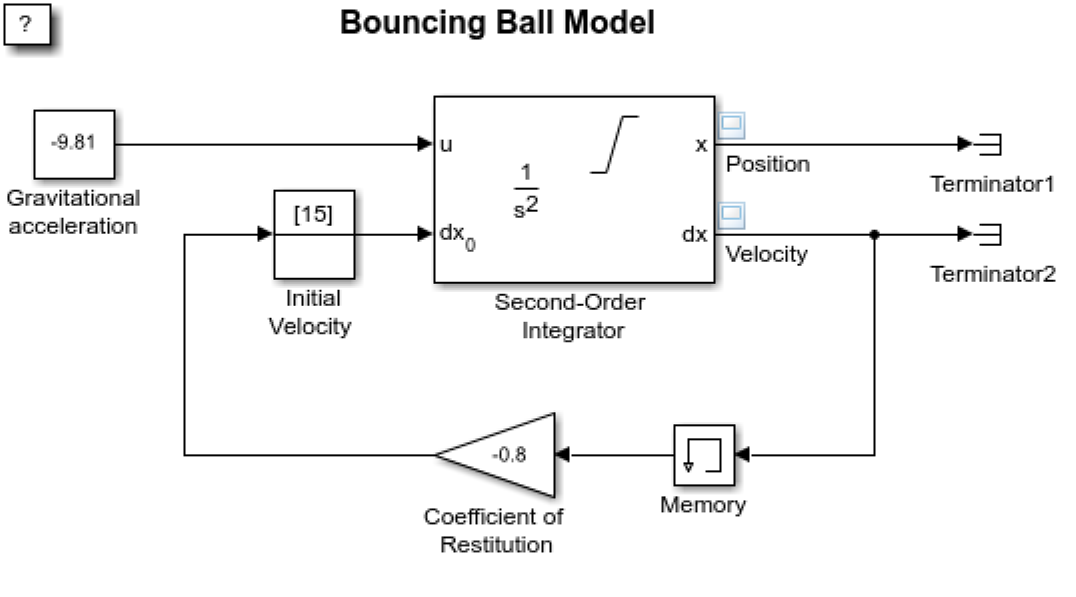


Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a fixed-step solver. Therefore, the sample time of the Memory block is the solver step size, or 1.

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.

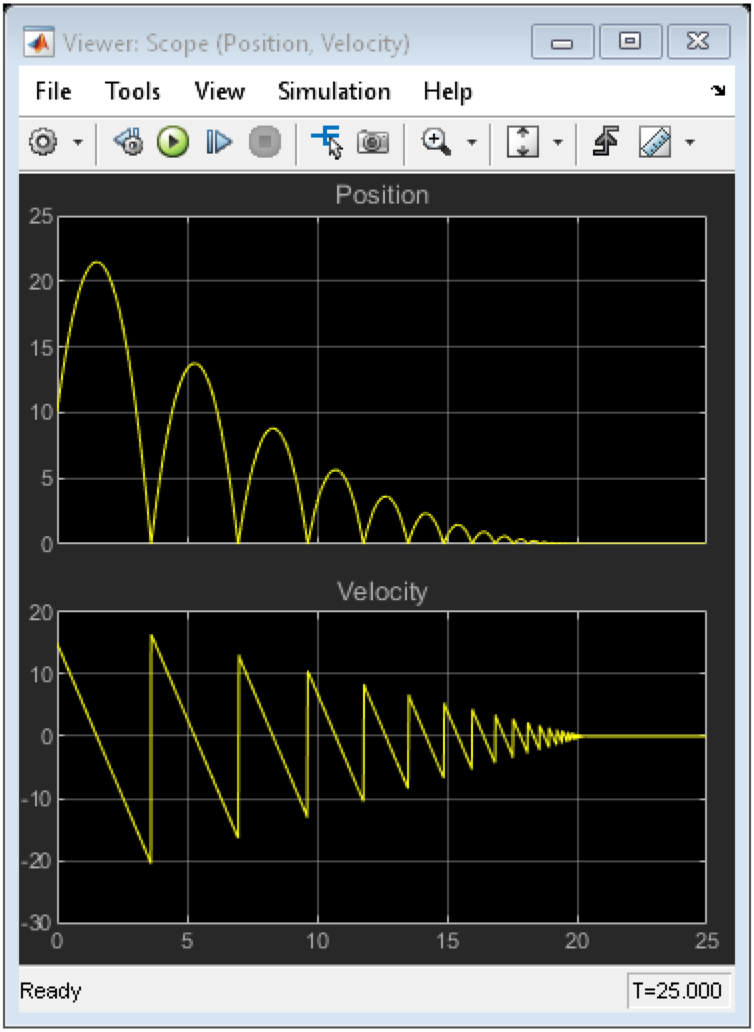
Capture the Velocity of a Bouncing Ball with the Memory Block

The `sldemo_bounce` example shows how to use the Second-Order Integrator and Memory blocks to capture the velocity of a bouncing ball just before it hits the ground.



Copyright 2004-2013 The MathWorks, Inc.

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:

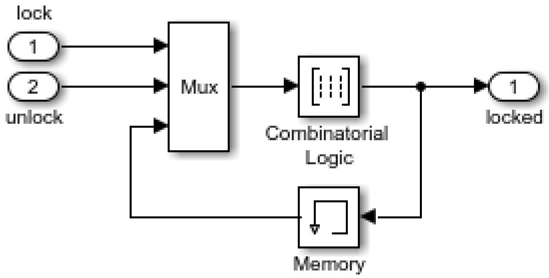


If you replace the Memory block with a Unit Delay block, you get the same results. However, a warning also appears due to the discrete Unit Delay block inheriting a continuous sample time.

For more information, see the model description.

Implement a Finite-State Machine with the Combinatorial Logic and Memory Blocks

The sldemo_clutch example 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:

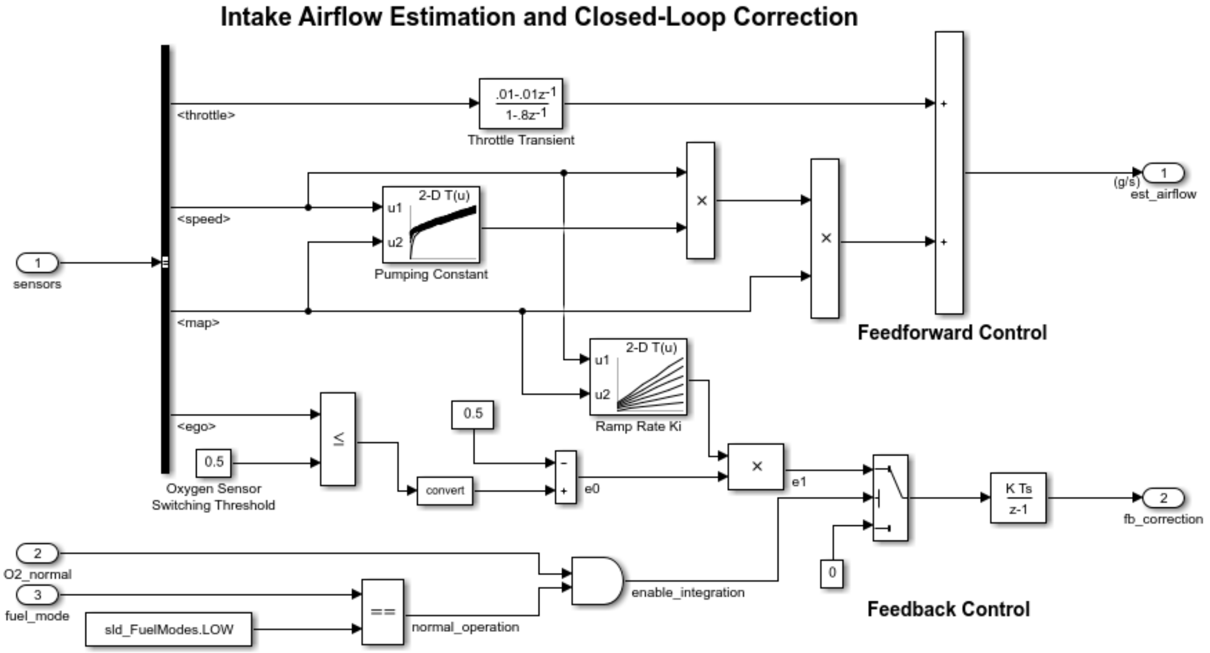


Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step (ode23) solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: [0, 1].

For more information, see the model description.

Discrete-Time Integration Using the Forward Euler Integration Method

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.

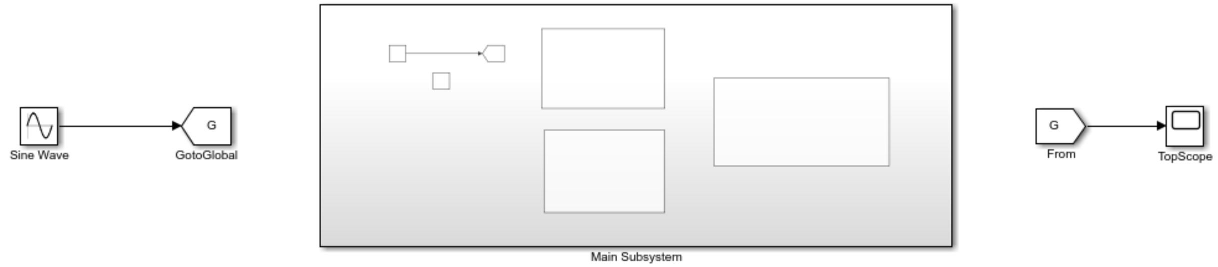


When the Switch block feeds a nonzero value into the Discrete-Time Integrator block, integration occurs. Otherwise, integration does not occur.

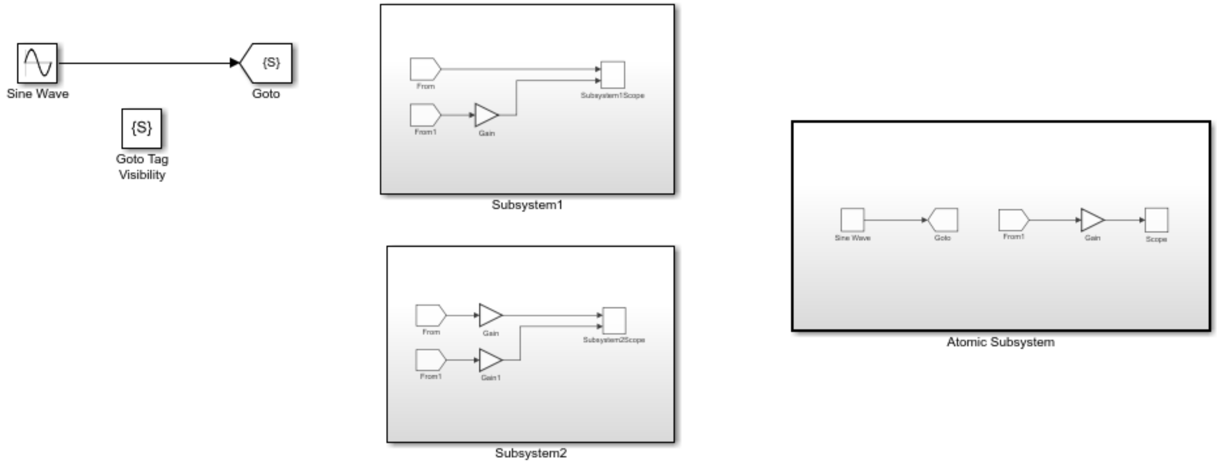
For more information, see the model description.

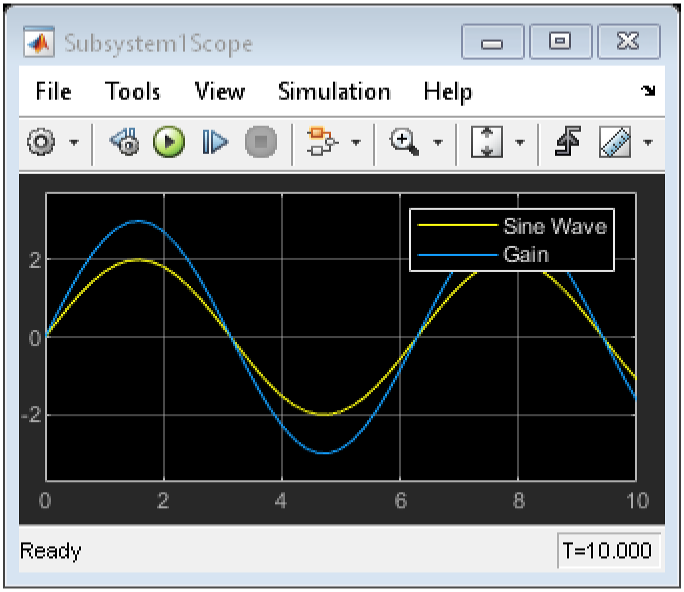
Signal Routing with the From, Goto, and Goto Tag Visibility Blocks

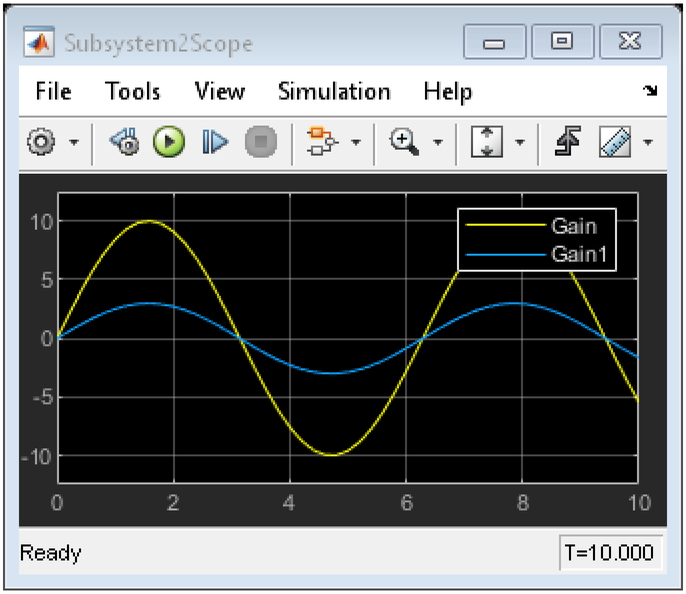
This example shows how to use the From, Goto, and Goto Tag Visibility blocks to route signals in your model. The GotoGlobal block at the top-level of the model has the **Goto tag** parameter set to `G` and the **Tag visibility** set to `global`. Thus, the `G` tag can be seen by From and Goto blocks at any level of the model hierarchy, except locations that span nonvirtual subsystem boundaries (like the Atomic Subsystem in this model). The From block at the top level of the model can see and connect to the global `G` tag, but cannot see or connect to the scoped `S` tag or `L` local tag that are specified on Goto blocks further down in the model hierarchy.



Inside of the Main Subsystem, the Goto block with **Goto tag** set to `S` has a **Tag visibility** of `scoped`. The Goto Tag Visibility block placed at the same level as that Goto block indicates the `S` tag can be seen by all From and Goto blocks at that level and below, except for locations that cross a nonvirtual subsystem boundary (i.e. the boundary with the Atomic Subsystem). Inside of Subsystem1 and Subsystem2, the From blocks can see and connect to the global Goto tag `G`, and the scoped Goto tag `S`.

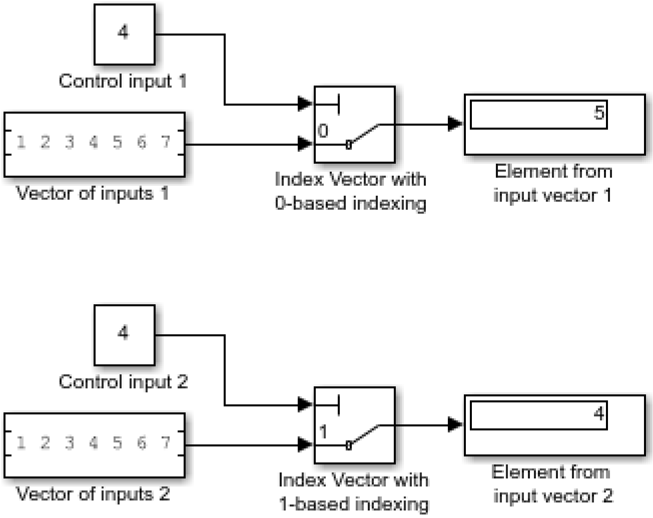

Sine Wave
{S}
Goto
{S}
Goto Tag Visibility
From
From1
Gain
Subsystem1Scope
Subsystem1
From
Gain
From1
Gain1
Subsystem2Scope
Subsystem2
Sine Wave
Goto
From1
Gain
Scope
Atomic Subsystem


Subsystem1Scope
File
Tools
View
Simulation
Help
Sine Wave
Gain
2
0
-2
0
2
4
6
8
10
Ready
T=10.000


Subsystem2Scope
File
Tools
View
Simulation
Help
10
5
0
-5
-10
0
2
4
6
8
10
Gain
Gain1
Ready
T=10.000

Zero-Based and One-Based Indexing with the Index Vector Block

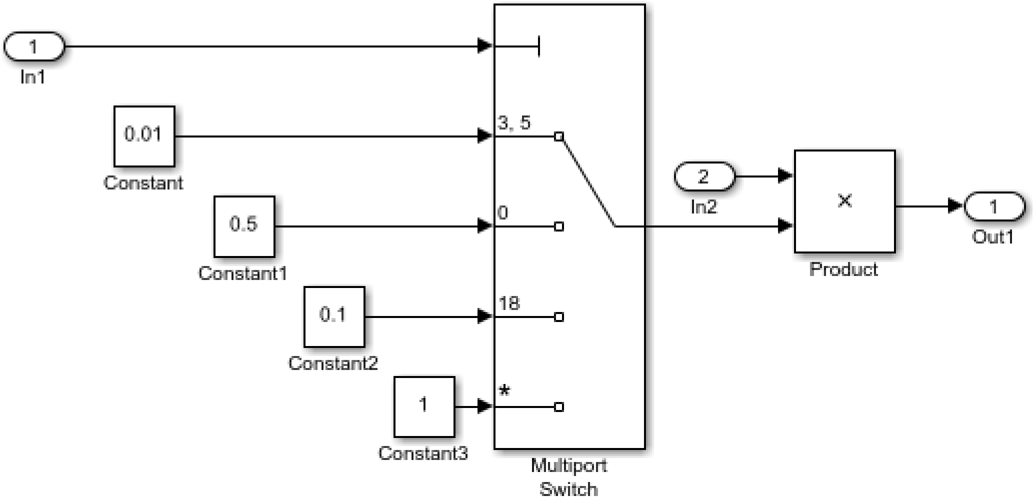
This example shows how the Index Vector block works with zero-based and one-based indexing.



The Index Vector block is from the Simulink Signal Routing library. It is a special configuration of the Multiport Switch block. To configure the Multiport Switch block as an Index Vector block, set the **Number of data ports** to `1` and **Data port order** to `Zero-based contiguous`.

Noncontiguous Values for Data Port Indices of Multiport Switch Block

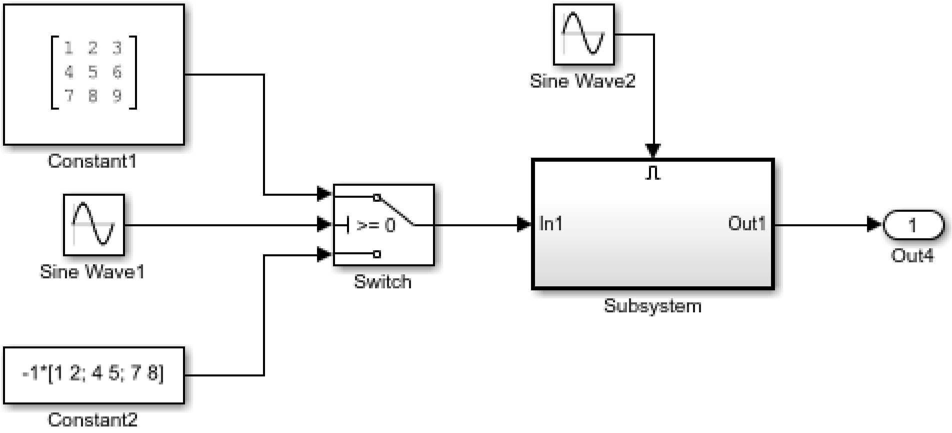
This example shows how to use a Multiport Switch block that specifies noncontiguous integer values for data ports. The values of the indices are visible on the data port labels. You do not have to open the block dialog box to determine which value maps to each data port.



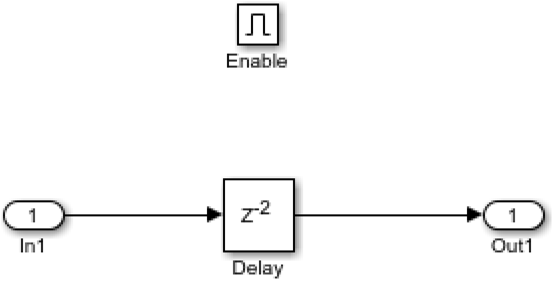
When you set **Data port for default case** to `Additional data port`, an extra port with a * label appears. This port corresponds to the default case, which applies when the control input does not match the data port indices 3, 5, 0, or 18. When that happens in this example, the Multiport Switch block outputs a value of `1`.

Using Variable-Size Signals on the Delay Block

This example shows how the Delay block supports vairable-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 these rules for variable-size signals while using sample-based processing.

- Signal dimensions change only during state reset when the block is enabled.

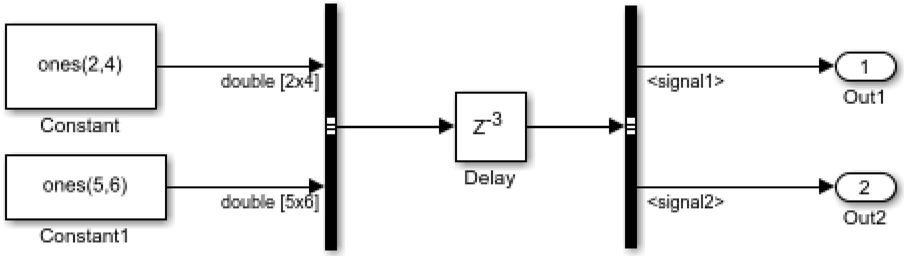
- Initial condition must be scalar.

The rules are implemented by these blocks.

- Enable block sets **Propagate sizes of variable-size signals** to `Only when enabling`.
- Delay block sets the **Initial condition** to the scalar value `0.0`.

Bus Signals with the Delay Block for Frame-Based Processing

This example 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 these rules for bus signals.

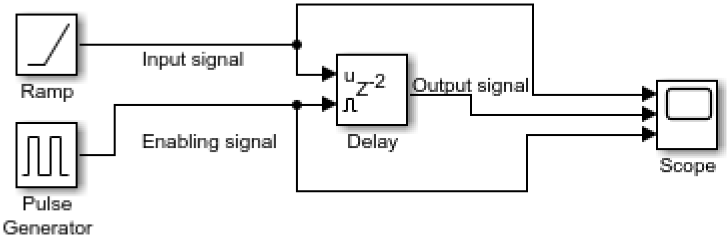
- For the initial condition, set the value on the dialog box.
- For frame-based processing, signal dimensions of the data input port `u` cannot be larger than two.

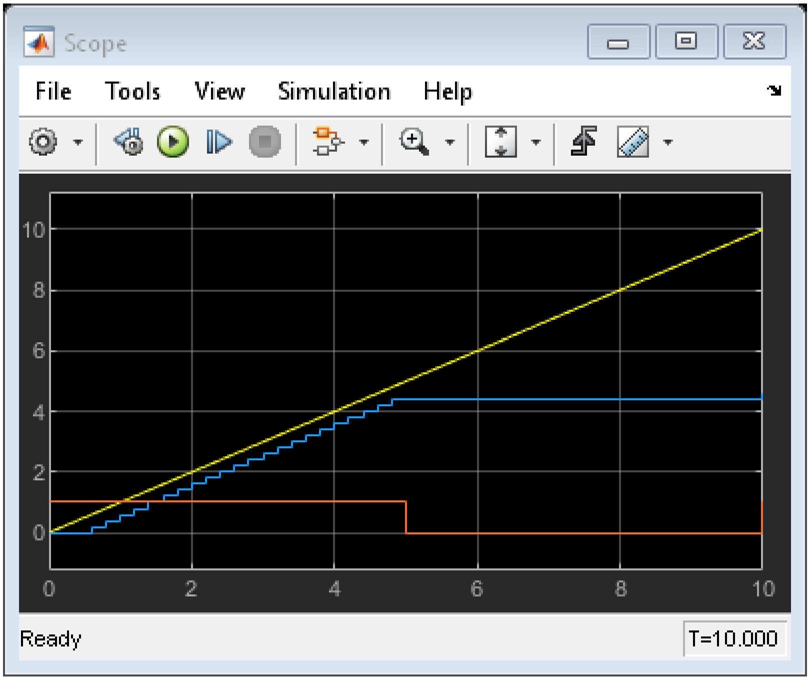
The model implements the rules by:

- Setting the **Initial condition** to a scalar value of `0`.
- Setting bus input to the Delay block as two dimensions.

Control Execution of Delay Block with Enable Port

This example shows how you can enable or disable the execution of the Delay block using an enable port. In this model, a ramp input signal feeds into a Delay block. The execution of the block is controlled by an enabling signal from the Pulse Generator block.

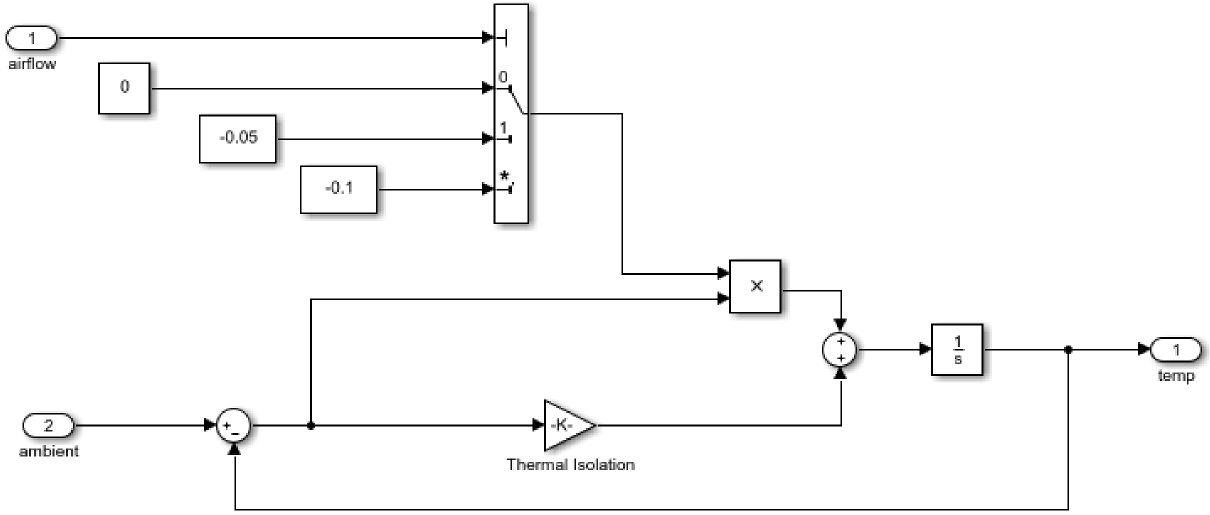




The blue line shows that the Delay block outputs the input signal value delayed by one time step while the enabling signal has a value of one. At `t=5` the enabling signal transitions to zero and the Delay block stops executing. The output is held at the last value until the block is enabled again.

Zero-Based Indexing for Multiport Switch 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.

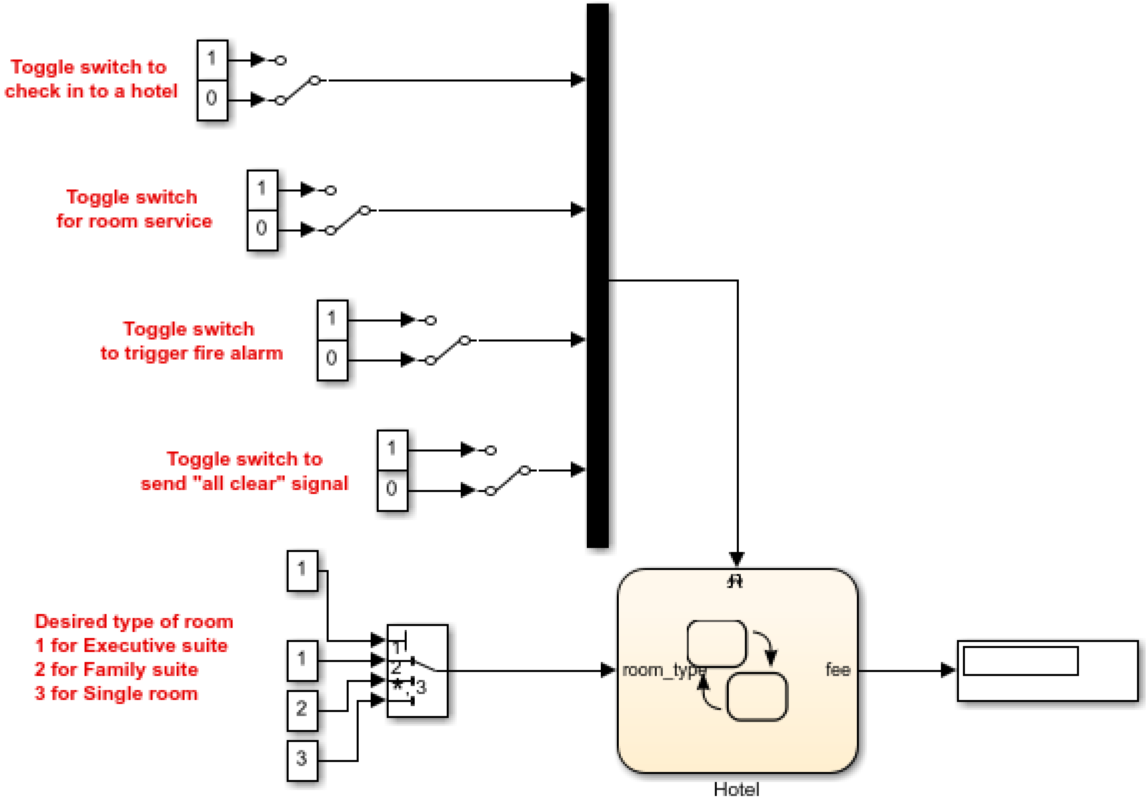


The indices are visible on the data port labels. You do not have to open the block dialog box to determine if the data ports use zero-based or one-based indexing.

When you set **Data port for default case** to `Last data port`, the last data port includes a `*` on the label (in this case, the label is `*,2`). 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. In this example, the Multiport Switch block outputs a value of `-0.1` when the control input does not match the data port indices of 0, 1, or 2.

One-Based Indexing for Multiport Switch 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.

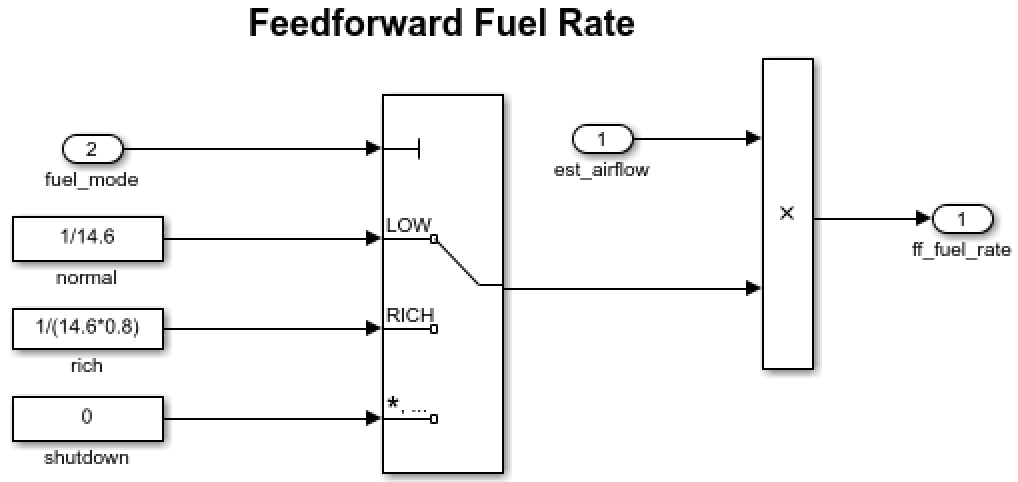


Copyright 2008-2013 The MathWorks, Inc.

When 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 of the Multiport Switch Block

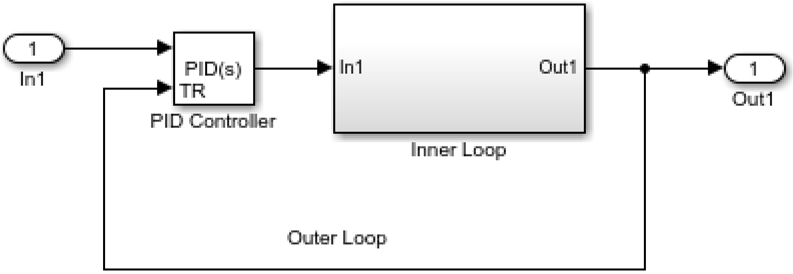
The sldemo_fuelsys model uses a Multiport Switch block in the `fuel_rate_control/fuel_calc/feedforward_fuel_rate` subsystem. This block uses the enumerated type `sld_FuelModes` to specify three data port indices: `LOW`, `RICH`, and `DISABLED`.



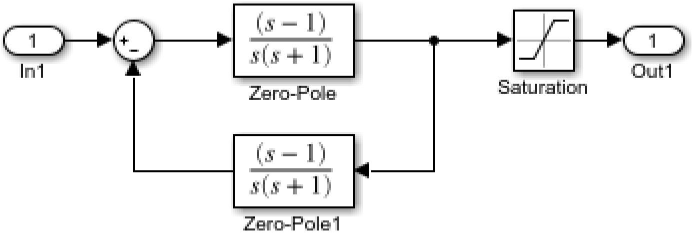
When you set **Data port for default case** to `Last data port`, the last data port includes a * on the label. The comma and ellipsis after the * indicate that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices `LOW`, `RICH`, or `DISABLED`. In this case, the Multiport Switch block outputs a value of `0`.

Prevent Block Windup in Multiloop Control

This example shows how to use signal tracking to prevent block windup in a two-loop control system.



The Inner Loop subsystem contains a saturation limit:

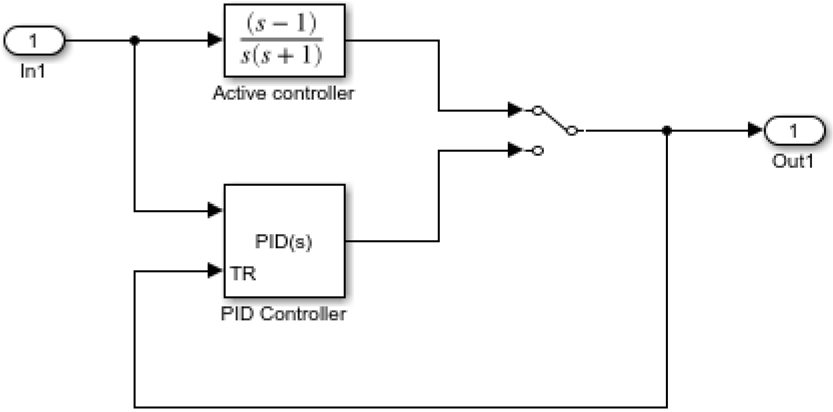


In this example, the inner loop has an effective gain of 1 when it does not saturate. When the inner loop does saturate, however, the integrator in the PID Controller can begin to wind up.

If the PID controller tracks the output of the inner loop, then its output never exceeds the saturated inner-loop output. To achieve this tracking, connect the Saturation block output to the tracking input of the PID Controller.

Bumpless Control Transfer

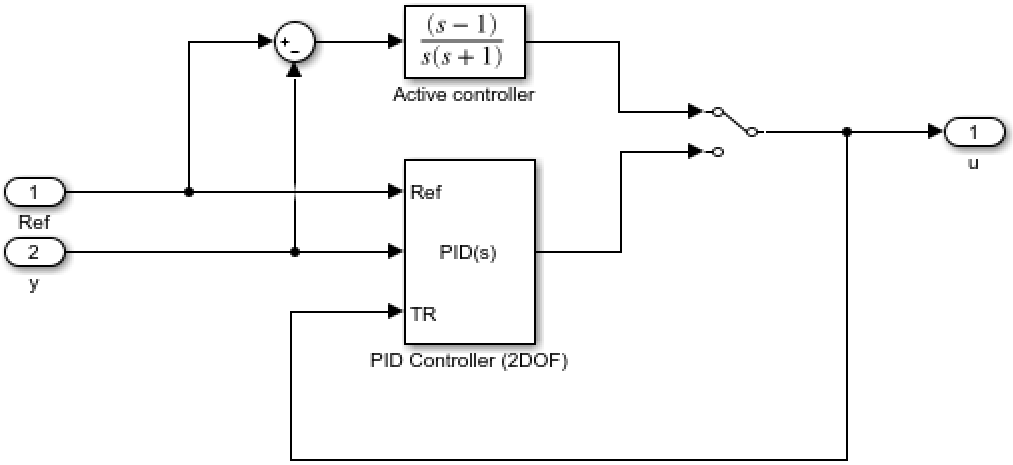
This example shows how to use signal tracking to achieve bumpless control transfer in a system that switches between a PID Controller block and another controller. You can make the PID controller track the output of the other controller by connecting the signal you want to track to the TR port of the PID Controller block. For example:



The input In1 takes a reference signal that feeds both controllers. The output Out1 drives a controlled system (not shown). A switch transfers control between the Active controller block (a Zero-Pole block) and the PID Controller block. When the Active controller is in the loop, the PID Controller block tracks its output. Thus, the two controllers have the same output when you switch to PID control, ensuring smooth operation.

Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller

This example shows how to use signal tracking to achieve bumpless control transfer in a system that switches between a PID Controller (2DOF) block and another controller. You can make the PID controller track the output of the other controller by connecting the signal you want to track to the TR port of the PID Controller block. For example:



The input Ref takes a reference signal, and the input y takes the measured feedback from the plant. These signals feed the 2DOF controller, and the difference between them feeds the Active controller block (a Zero-Pole block). The output u drives the plant. A switch transfers control between the active controller and the PID Controller (2DOF) block. When the Active controller is in the loop, the PID Controller (2DOF) block tracks its output. Thus, the two controllers have the same output when you switch to PID control, ensuring smooth operation.

Using a Bit Set block

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].

Using a Bit Clear block

If the Bit Clear block is turned on for bit 2, bit 2 is set to 0.

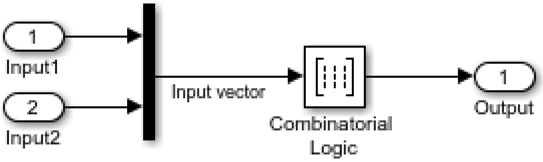
A vector of constants 2.^[0 1 2 3 4] is represented in binary as [00001 00010 00100 01000 10000].

With bit 2 set to 1, the result is [00101 00110 00100 01100 10100], which is represented in decimal as [5 6 4 12 20].

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]

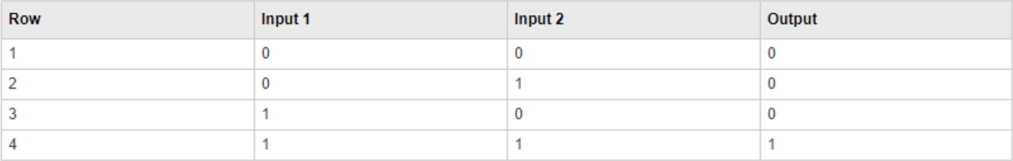
Two-Input AND Logic

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` 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:

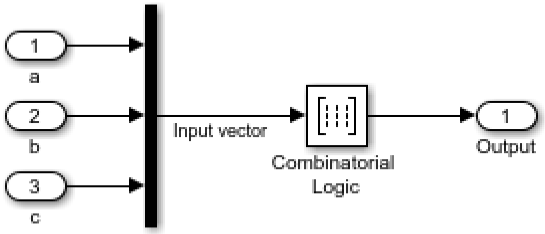
(2^1*1 + 1) The output value is 0.



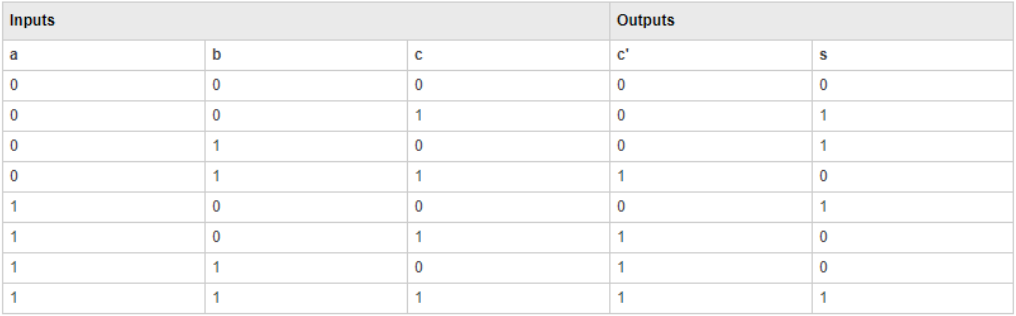
Row	Input 1	Input 2	Output
1	0	0	0
2	0	1	0
3	1	0	0
4	1	1	1

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.

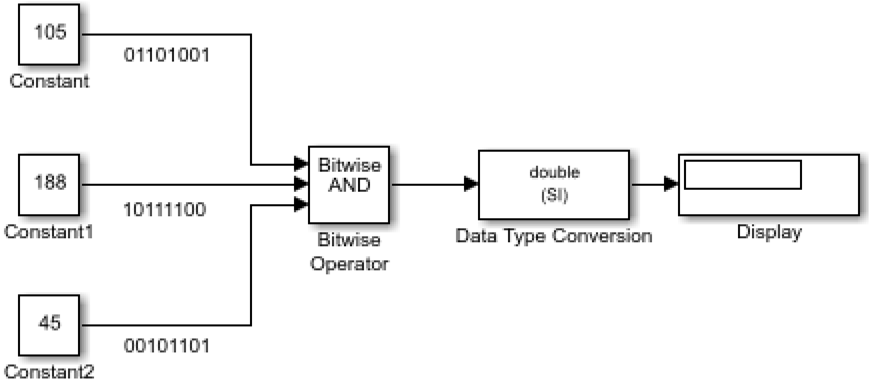


Inputs			Outputs	
a	b	c	c'	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

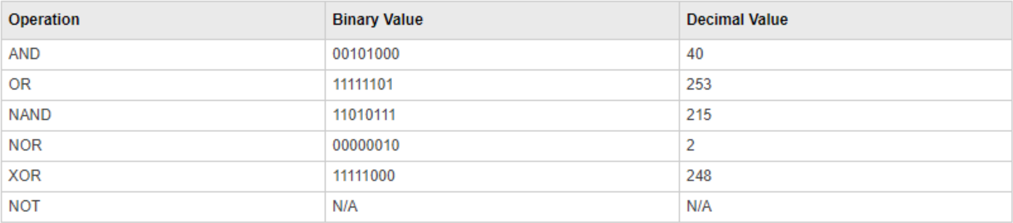
To implement this adder with the Combinatorial Logic block, you enter the 8-by-2 matrix formed by columns **c'** and **s** as the **Truth table** parameter. You can also implement sequential circuits (that is, circuits with states) with the Combinatorial Logic block by including an additional input for the state of the block and feeding the output of the block back into this state input.

Unsigned Inputs for the Bitwise Operator Block

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



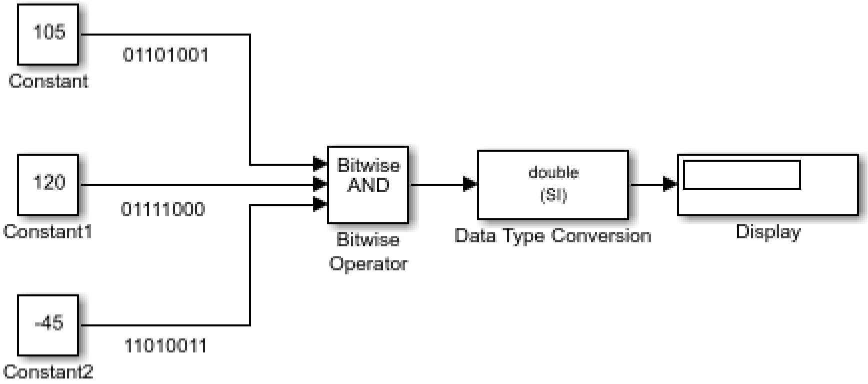
Each Constant block outputs an 8-bit unsigned integer (uint8). To determine the binary value of each Constant block output, use the dec2bin function. The results for all logic operations appear in the next table.



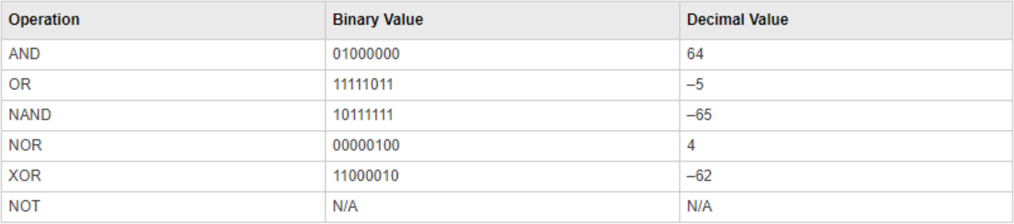
Operation	Binary Value	Decimal Value
AND	00101000	40
OR	11111101	253
NAND	11010111	215
NOR	00000010	2
XOR	11111000	248
NOT	N/A	N/A

Signed Inputs for the Bitwise Operator Block

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



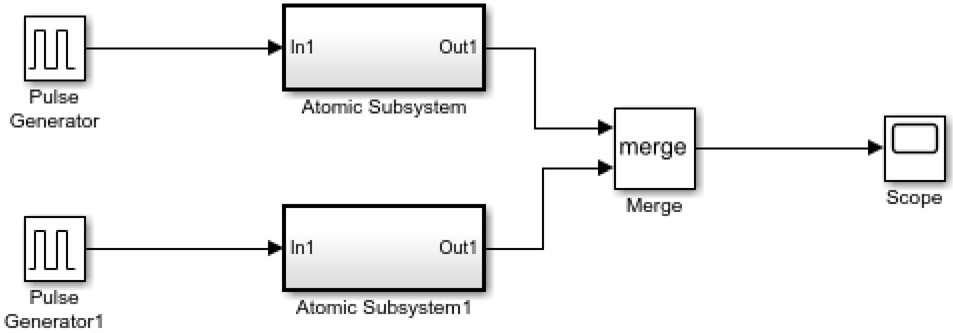
Each Constant block outputs an 8-bit signed integer (`int8`) . To determine the binary value of each Constant block output, use the dec2bin function. The results for all logic operations appear in the next table.



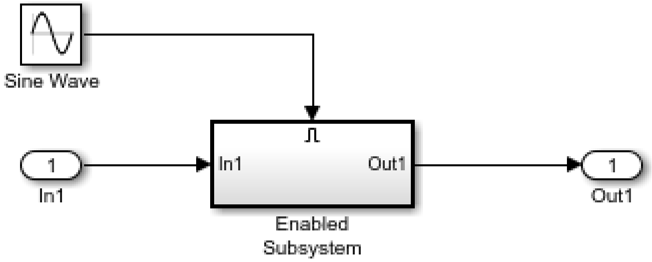
Operation	Binary Value	Decimal Value
AND	01000000	64
OR	11111011	–5
NAND	10111111	–65
NOR	00000100	4
XOR	11000010	–62
NOT	N/A	N/A

Merge Block with Input from Atomic Subsystems

This example shows a Merge block with inputs from two atomic subsystems.

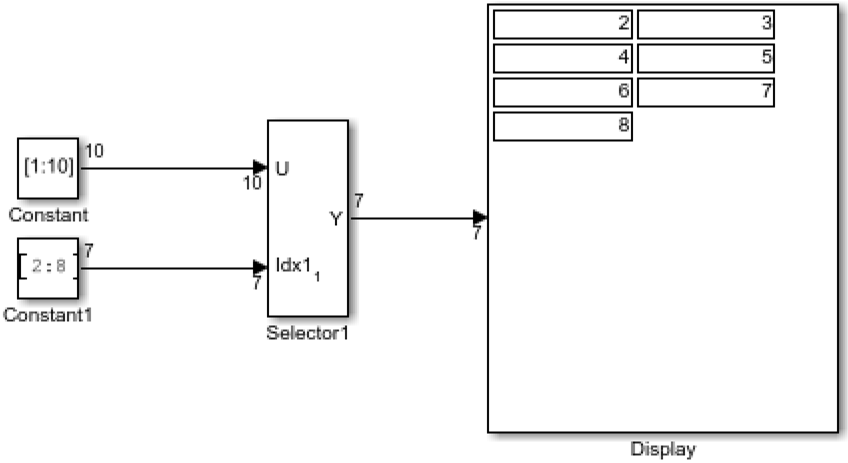


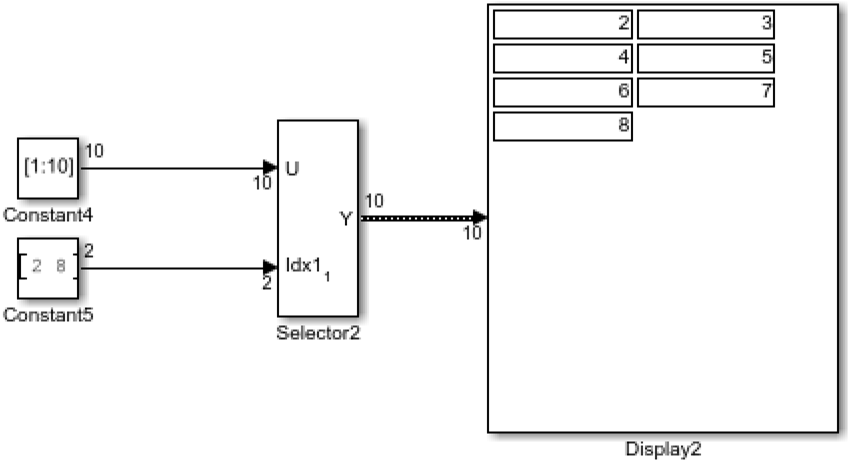
Each Atomic Subsystem block contains an enabled subsystem. This satisfies the requirement that inputs to a Merge block are from a conditionally executed subsystem.



Index Options with the Selector Block

This example shows two Selector blocks with the same kind of input signals, but two different **Index Option** settings.





Both Selector blocks select 7 values from the input signal that feeds the input port. The `Selector1` block outputs a fixed-size signal, whereas the `Selector2` block outputs a variable-size signal whose compiled signal dimension is 10 instead of 7.

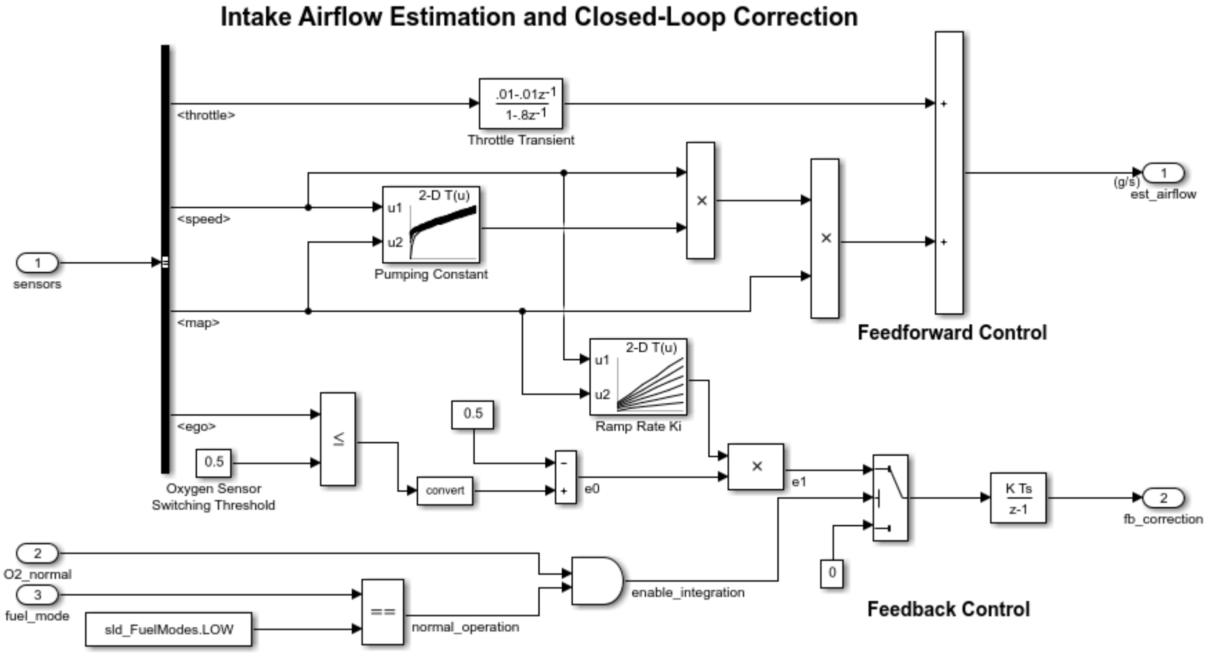
The `Selector1` block sets **Index Option** to `Index vector (port)`, which uses the input signal from `Constant1` as the index vector. The dimension of the input signal is 7, so the `Display` block shows the 7 values of the `Constant1` block. The `Selector2` block sets the **Input port size** parameter to 10, which is the size of the largest input signal to the `Selector2` block.

The `Selector2` block also sets the **Index Option** to `Starting and ending indices (port)`. The output is then set to the size of **Input port size** parameter (10), even though the size of the input signal is 7.

Switch Block with a Boolean Control Port Example

This example shows a Switch block with a Boolean input for the control port.

```
open_system('sldemo_fuelsys');
open_system('sldemo_fuelsys/fuel_rate_control/airflow_calc');
```



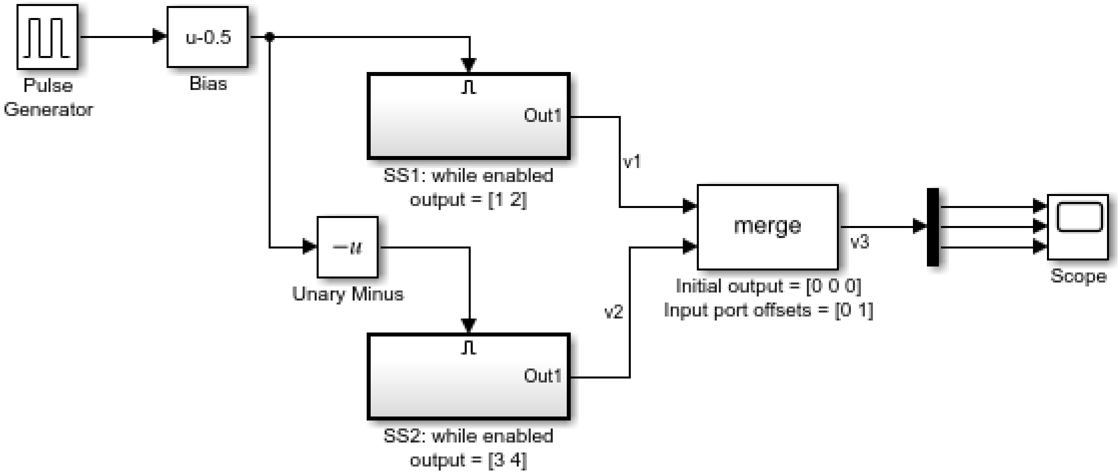
The value of the control port on the Switch block determines whether or not the feedback correction occurs. The control port value depends on the output of the Logical Operator block. When the Logical Operator block out is `true`, then the Switch block control port is 1 and the feedback control occurs. If the Logical Operator block output is `false` then the feedback control does not occur.

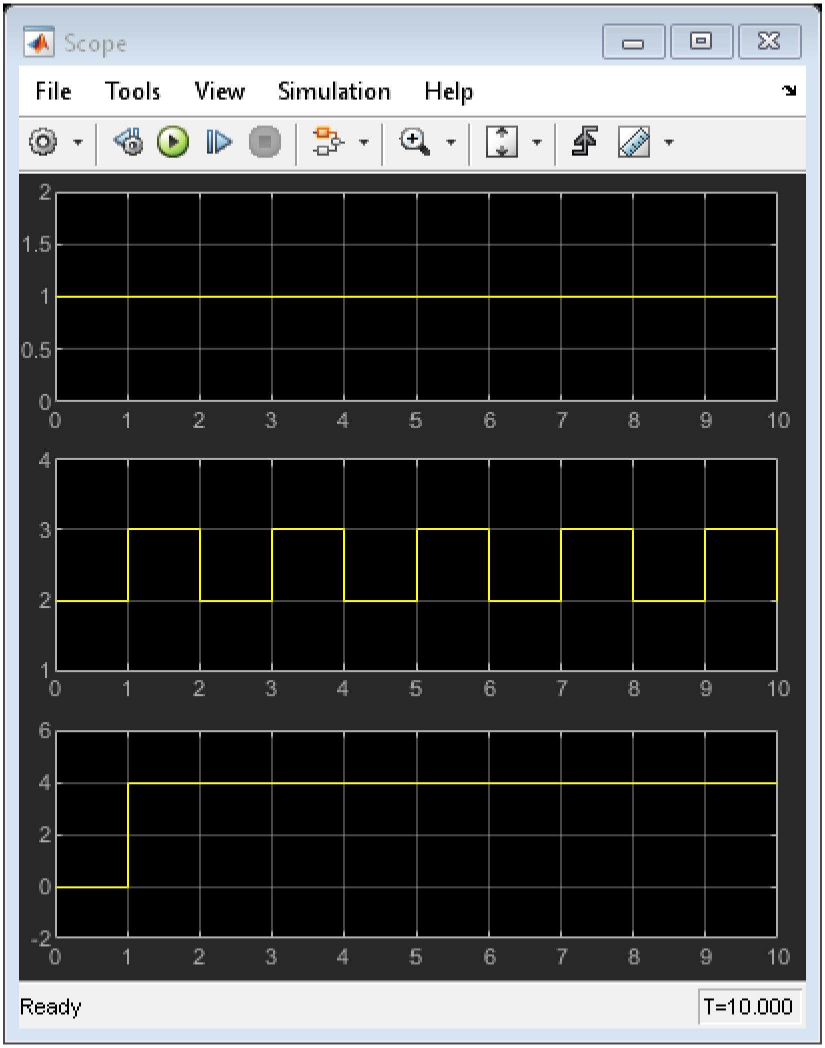
Merge Block with Unequal Input Widths Example

This example shows how to use the Merge block with inputs ports that have different widths. If you select `Allow unequal port widths`, the block accepts scalar and vector inputs having differing numbers of elements. You can specify an offset for each input signal relative to the beginning of the output signal. The width of the output signal is:

$$max(w1 + o1, w2 + o2, ...wn + on)$$

where wn are the widths of the input signals, and on are the offsets.


Pulse Generator
u-0.5
Bias
Out1
SS1: while enabled output = [1 2]
v1
−u
Unary Minus
merge
v3
Initial output = [0 0 0]
Input port offsets = [0 1]
Scope
v2
Out1
SS2: while enabled output = [3 4]



The Merge block has the following output width.

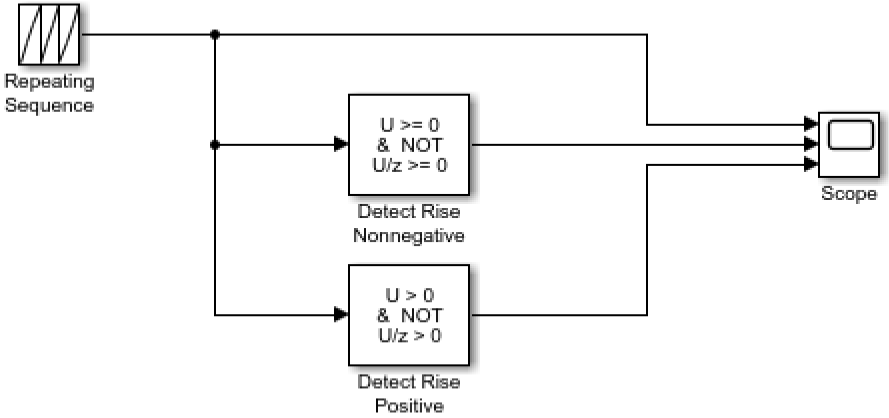
$$max(2+0,2+1)=3$$

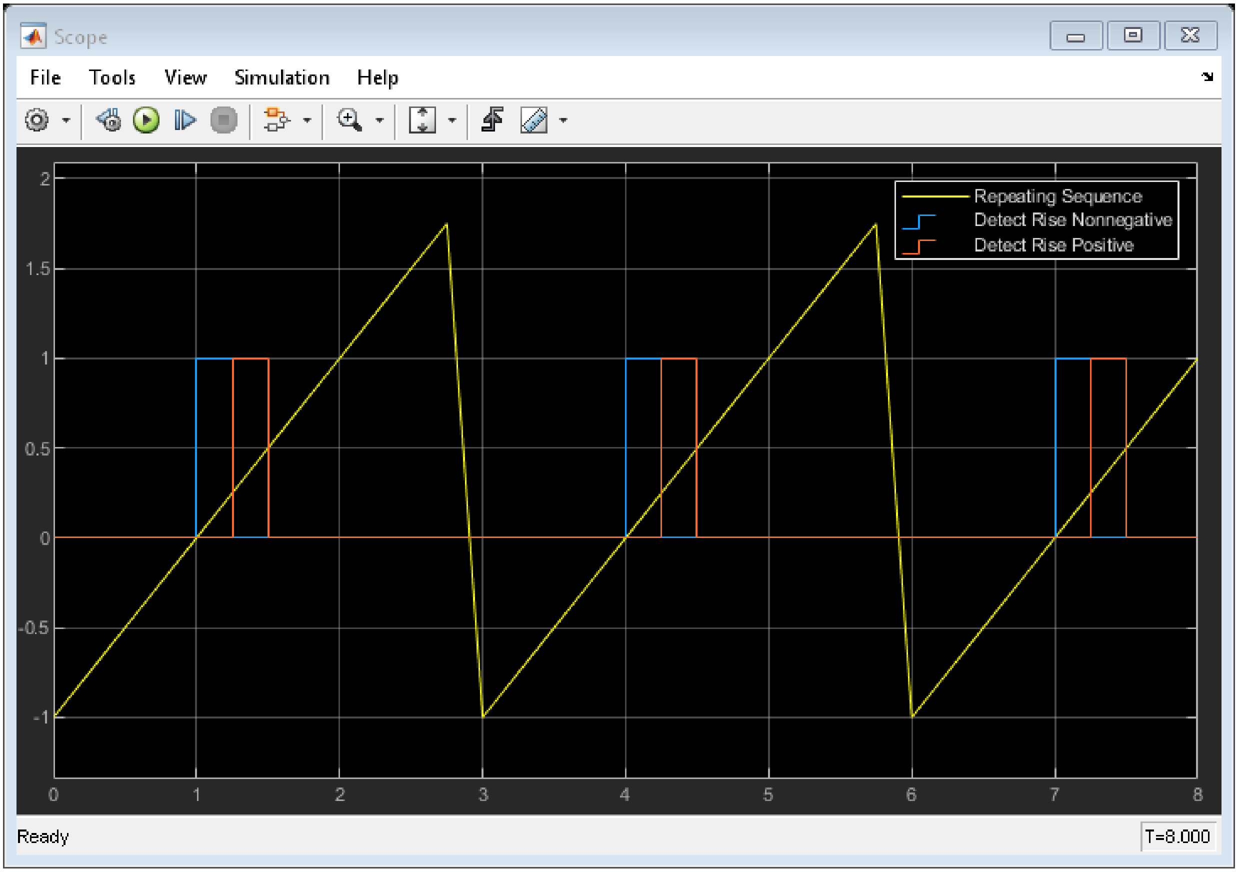
In this example, the offset of $v1$ is `0` and the offset of $v2$ is `1`. The Merge block maps the elements of $v1$ to the first two elements of $v3$ and the elements of $v2$ to the last two elements of $v3$. Only the second element of $v3$ is effectively merged, as show in the scope output.

If you use Simplified Initialization Mode, you must clear the `Allow unequal port widths` check box. The input port offsets for all signals must be zero.

Detect Rising Edge of Signals

This example shows how to detect the rising edge of a signal using the Detect Rise Nonnegative and Detect Rise Positive blocks.

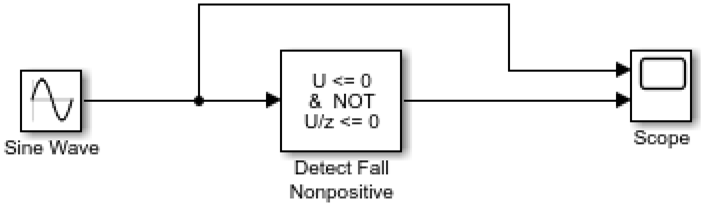


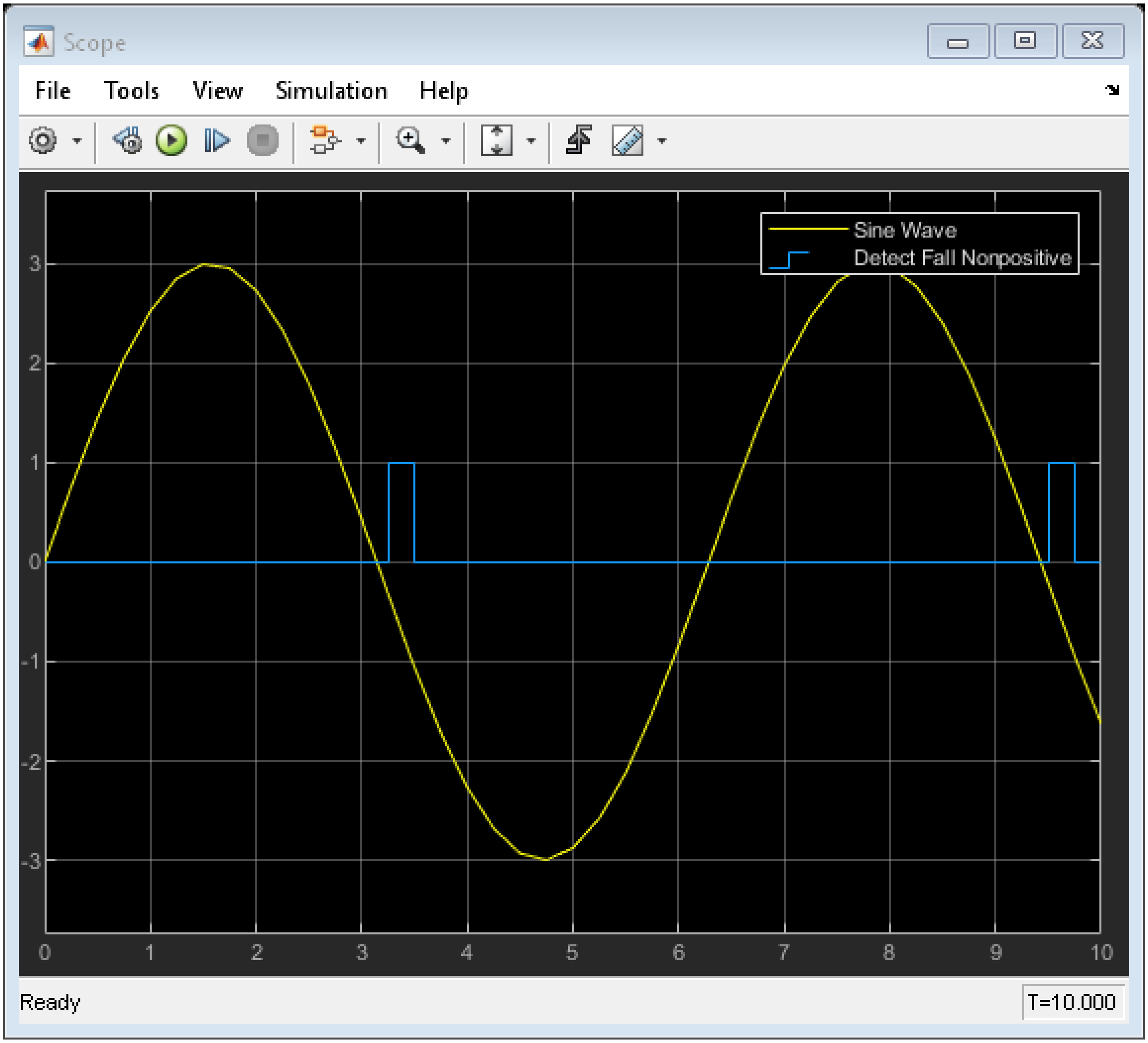


With a fixed-step size of 0.25, this example illustrates the difference between the Detect Rise Nonnegative and Detect Rise Positive blocks. The Detect Rise Nonnegative block outputs true (`1`) at `t=1` because the input signal increased from a negative value to a nonnegative value (`0`). The Detect Rise Positive block outputs true (`1`) at `t=1.25` because the input signal increased from a nonpositive value (`0`) to a strictly positive value.

Detect Falling Edge Using the Detect Fall Nonpositive Block

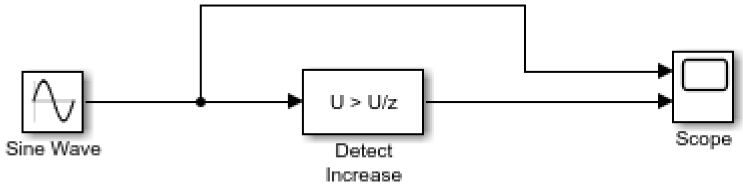
This example shows how to use the Detect Fall Nonpositive block to detect a falling edge in the input signal. The block detects a falling edge when the signal value decreases from a strictly positive value to a nonpositive value. In this example, the **Initial condition** of the Detect Fall Nonpositive block is set to `1`. This means that the Boolean expression `U/z <= 0` evaluates to true and the block assumes the initial value of the input signal is nonpositive.

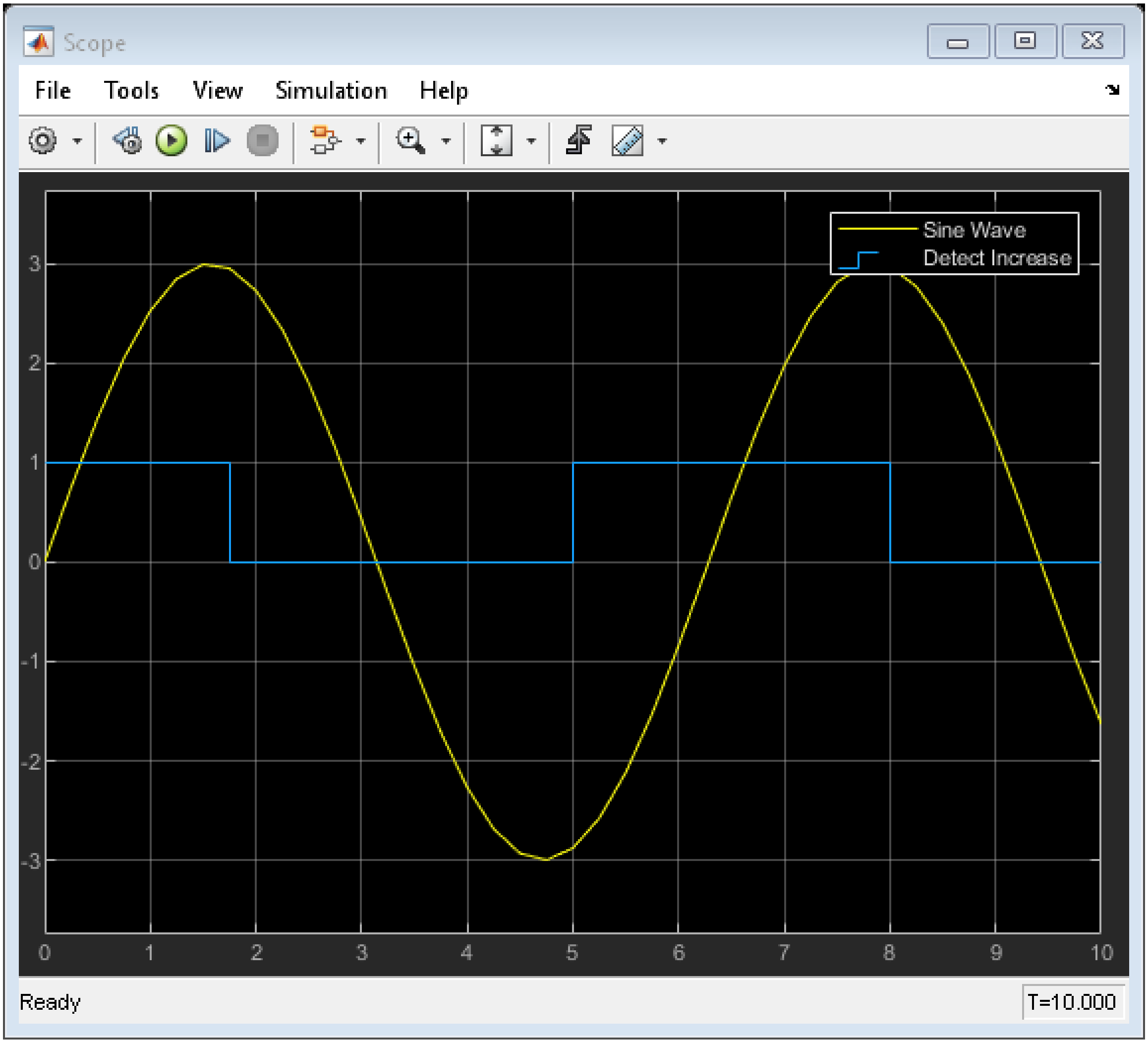



Scope
File
Tools
View
Simulation
Help
Sine Wave
Detect Fall Nonpositive
3
2
1
0
-1
-2
-3
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

Detect Increasing Signal Values with the Detect Increase Block

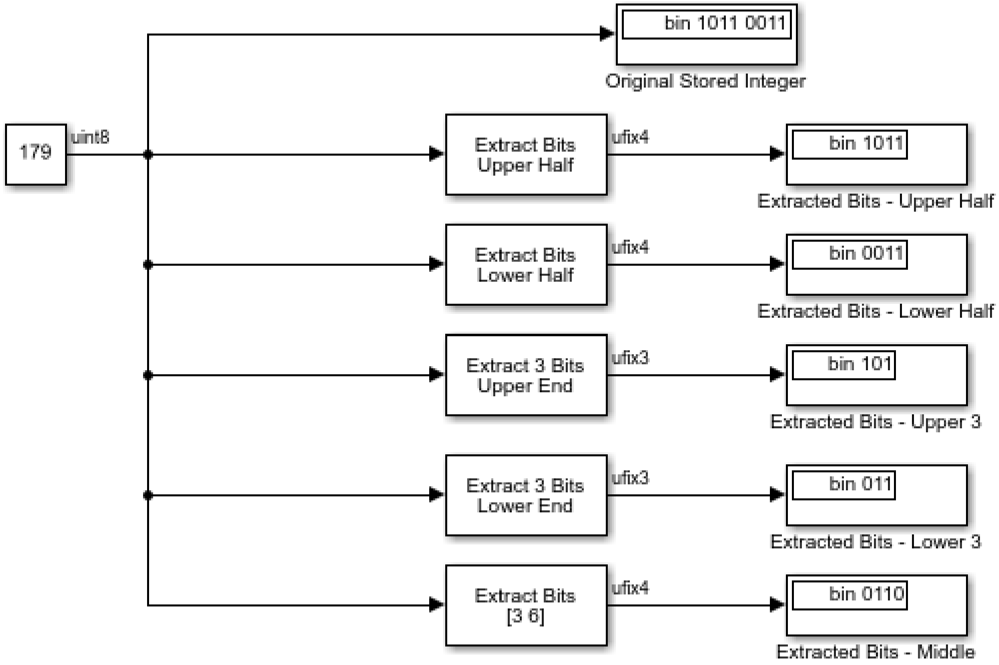
This example shows how to use the Detect Increase Block to detect increasing signal values. Because the **Initial condition** is set to `-1`, the block detects an increasing signal value starting at time `t=0`. If you change the **Initial condition** parameter to a nonnegative value, the block detects the first increasing signal value at `t=0.25`.




Scope
File
Tools
View
Simulation
Help
Sine Wave
Detect Increase
3
2
1
0
-1
-2
-3
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

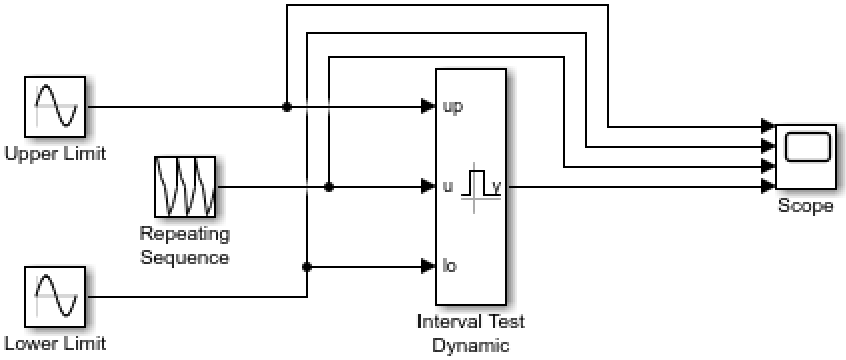
Extract Bits from Stored Integer Value

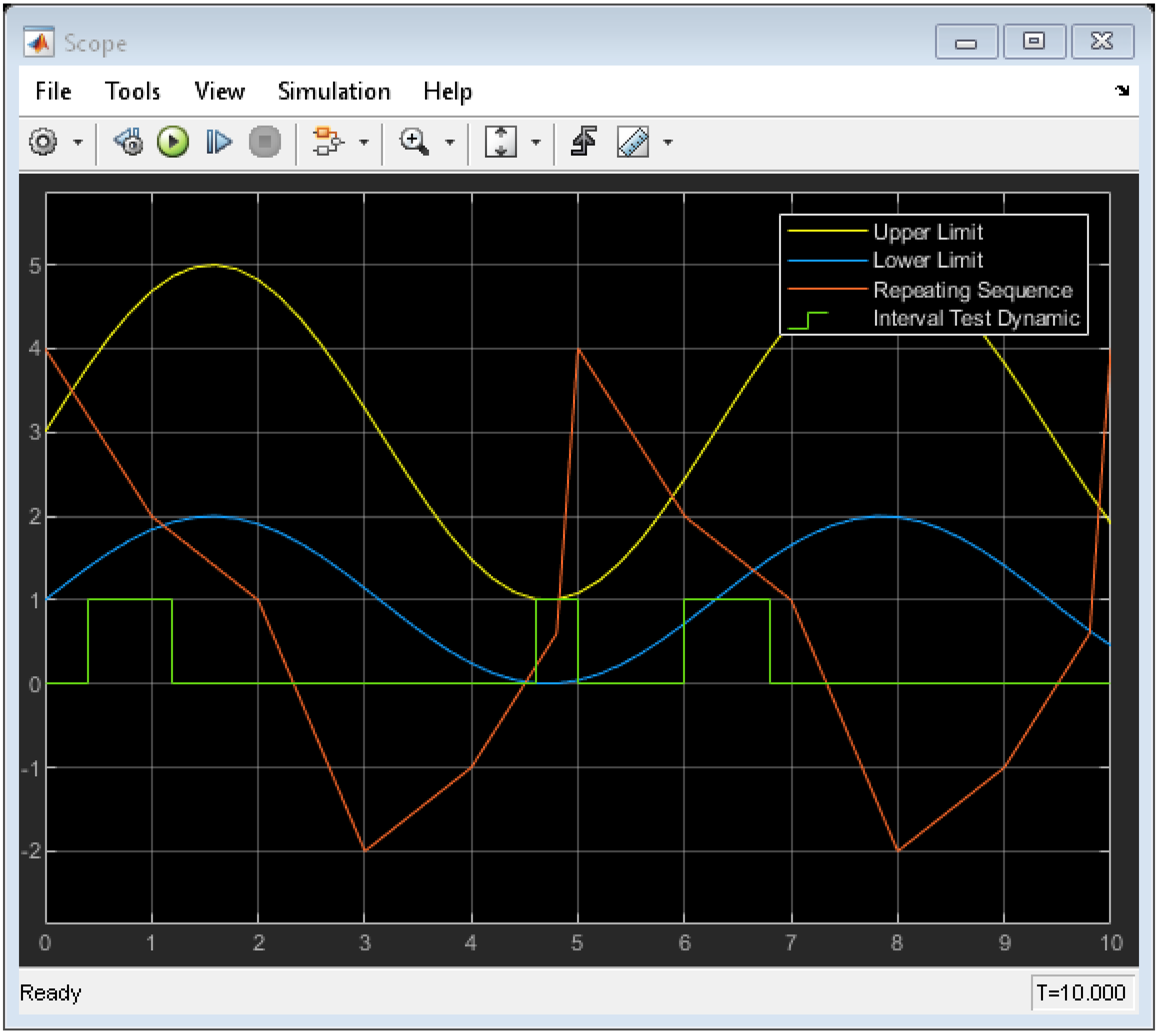
This example shows how to extract specific bits from the stored integer value of an input signal.



Detect Signal Values Within a Dynamically Specified Interval

This example shows how to detect when an input signal falls within a dynamically specified interval. The interval is defined by two Sine Wave blocks. When the input to the Interval Test Dynamic block falls between those sine waves, the Interval Test Dynamic block outputs true (1).

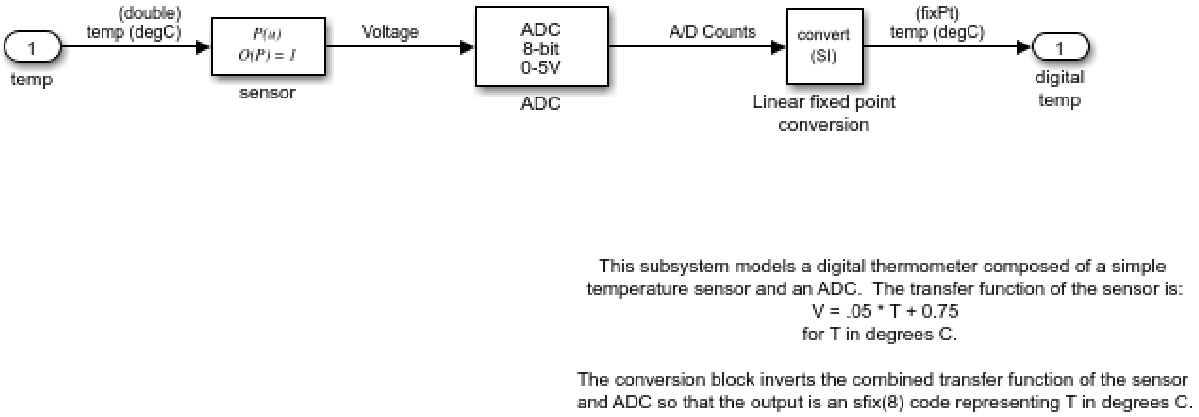



Scope
File
Tools
View
Simulation
Help
Upper Limit
Lower Limit
Repeating Sequence
Interval Test Dynamic
5
4
3
2
1
0
-1
-2
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

Model a Digital Thermometer Using the Polynomial Block

This example shows how the `ex_sldemo_boiler` model uses 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]

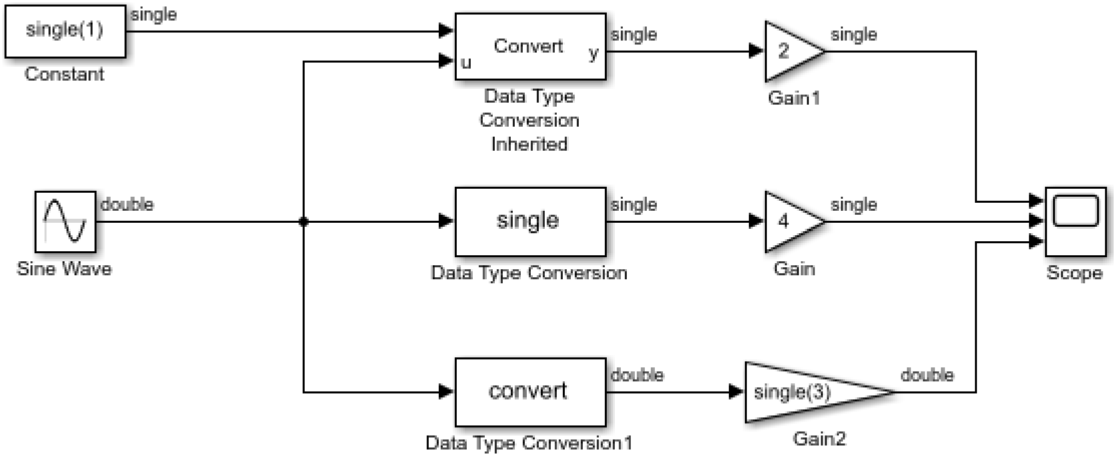


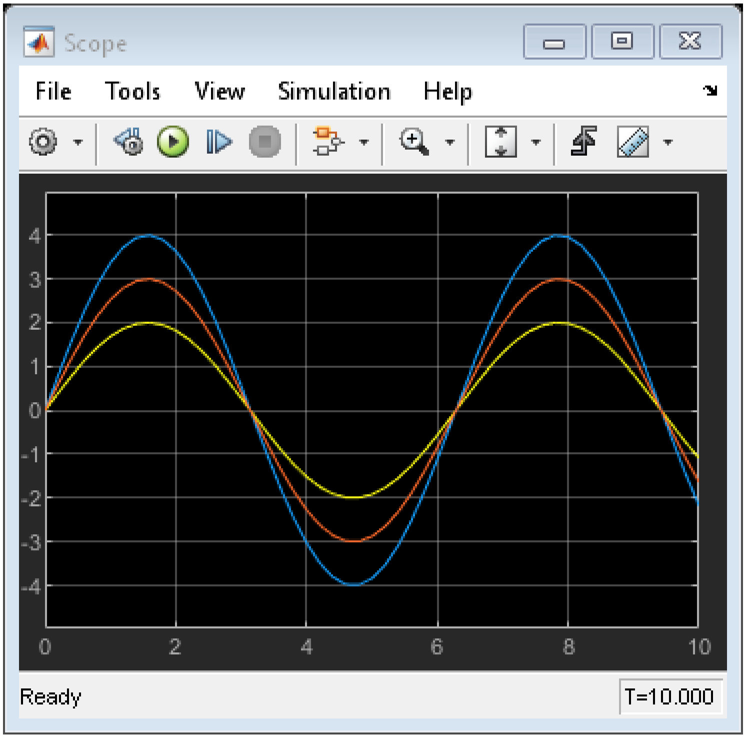
This subsystem models a digital thermometer composed of a simple temperature sensor and an ADC. The transfer function of the sensor is:
V = .05 * T + 0.75
for T in degrees C.

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.

Convert Data Types in Simulink Models

This example shows three different methods of converting data types in your model using the Data Type Conversion and Data Type Conversion Inherited blocks. In this model, a Sine Wave block generates the input signal. The Sine Wave block only outputs double-precision data types, so to generate a sine wave with a data type of single, you must perform a data type conversion.





In the first row, the Data Type Conversion Inherited block uses the data type coming from the Constant block (single) as the reference data type, and converts the sine wave to single.

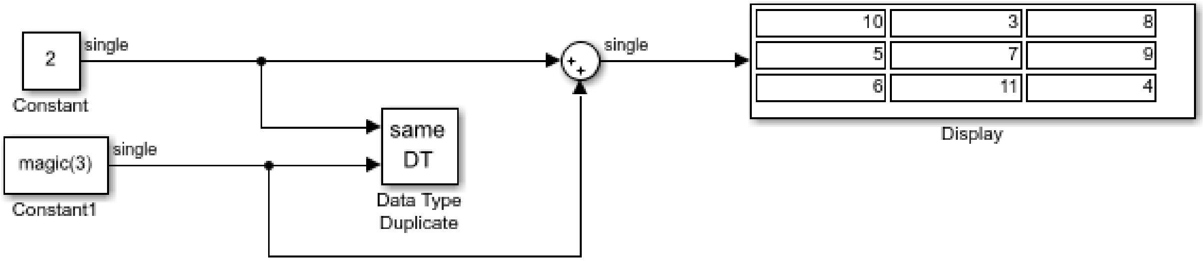
In the second row, the Data Type Conversion block has the **Output data type** set to `single`, and the sine wave is converted accordingly.

In the third row, the Data Type Conversion1 block has the **Output data type** set to `Inherit: Inherit via back propagation`. Because the downstream Gain2 block has a data type of single, the Data Type Conversion1 block converts the sine wave to a data type of single.

Control Data Types with the Data Type Duplicate Block

This example shows how to control data types in your model using the Data Type Duplicate block. In this model, the data type of the Constant block (currently `single`) drives the data types throughout the model.

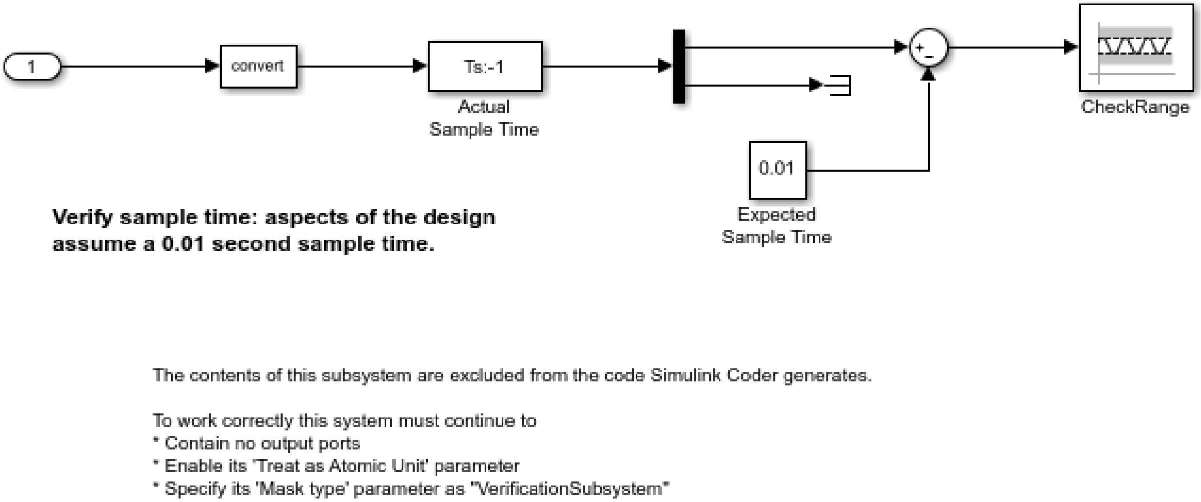
The Constant1 block has its **Output data type** parameter set to `Inherit: Inherit via back propagation`. Because the Constant1 and Constant blocks are both connected to the Data Type Duplicate block, the Constant1 block can inherit its data type from the Constant block. The Sum block has its **Output data type** set to `Inherit: Same as first input`, so it is also able to inherit its data type from the Constant block.



If you change the data type of the Constant block from `single` to `int32`, the `int32` data type propagates throughout the model.

Probe Sample Time of a Signal

The sldemo_fuelsys model shows how to check the sample time of a signal using the Probe block. This enables you to verify that the sample time matches the assumed value of the design.



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

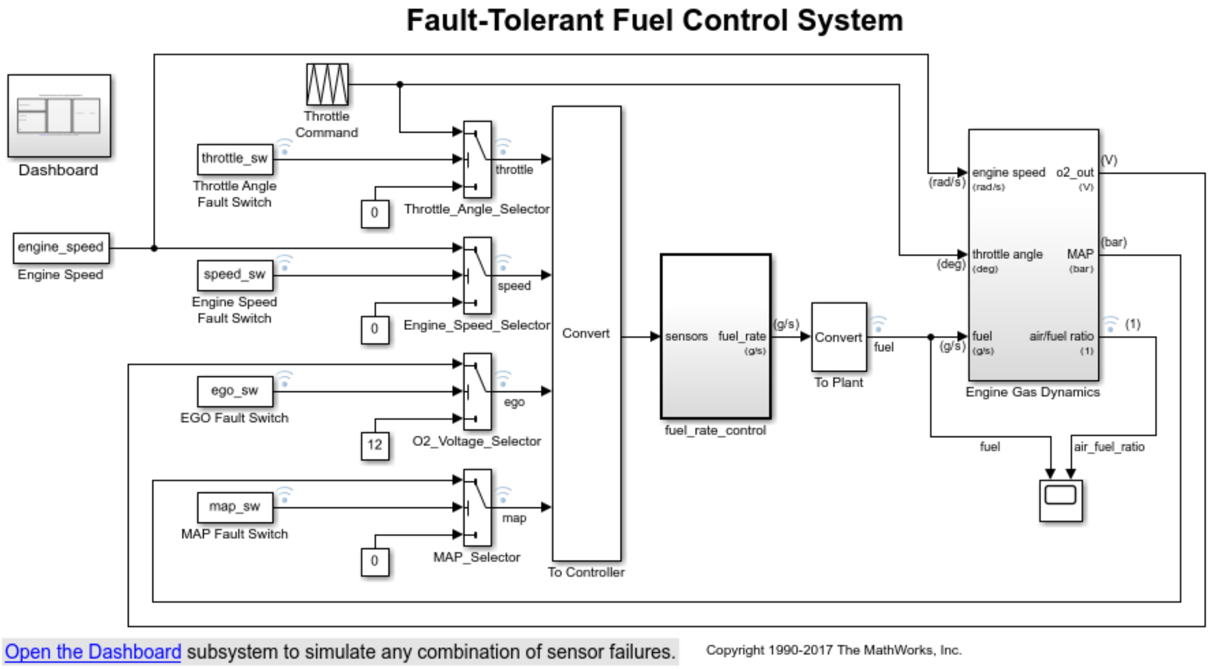
The contents of this subsystem are excluded from the code Simulink Coder generates.

To work correctly this system must continue to
* Contain no output ports
* Enable its 'Treat as Atomic Unit' parameter
* Specify its 'Mask type' parameter as "VerificationSubsystem"

For more information, see the model description.

Convert Signals Between Continuous Time and Discrete Time

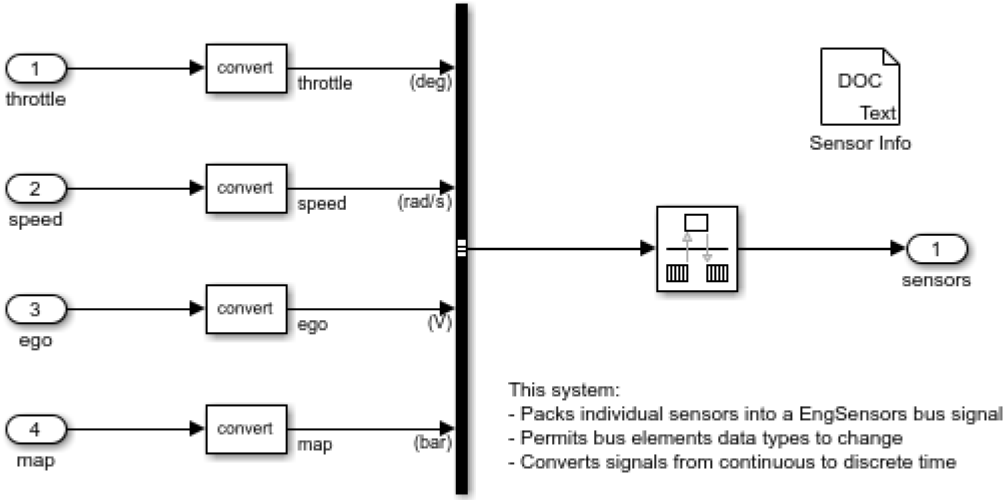
The sldemo_fuelsys model shows how to use the Rate Transition block to convert signals between continuous time and discrete time.



Open the Dashboard subsystem to simulate any combination of sensor failures.

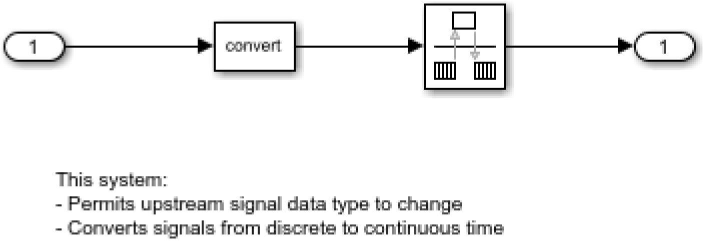
Copyright 1990-2017 The MathWorks, Inc.

In the `To Controller` subsystem, the Rate Transition block converts the signal from continuous time to discrete time. This discrete-time signal can then be processed by the `fuel_rate_control` subsystem.



This system:
- Packs individual sensors into a EngSensors bus signal
- Permits bus elements data types to change
- Converts signals from continuous to discrete time

The `To Plant` subsystem uses the Rate Transition block to convert the discrete-time output of the `fuel_rate_control` subsystem back to continuous time.

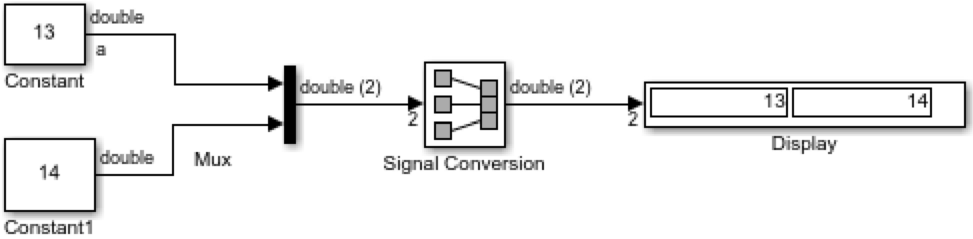


This system:
- Permits upstream signal data type to change
- Converts signals from discrete to continuous time

For more information, see the model description.

Convert Muxed Signal to a Vector

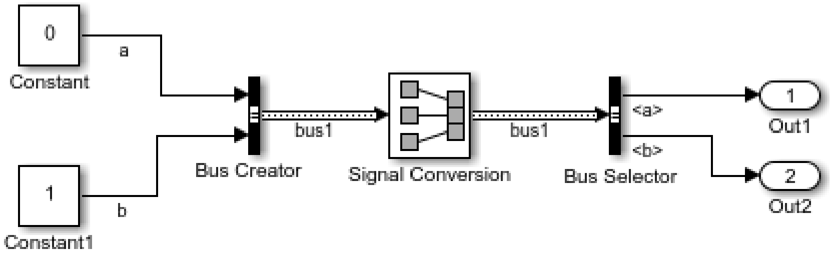
This example shows how to use the Signal Conversion block to convert a muxed input signal to a vector.



To convert the muxed signal to a vector, the **Output** parameter of the Signal Conversion block is set to `Signal copy`.

Create Contiguous Copy of a Bus Signal

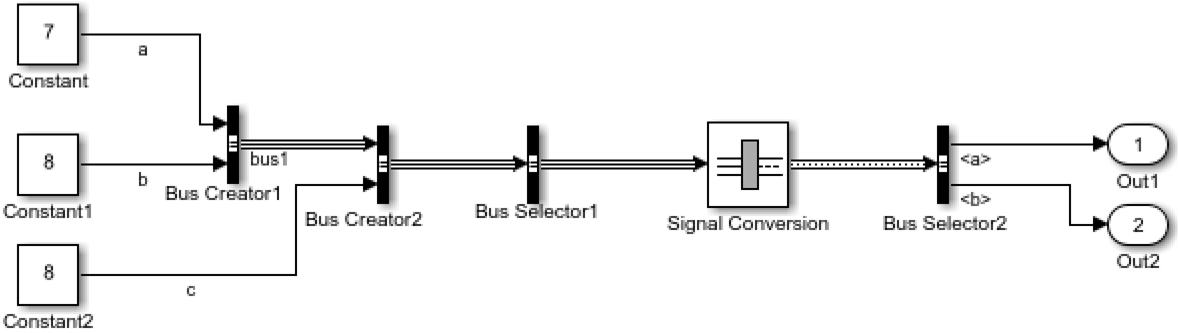
This example shows how to use the Signal Conversion block to create a contiguous copy of a bus signal.



The Bus Creator block creates a nonvirtual bus signal that is input to the Signal Conversion block. With the **Output** parameter set to `Signal copy`, the Signal Conversion block creates another contiguous copy of that input bus signal.

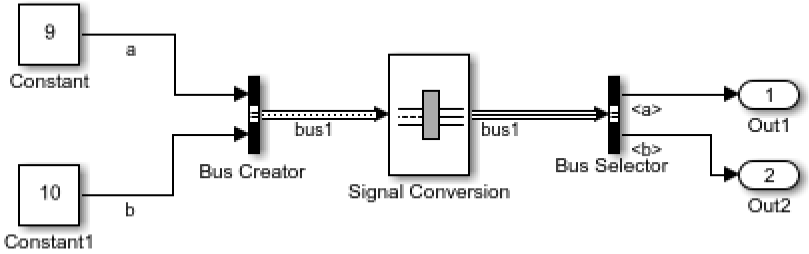
Convert Virtual Bus to a Nonvirtual Bus

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.



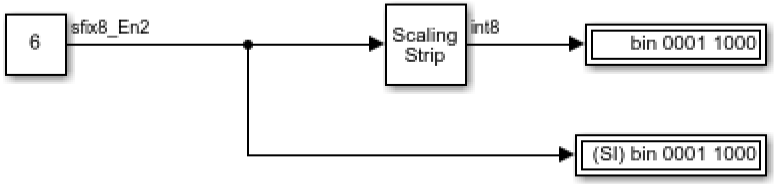
Convert Nonvirtual Bus to Virtual Bus

In the following example, 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 Signal Conversion block has its **Output** set to `Virtual bus`.



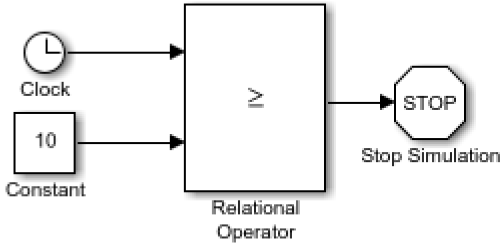
Remove Scaling from a Fixed-Point Signal

This example shows how to strip the scaling from a fixed-point input signal. To do so, the Data Type Scaling Strip block maps the input 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.



Stop Simulation Block with Relational Operator Block

This example shows how to control when a simulation stops by using a Stop Simulation block with a Relational Operator block. When you simulate the model, the model stops simulation when the simulation time reaches 10.

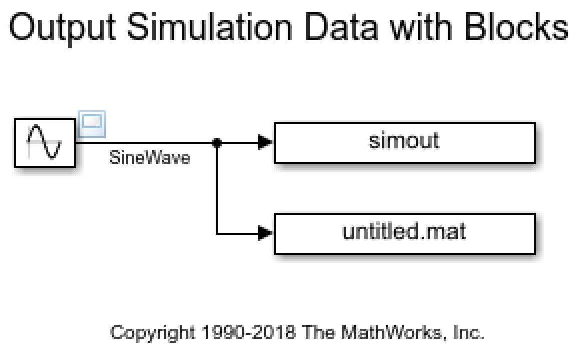


Output Simulation Data with Blocks

This example shows how To Workspace and To File blocks write data to the workspace and to a file respectively.

Open Example Model

```
open_system('ex_ToWorkspace_ToFile');
```


Output Simulation Data with Blocks
SineWave
simout
untitled.mat
Copyright 1990-2018 The MathWorks, Inc.

Simulate with Default Parameter Values

1. To name the output variables and file, modify the **Variable name** and **File name** block parameter values by using the Block Parameters dialog boxes or the command line.

```
set_param('ex_ToWorkspace_ToFile/To Workspace',...
    'VariableName','simoutToWorkspace')

set_param('ex_ToWorkspace_ToFile/To File',...
    'FileName','simoutToFile.mat',...
    'MatrixName','simoutToFileVariable')
```

2. Simulate the model.

```
sim('ex_ToWorkspace_ToFile');
```

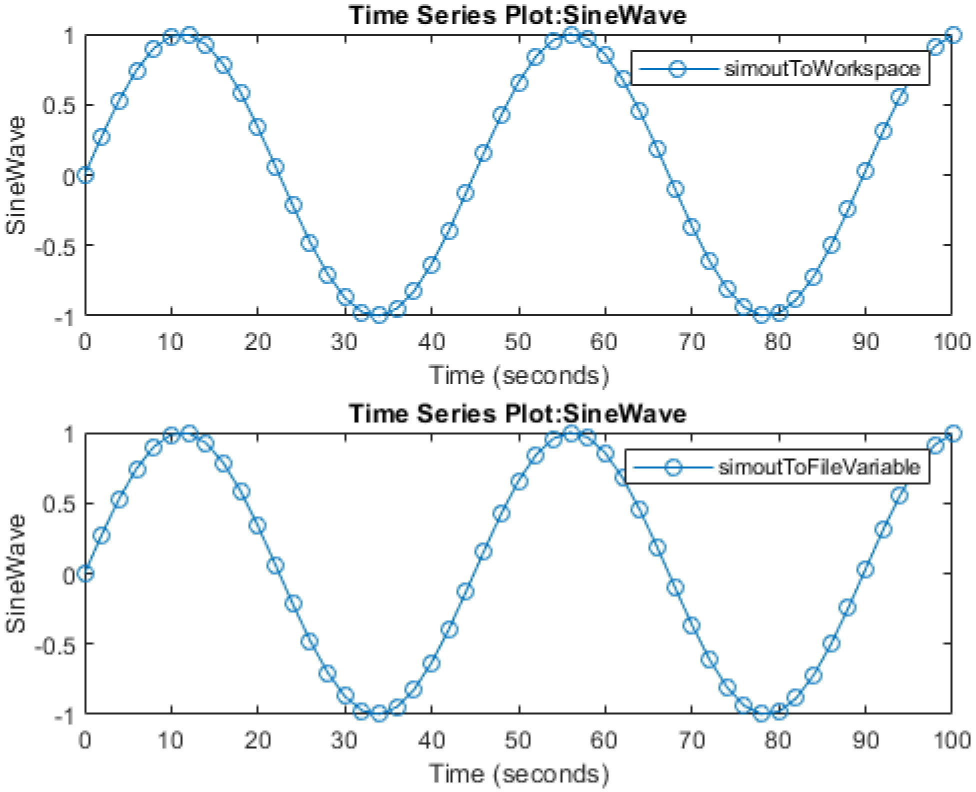
3. To view the input signal for the To Workspace and To File blocks, open the scope viewer.

4. To access the data stored by the To File block, load the output file.

```
load('simoutToFile.mat')
```

5. Plot the data stored by the To Workspace and To File blocks.

```
subplot(2,1,1)
plot(simoutToWorkspace,'-o')
legend('simoutToWorkspace')

subplot(2,1,2)
plot(simoutToFileVariable,'-o')
legend('simoutToFileVariable')
```



As shown by the plots, the data stored by each block is the same given the default block parameter values.

Simulate with Custom Parameter Values

1. To keep the data from the previous simulation, specify new names for the output variables and file.

```
set_param('ex_ToWorkspace_ToFile/To Workspace',...
    'VariableName','simoutToWorkspace2')

set_param('ex_ToWorkspace_ToFile/To File',...
    'FileName','simoutToFile2.mat',...
    'MatrixName','simoutToFileVariable2')
```

2. To change the amount of data collected, modify the **Limit data points to last**, **Decimation**, and **Sample time** block parameter values.

```
set_param('ex_ToWorkspace_ToFile/To Workspace',...
    'MaxDataPoints','3',...
    'Decimation','20',...
    'SampleTime','0.5')

set_param('ex_ToWorkspace_ToFile/To File',...
    'Decimation','20',...
    'SampleTime','1')
```

The To File block does not provide the option to limit data points to the last data points collected.

3. Select **Single simulation output**, then modify the **Logging intervals** configuration parameter value.

```
set_param('ex_ToWorkspace_ToFile',...
    'ReturnWorkspaceOutputs','on',...
    'LoggingIntervals','[20,90]')
```

4. Simulate the model.

```
out = sim('ex_ToWorkspace_ToFile');
```

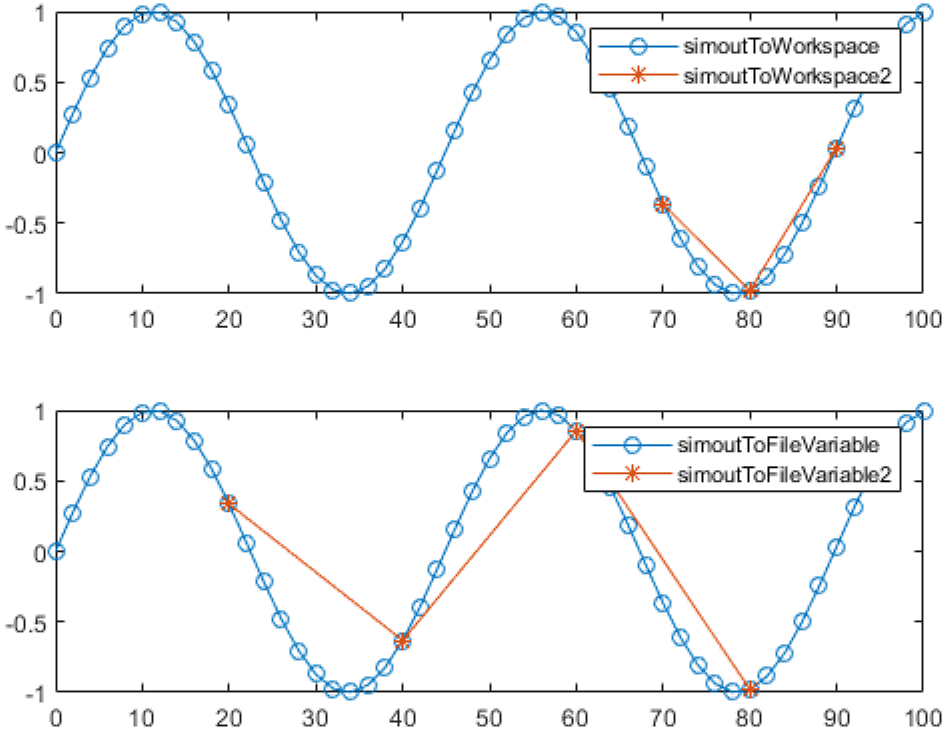
5. To access the data stored by the To File block, load the output file.

```
load('simoutToFile2.mat')
```

6. Plot the data stored by the To Workspace and To File blocks.

```
subplot(2,1,1)
hold on
```

```
plot(out.simoutToWorkspace2,'-*','DisplayName','simoutToWorkspace2')
hold off

subplot(2,1,2)
hold on
plot(simoutToFileVariable2,'-*','DisplayName','simoutToFileVariable2')
hold off
```



In this example, the To Workspace block collects data at 20, 30, 40, ..., 90 seconds. The data represents every 20th sample time within the logging intervals. When the simulation is completed or paused, the To Workspace block writes only the last three collected sample points to the workspace: 70, 80, and 90 seconds.

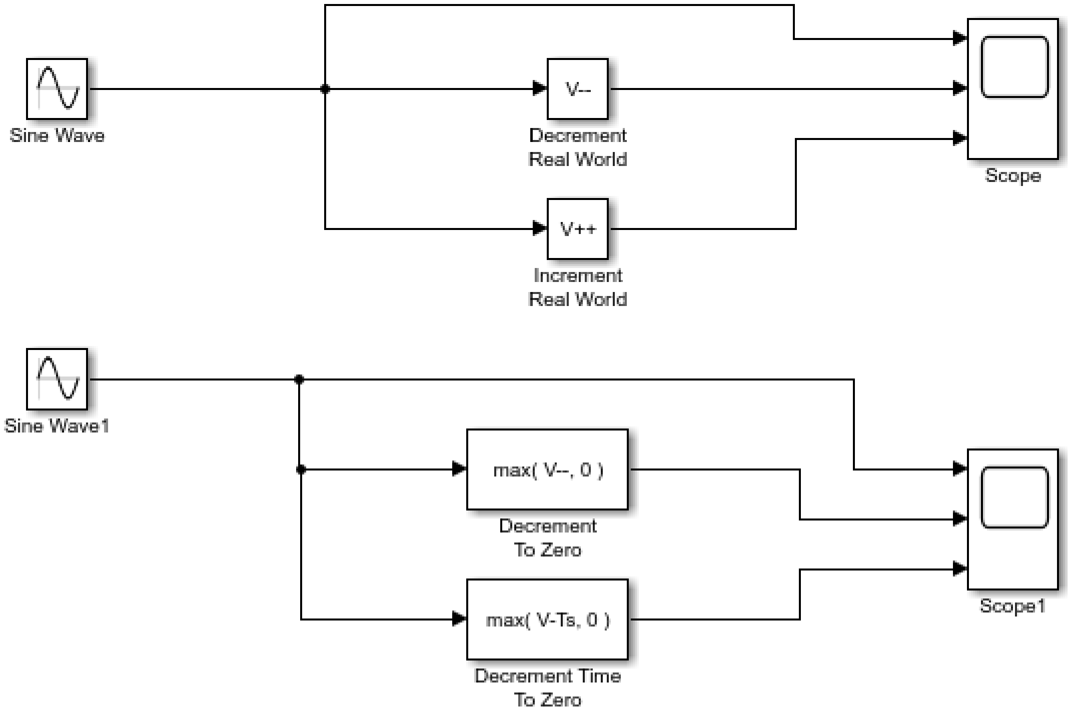
The To File block collects data at 20, 40, 60, and 80 seconds. The data similarly represents every 20th sample time within the logging intervals. However, the sample time for the To File block is double the sample time for the To Workspace block.

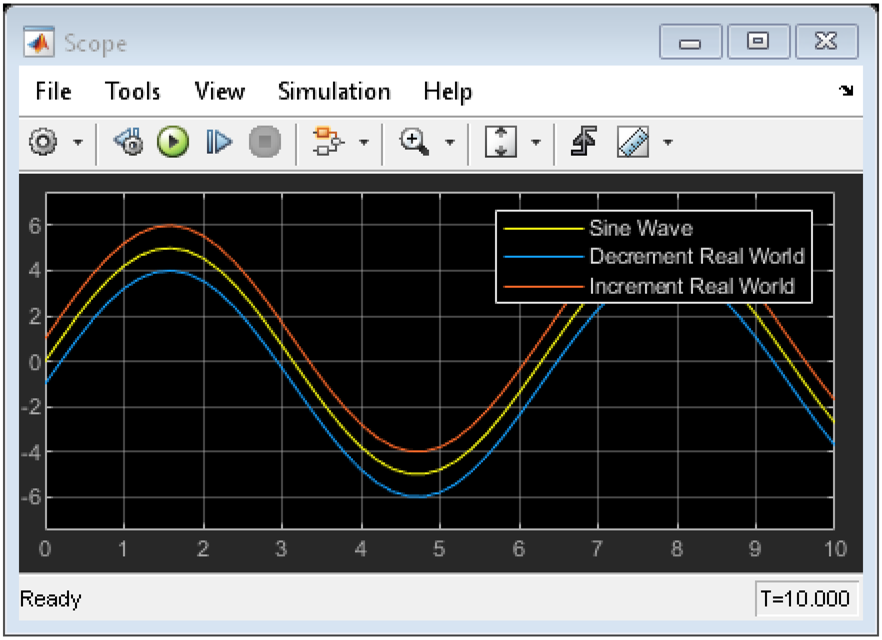
Increment and Decrement Real-World Values

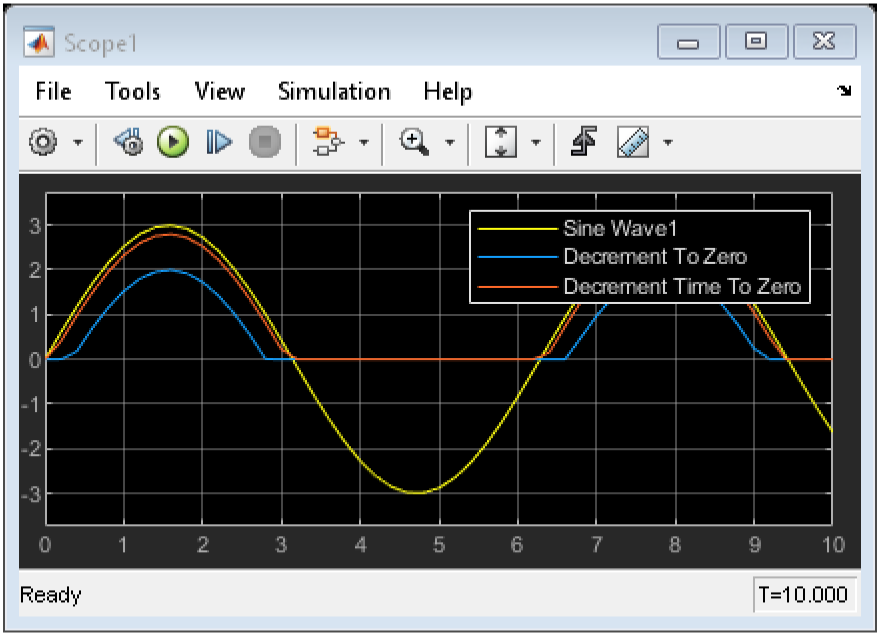
This example shows how to increase and decrease the real-world value of a signal using the following blocks:

- Increment Real World
- Decrement Real World
- Decrement Time To Zero
- Decrement To Zero

The Scope block shows the output of a Sine Wave block with amplitude 5, as well as the real-world value of that signal incremented and decremented by one.




Scope
File
Tools
View
Simulation
Help
6
4
2
0
-2
-4
-6
0
1
2
3
4
5
6
7
8
9
10
Sine Wave
Decrement Real World
Increment Real World
Ready
T=10.000



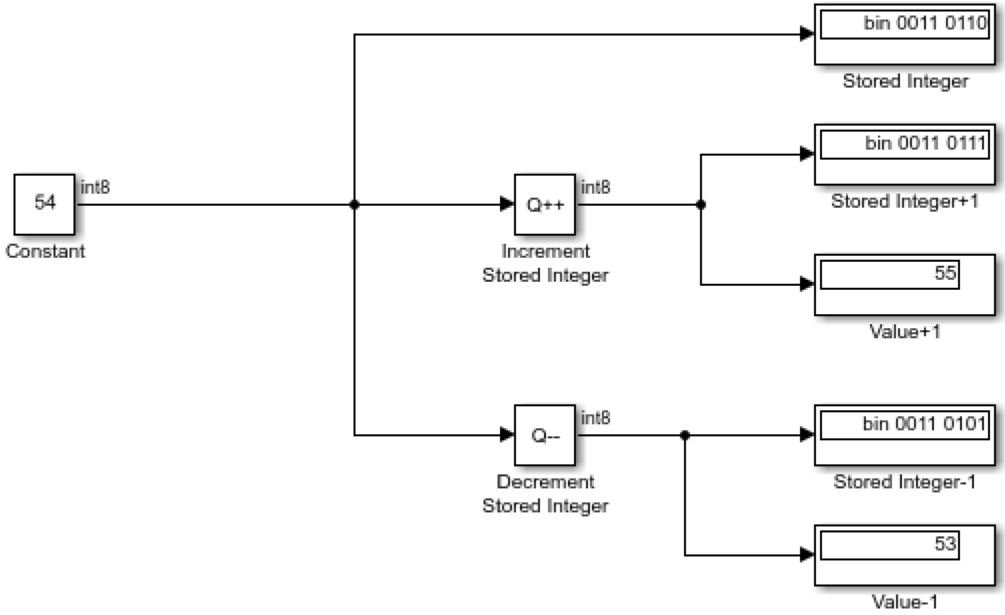
The Scope1 block shows the output of a Sine Wave block with amplitude 3, as well as the output of the Decrement To Zero and Decrement Time To Zero blocks:

- The Decrement To Zero block decreases the input sine wave signal by one, and ensures the value never goes below zero.
- The Decrement Time To Zero block decreases the input sine wave signal by the sample time, `Ts`, and ensures that the value never goes below zero.

Increment and Decrement Stored Integer Values

This example shows how to increase and decrease the stored integer value of a signal by one.

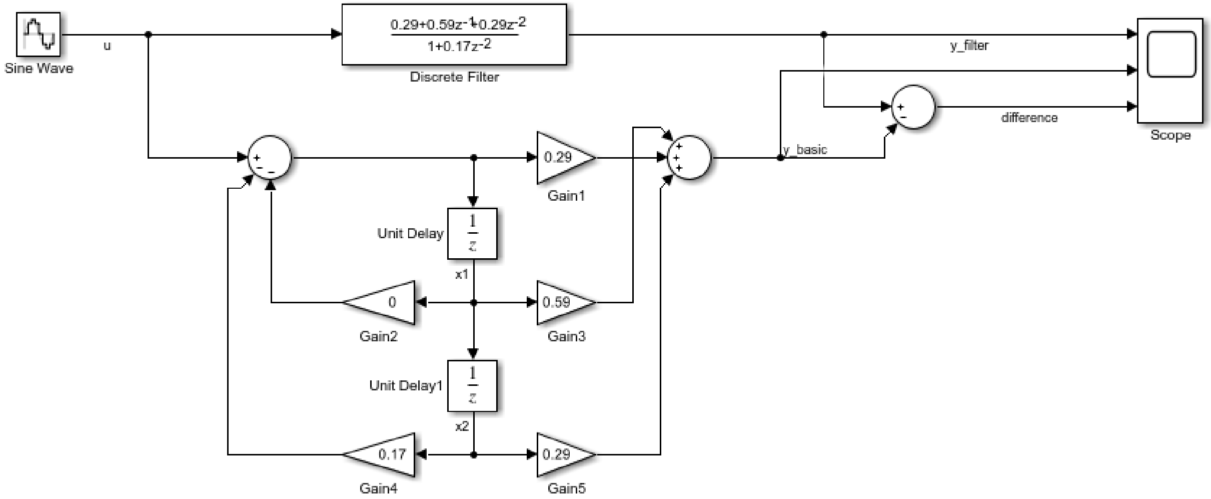
- The Increment Stored Integer block increases the stored integer value of the input signal by one.
- The Decrement Stored Integer block decreases the stored integer value of the input signal by one.

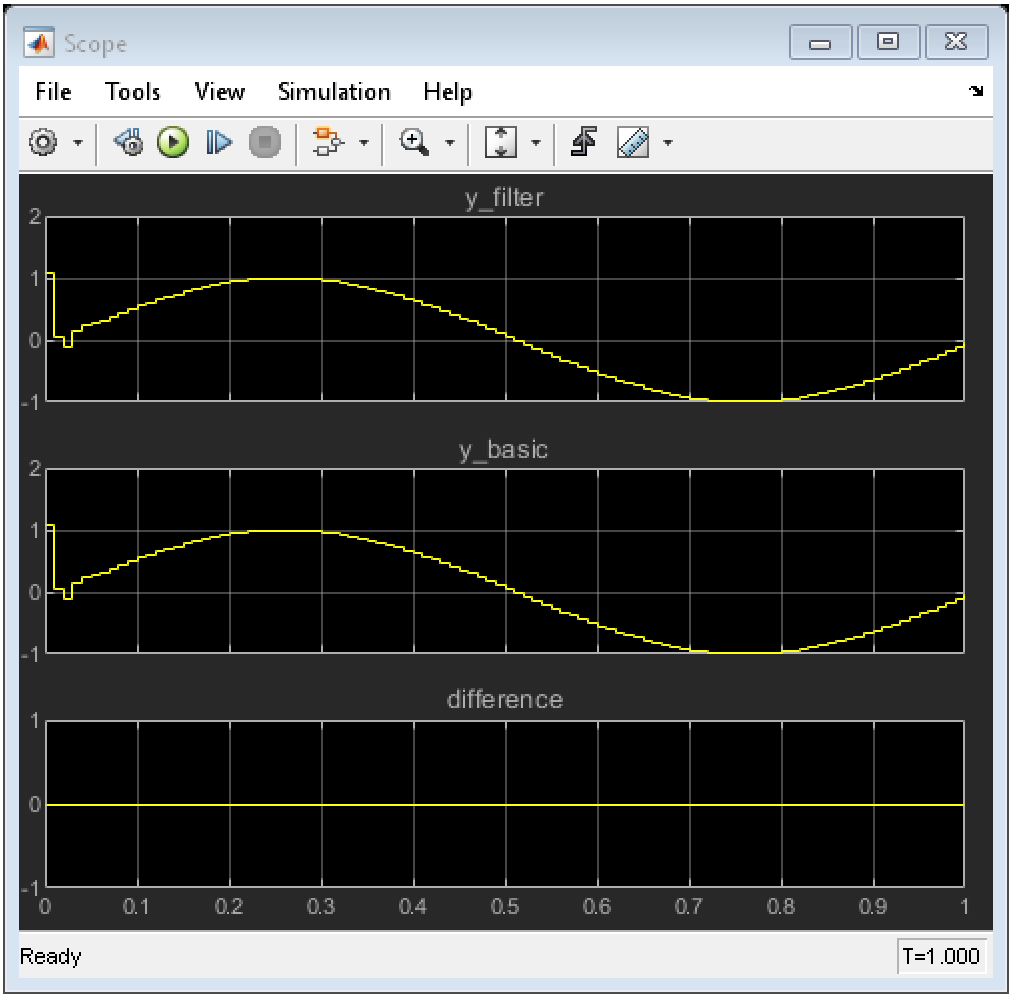


If you change the value of the input signal to 127 (the maximum value representable by an `int8` data type), incrementing the stored integer value by one causes an overflow. Because overflows in the Increment and Decrement Stored Integer blocks always wrap, the Increment Stored Integer block will output a value of -128.

Specify a Vector of Initial Conditions for a Discrete Filter Block

This example shows how to specify a vector of non-zero initial conditions for the Discrete Filter block.



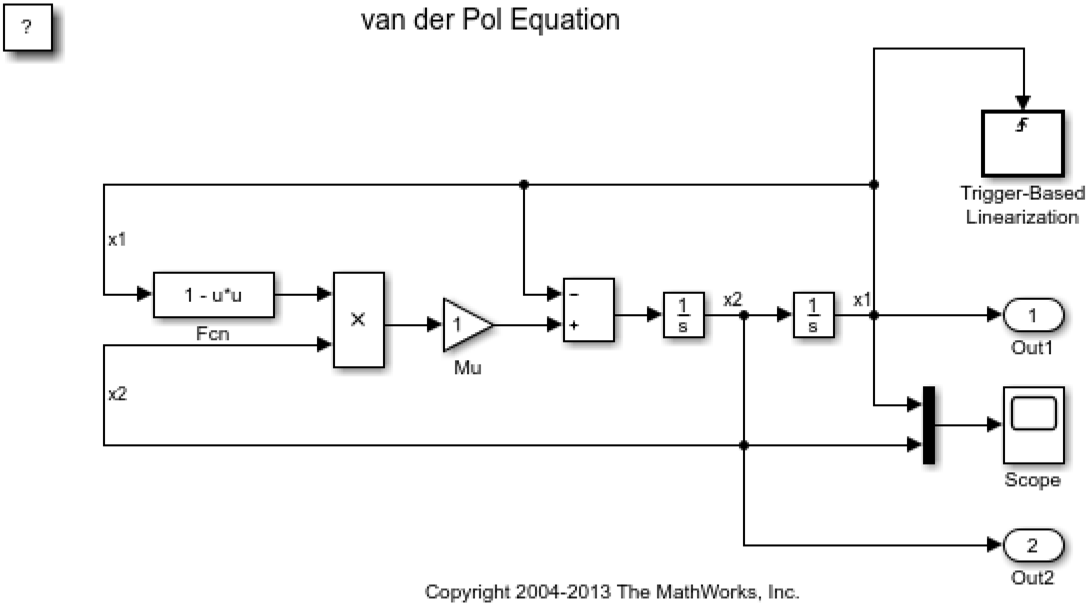


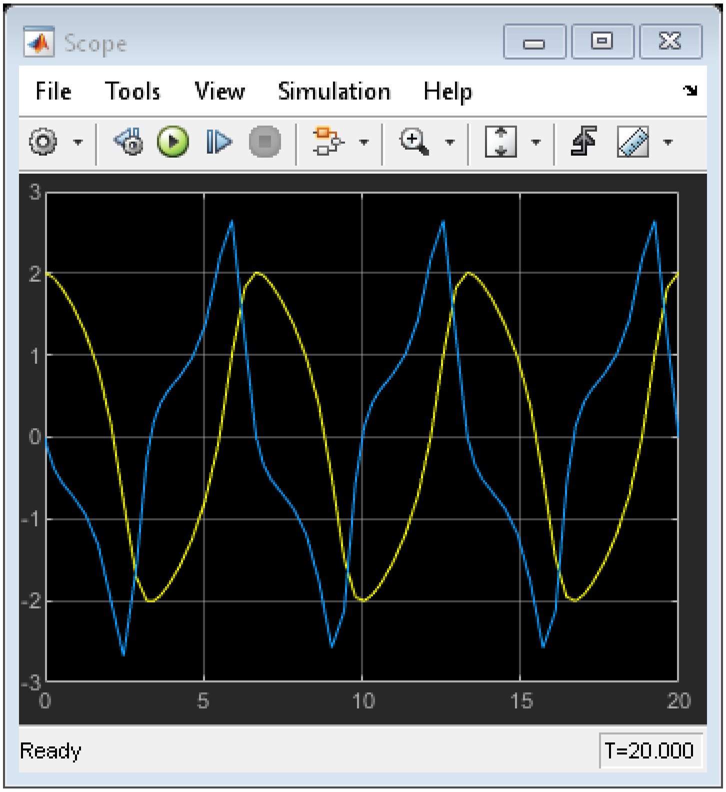
The Scope shows that with the **Initial states** of the Discrete Filter block set to `[1 2]`, the difference between the signal filtered by the Discrete Filter block and the signal from the filter's building blocks is zero. This demonstrates that you can enter the initial conditions of the Discrete Filter block as a vector of `[1 2]`. As an alternative, you can set the initial condition of the Unit Delay Block to 2. The resulting output is the same.

Generate Linear Models for a Rising Edge Trigger Signal

You can use state and simulation time logging to extract the model states at operating points. In this example, the model is configured to get the states when the `x1` signal triggers the Trigger-Based Linearization block on a rising edge.

In this model, the **Trigger type** of the Trigger-Based Linearization block is set to `rising`. On the Data Import/Export pane of the Model Configuration Parameters dialog box, the `States` and `Time` check boxes are selected.


?
van der Pol Equation
Trigger-Based Linearization
x1
1 - u*u
Fcn
×
1
Mu
-
+
1/s
x2
1/s
x1
1
Out1
x2
Scope
2
Out2
Copyright 2004-2013 The MathWorks, Inc.



After simulating the model, the following variables appear in the MATLAB workspace:

- ex_vdp_triggered_linearization_Trigger_Based_Linearization
- tout
- xout

To get the index to the first operating point time, execute the following command:

```
ind1 = find(ex_vdp_triggered_linearization_Trigger_Based_Linearization(1).OperPoint.t==
```

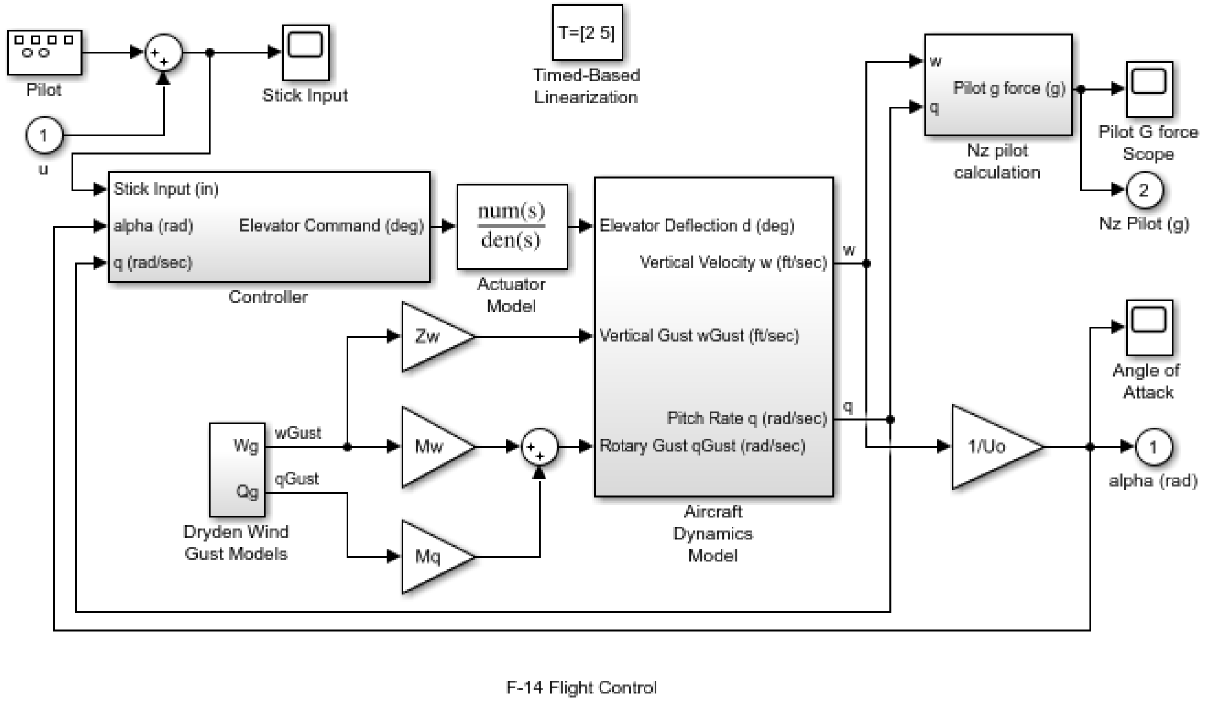
To get the state vector at this operating point, execute the following command:

```
x1 = xout(ind1,:);
```

Generate Linear Models at Predetermined Times

This example shows how to use the Timed-Based Linearization block to generate linear models at predetermined times.

In this model, the **Linearization time** of the Timed-Based Linearization block is set to [2 5]. On the Data Import/Export pane of the Model Configuration Parameters dialog box, the `States` and `Time` check boxes are selected. These settings enable you to get the states of the model at the simulation times of 2 and 5 seconds.



After simulating the model, the following variables appear in the MATLAB workspace:

- ex_f14_linearization_Timed_Based_Linearization

- tout
- xout

To get the indices to the operating point times, execute the following command:

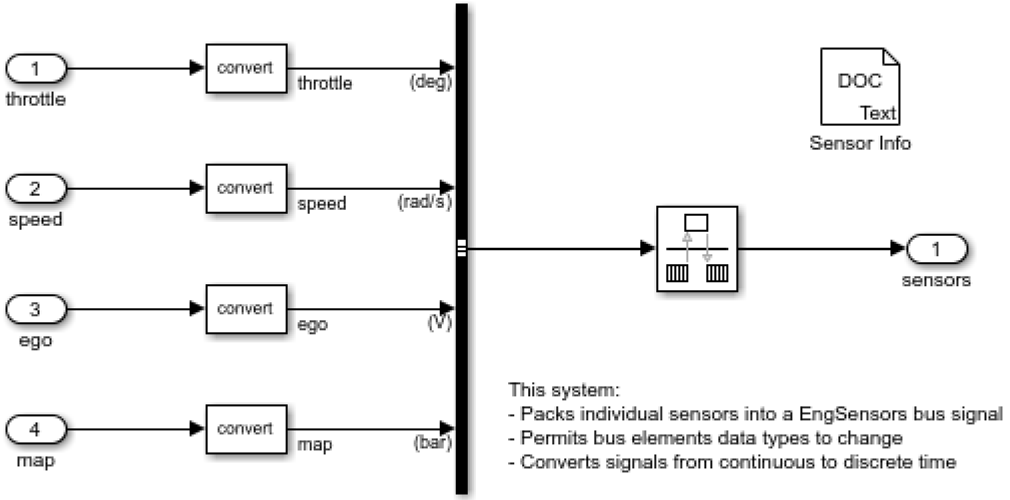
```
ind1 = find(ex_f14_linearization_Timed_Based_Linearization(1).OperPoint.t==tout);
```

To get the state vectors at the operating points, execute the following command:

```
ind2 = find(ex_f14_linearization_Timed_Based_Linearization(1).OperPoint.t==tout);
```

Capture Measurement Descriptions in a DocBlock

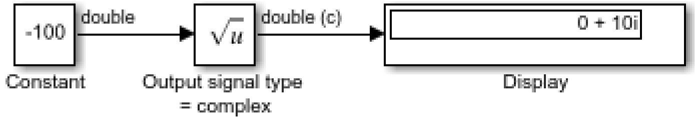
This example shows how to use a DocBlock to capture information about a model. In the sldemo_fuelsys model, a DocBlock labeled `Sensor Info` is used to document measurement descriptions inside of the `To Controller` subsystem.


1
throttle
convert
throttle
(deg)
2
speed
convert
speed
(rad/s)
3
ego
convert
ego
(V)
4
map
convert
map
(bar)
DOC
Text
Sensor Info
1
sensors
This system:
- Packs individual sensors into a EngSensors bus signal
- Permits bus elements data types to change
- Converts signals from continuous to discrete time

For more information, see the model description.

Square Root of Negative Values

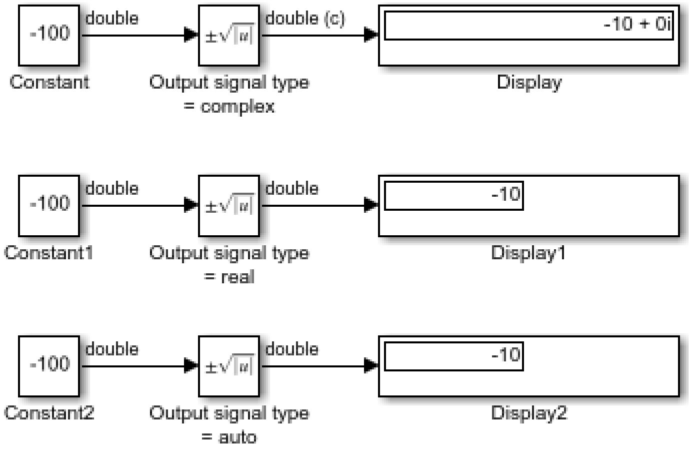
This example shows how to compute the square root of a negative-valued input signal as complex-valued output.



By setting the **Function** to `sqrt` and **Output signal type** to `complex`, the block produces the correct result of `0 + 10i` for an input of `-100`. If you change the **Output signal type** to `auto` or `real`, the block outputs `NaN`.

Signed Square Root of Negative Values

This example shows how to compute the signed square root of a negative-valued input signal.



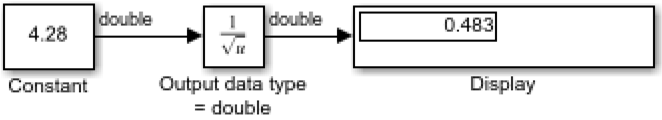
When the block input is negative and you set the **Function** to `signedSqrt`, the Sqrt block output is the same for any setting of the **Output signal type** parameter. By setting the **Format** of the first Display block to `decimal (Stored Integer)`, you can see the value of the imaginary part for the complex output.

rSqrt of Floating-Point Inputs

This example shows how to compute the rSqrt of a floating-point input signal. The Sqrt block has the following settings:

- **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).

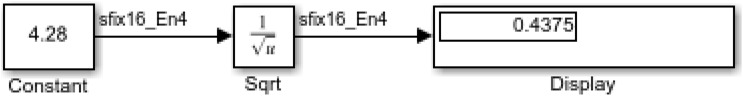


rSqrt of Fixed-Point Inputs

This example shows how to compute the rSqrt of a fixed-point input signal. The Sqrt block has the following settings:

- **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).



Model a Series RLC Circuit

Physical systems can be described as a series of differential equations in an implicit form, $F\left(t, x, \{\dot{x}\}\right) = 0$, or in the implicit state-space form $E\dot{x} = A\,x + B\,u$

If E is nonsingular, then the system can be easily converted to a system of ordinary differential equations (ODEs) and solved as such:

$$\dot{x} = \left(E^{-1}A\right)x + \left(E^{-1}B\right)u$$

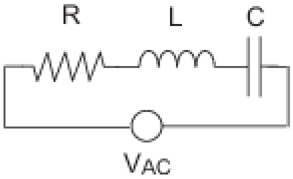
Many times, states of a system appear without a direct relation to their derivatives, usually representing physical conservation laws. For example:

$$\begin{aligned}\dot{x_1} &= x_2\\ 0 &= x_1 + x_2\end{aligned}$$

In this case, E is singular and cannot be inverted. This class of systems are commonly called *descriptor* systems and the equations are called differential-algebraic equations (DAEs).

Series RLC Circuit

Consider the simple series RLC circuit.



From Kirchoff's Voltage Law, the voltage drop across the circuit is equalt to the sum of the voltage drop across each of its elements:

$$V_{AC} = V_R + V_L + V_C$$

From Kirchoff's Current Law:

$$I_{AC} = I_R = I_L = I_C$$

where the subscripts R, L, and C denote the resistance, inductance, and capacitance respectively.

$$V_R = I\left(t\right)R$$

$V_L = L\dot{I}_L$ or $\dot{I}_L = \frac{1}{L}V_L$

$V_C = V_{AC}\left(0\right) + \int_0^t I_C\left(\tau\right)d\tau$ or $\dot{V}_c = \frac{1}{C}I_c$

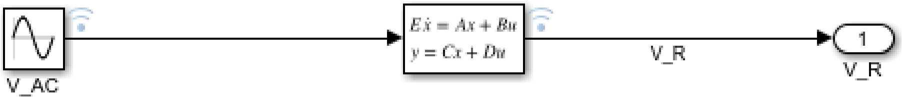
In Implicit State-Space Form

Model the system in Simulink with $R = 10\ \Omega$, $L = 1 \times 10^{-6}\ H$, $C = 1 \times 10^{-4}F$ to find the voltage across the resistor V_R. To use the Descriptor State-Space block, the system can be written in the implicit, or *descriptor*, state-space form $E\dot{x} = Ax + Bu$ as shown below.

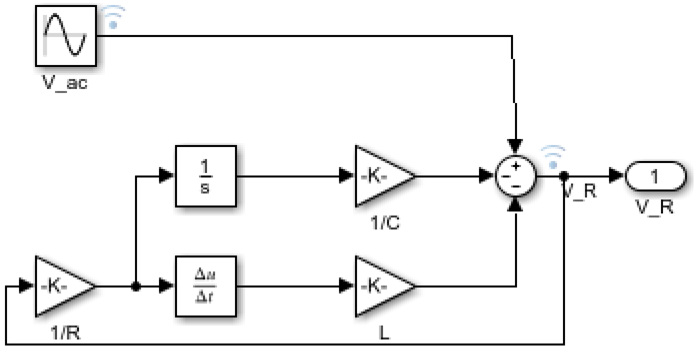
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{V}_C \\ \dot{V}_L \\ \dot{V}_R \\ \dot{I}_L \\ \dot{I}_{AC} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \frac{1}{C} & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & R & 0 \\ 0 & \frac{1}{L} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} V_C \\ V_L \\ V_R \\ I_L \\ I_{AC} \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} V_{AC}$$

where $x = \begin{bmatrix} V_C & V_L & V_R & I_L & I_{AC} \end{bmatrix}^T$ is the state vector.

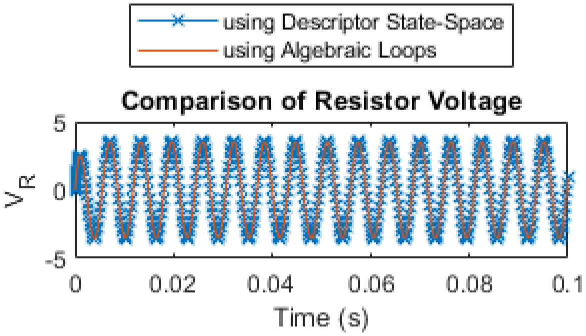
Set $C = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}$ since the voltage across the resistor is being measured.



Compare this to modeling the system with an algebraic loop in order to find V_R.



The simulation of both models produces identical results. However, the Descriptor State-Space block allows you to make a simpler block diagram and avoid algebraic loops.



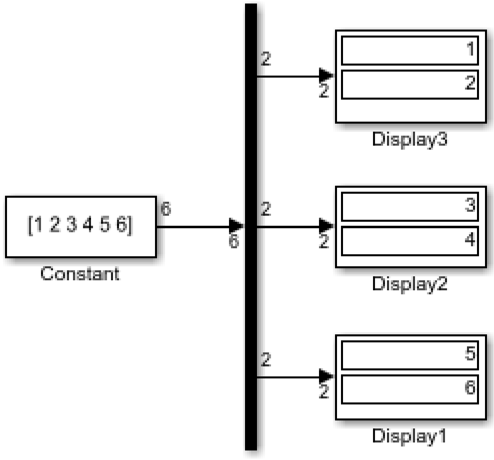
See Also

“Algebraic Loops”

“Model Differential Algebraic Equations”

Extract Vector Elements and Distribute Evenly Across Outputs

This example shows how you can use the Demux block to distribute an input signal evenly over the desired number of outputs. For an input vector of length 6, when you set the **Number of outputs** parameter to 3, the Demux block creates three output signals, each of size 2.



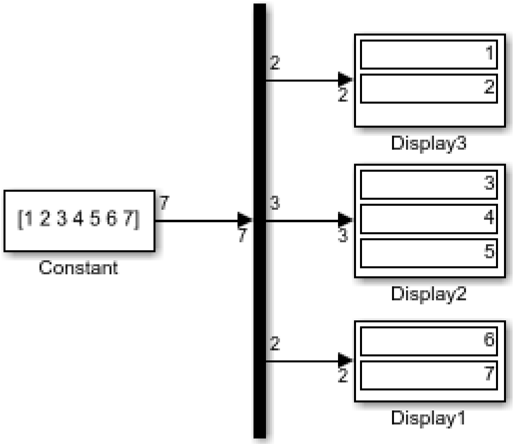
See Also

Demux

Extract Vector Elements Using the Demux Block

When using the Demux block to extract and output elements from a vector input, you can use -1 in a vector expression to indicate that the block dynamically sizes the corresponding port. When a vector expression comprises both 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.

In this example, the **Number of outputs** parameter of the Demux block is set to [-1, 3, -1]. Thus, the block outputs 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. For an input vector with seven elements, the Demux block outputs two elements on the first port, three elements on the second port, and two elements on the third port.

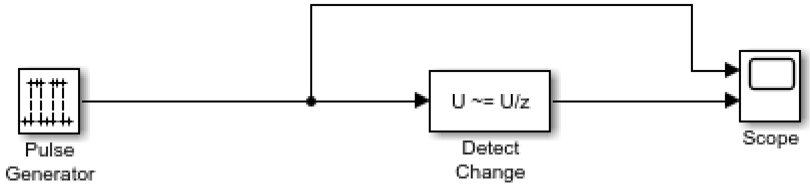


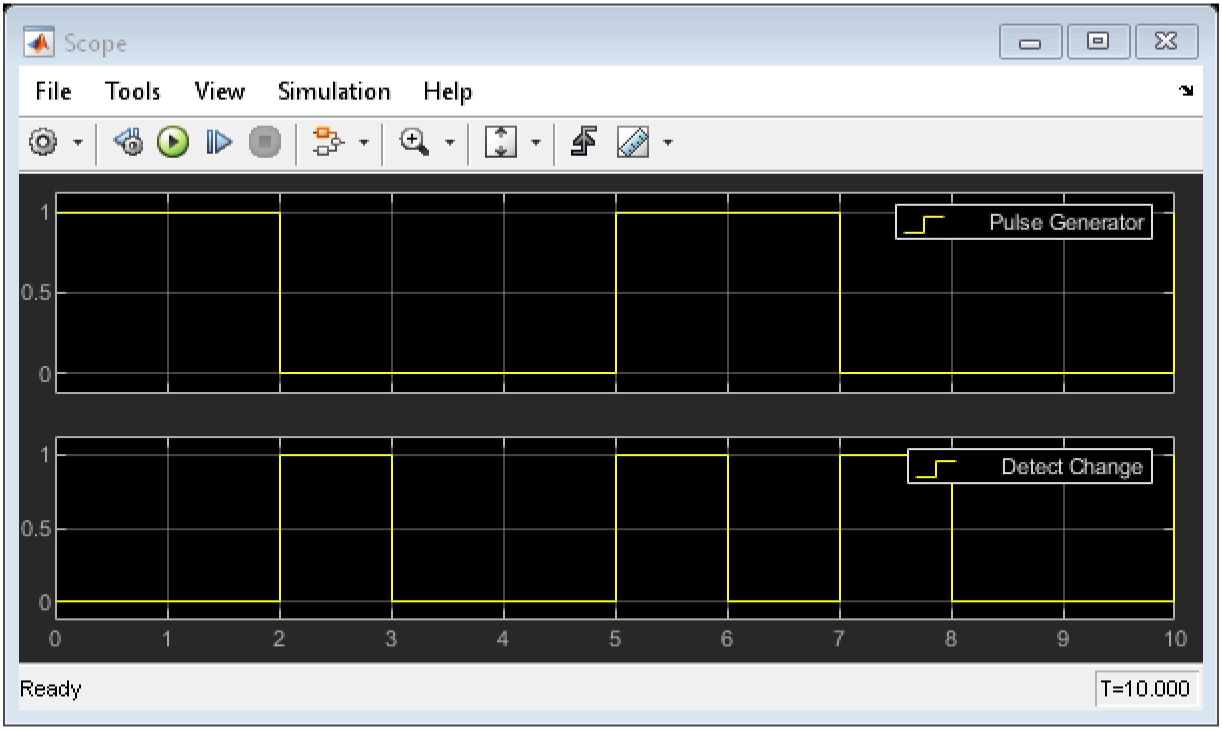
See Also

Demux

Detect Change in Signal Values

This example shows how to detect a change in signal values using the Detect Change block. When the input from the Pulse Generator block remains the same, the Detect Change block outputs zero (false), indicating that there was no change in signal values. When the value of the Pulse Generator block changes, the Detect Change block outputs one (true) indicating that the current signal value does not equal its previous value.



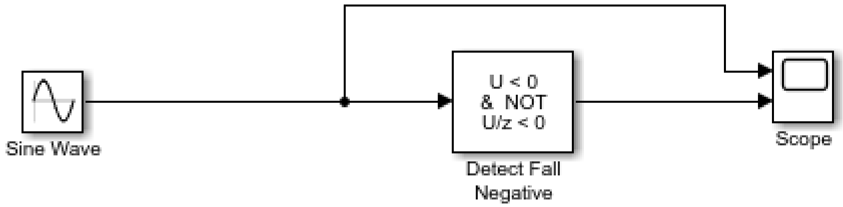


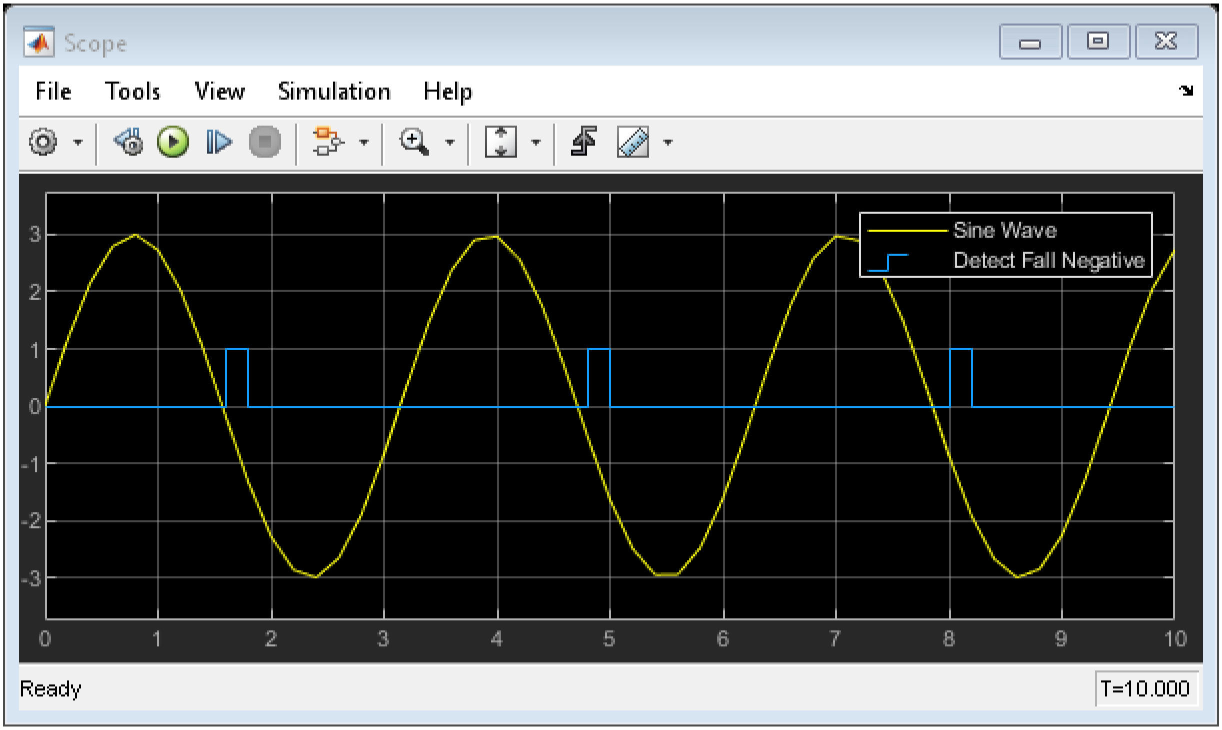
See Also

Detect Change | Pulse Generator | Scope

Detect Fall to Negative Signal Values

This example shows how to detect when a signal value decreases to a strictly negative value from a value that was greater than or equal to zero.



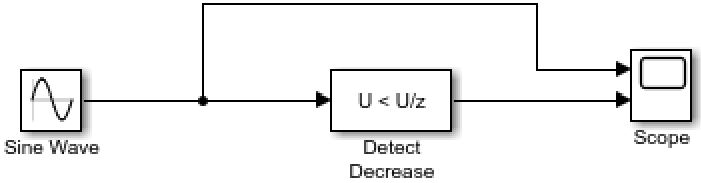


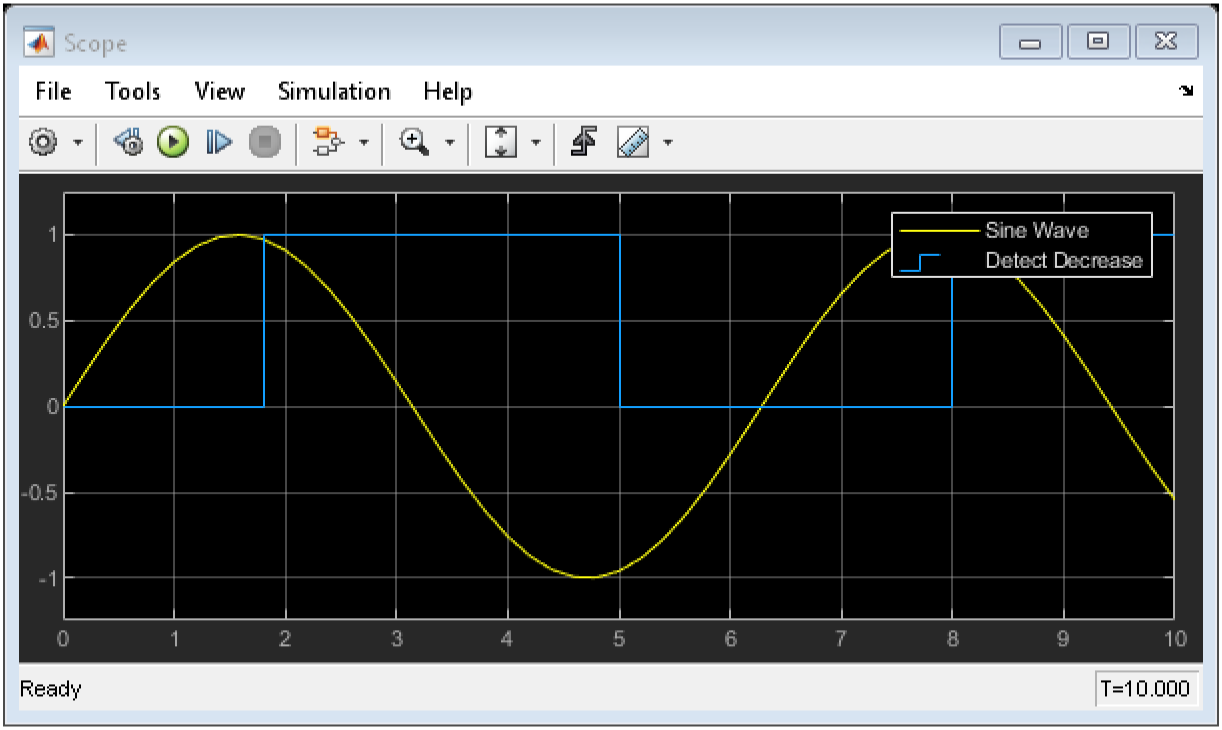
See Also

Detect Fall Negative | Scope | Sine Wave

Detect Decreasing Signal Values

This example shows how to determine if an input signal is strictly less than its previous value using the Detect Decrease block. When the current input to the Detect Decrease block is less than its previous value, the block outputs one (true). When the current input is greater than or equal to the previous signal value, the Detect Decrease block outputs zero (false).



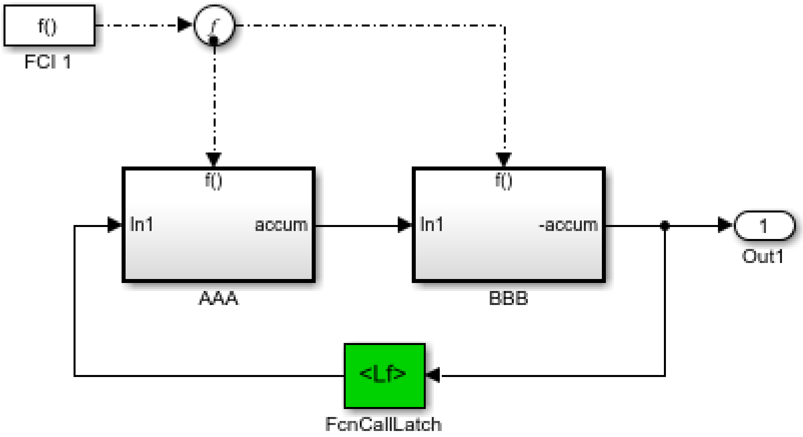


See Also

Detect Decrease | Scope | Sine Wave

Function-Call Blocks Connected to Branches of the Same Function-Call Signal

In this model, a Function-Call Feedback Latch block is 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.

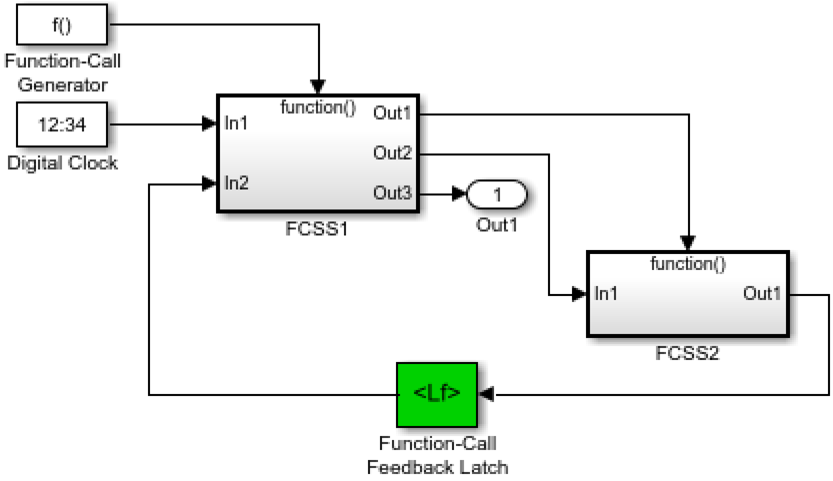


See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Function-Call Subsystem

Function-Call Feedback Latch on Feedback Signal Between Child and Parent

In this model, the Function-Call Feedback Latch block is 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).

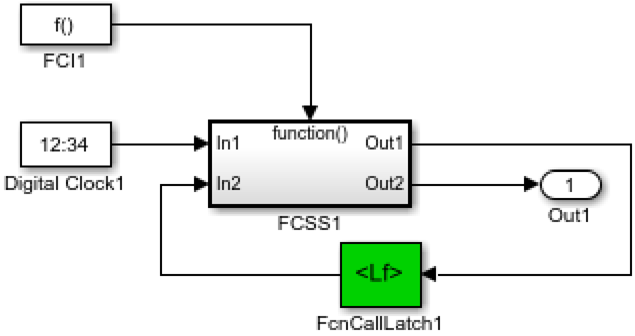


See Also

Digital Clock | Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem

Single Function-Call Subsystem

In this example, a single function-call subsystem output serves as its own input.

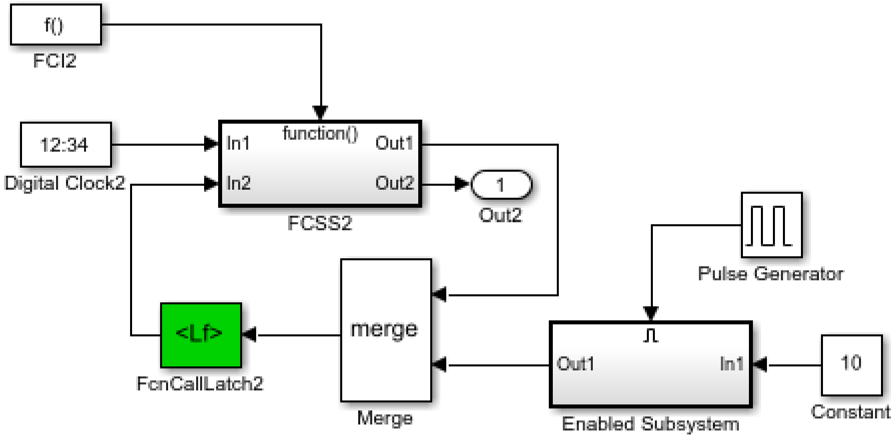


See Also

Digital Clock | Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem

Function-Call Subsystem with Merged Signal As Input

In this model 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 inport of FCSS2.

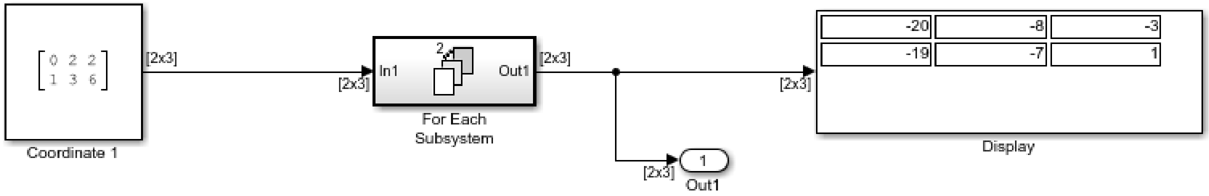


See Also

Digital Clock | Enabled Subsystem | Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem | Merge | Pulse Generator

Partitioning an Input Signal with the For Each Block

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.



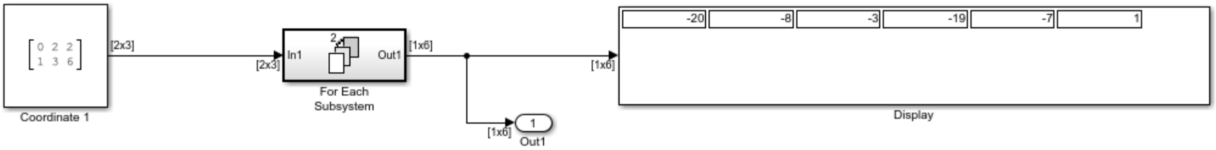
By placing the process of summing an input signal and the new origin inside of a For Each Subsystem block, you can operate on each set of coordinates by partitioning the input signal into two row vectors. To accomplish such partitioning, use the default settings of 1 for both the partition dimension and the partition width. If you also use the default concatenation dimension of 1, each new set of coordinates stacks in the d1 direction, making your display a 2-by-3 array.

See Also

For Each | For Each Subsystem

Specifying the Concatenation Dimension in the For Each Block

This example shows how to specify a concatenation dimension in the For Each block. When you specify a **Concatenation Dimension** of 2 on the **Output Concatenation** tab, each set of results stacks in the d2 direction, and the result is a single row vector.



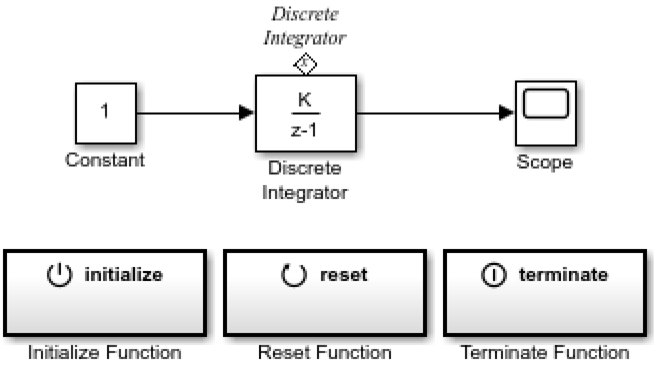
To learn how the For Each block and subsystem handle a model with states, see the For Each Subsystem block reference page.

See Also

For Each | For Each Subsystem

Working with the Initialize Function, Reset Function, and Terminate Function Blocks

This example shows how to use the Initialize Function, Reset Function, and Terminate Function blocks to read and write states in a Simulink model. For more information on configuring the block settings, see "Customize Initialize, Reset, and Terminate Functions".



See Also

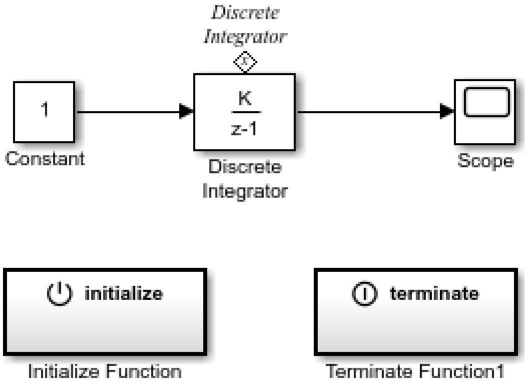
Discrete-Time Integrator | Event Listener | Initialize Function | Reset Function | State Reader | State Writer | Terminate Function

Related Examples

- "Customize Initialize, Reset, and Terminate Functions"

Reading and Writing States with the Initialize Function and Terminate Function Blocks

In this example, the Initialize Function block uses the State Writer block to set the initial condition of a Discrete Integrator block to 10. The Terminate Function block includes a State Reader block, which reads the state of the Discrete Integrator block. The **Event type** parameter of the Event Listener block for the initialize and terminate functions is set to `Initialize` and `Terminate`, respectively.



See Also

Discrete-Time Integrator | Event Listener | Initialize Function | State Reader | State Writer | Terminate Function

Related Examples

- “Customize Initialize, Reset, and Terminate Functions”

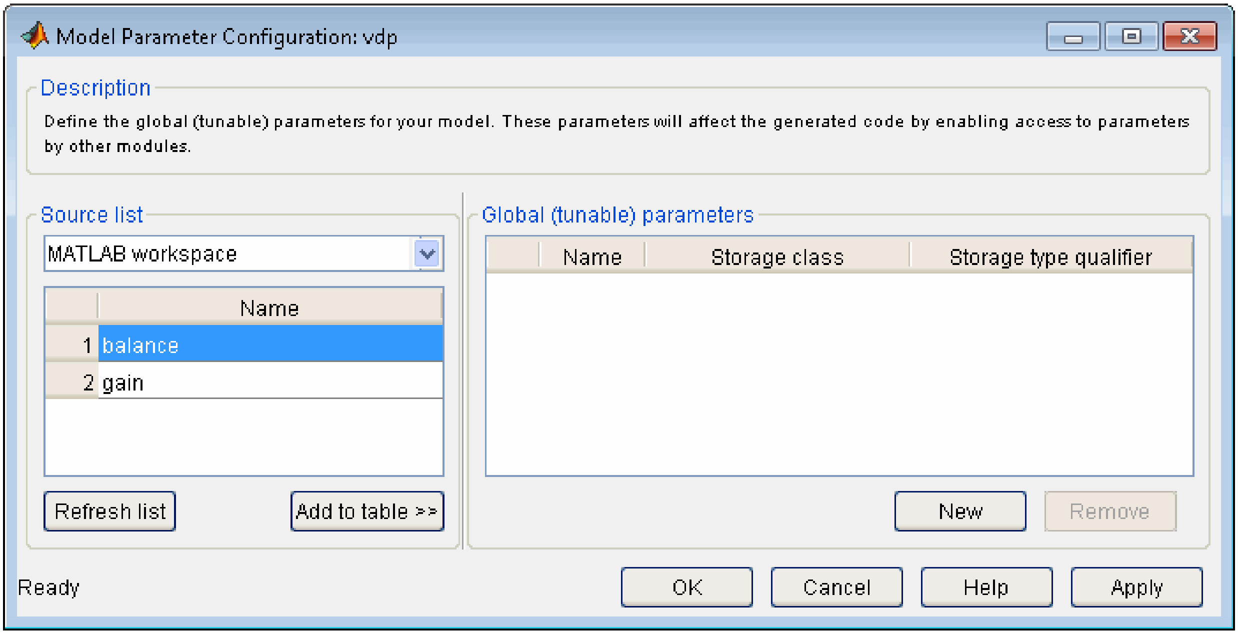
15

Model Parameter Configuration Dialog Box

Model Parameter Configuration Dialog Box

The **Model Parameter Configuration** dialog box allows you to declare specific tunable parameters when you set **Default parameter behavior** to `Inlined`. The parameters that you select appear in the generated code as tunable parameters. For more information about **Default parameter behavior**, see “Default parameter behavior” (Simulink Coder).

To declare tunable parameters, use `Simulink.Parameter` objects instead of the **Model Parameter Configuration** dialog box. See “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder).



Model Parameter Configuration: vdp

Description

Define the global (tunable) parameters for your model. These parameters will affect the generated code by enabling access to parameters by other modules.

Source list: MATLAB workspace

	Name
1	balance
2	gain

Global (tunable) parameters

	Name	Storage class	Storage type qualifier

Ready

Note Simulink Coder software ignores the settings of this dialog box if a model contains references to other models. However, you can still generate code that uses tunable parameters with model references, using `Simulink.Parameter` objects. See “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder).

The dialog box has the following controls.

Source list

Displays a list of workspace variables. The options are:

- MATLAB workspace — Lists all variables in the MATLAB workspace that have numeric values.
- Referenced workspace variables — Lists only those variables referenced by the model.

Refresh list

Updates the source list. Click this button if you have added a variable to the workspace since the last time the list was displayed.

Add to table

Adds the variables selected in the source list to the adjacent table of tunable parameters.

New

Defines a new parameter and adds it to the list of tunable parameters. Use this button to create tunable parameters that are not yet defined in the MATLAB workspace.

Note This option does not create the corresponding variable in the MATLAB workspace. You must create the variable yourself.

Storage class

Used for code generation. For more information, see “Storage class”.

Storage type qualifier

Used for code generation. For more information, see “Type qualifier”.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” (Simulink Coder)
- “How Generated Code Stores Internal Signal, State, and Parameter Data” (Simulink Coder)