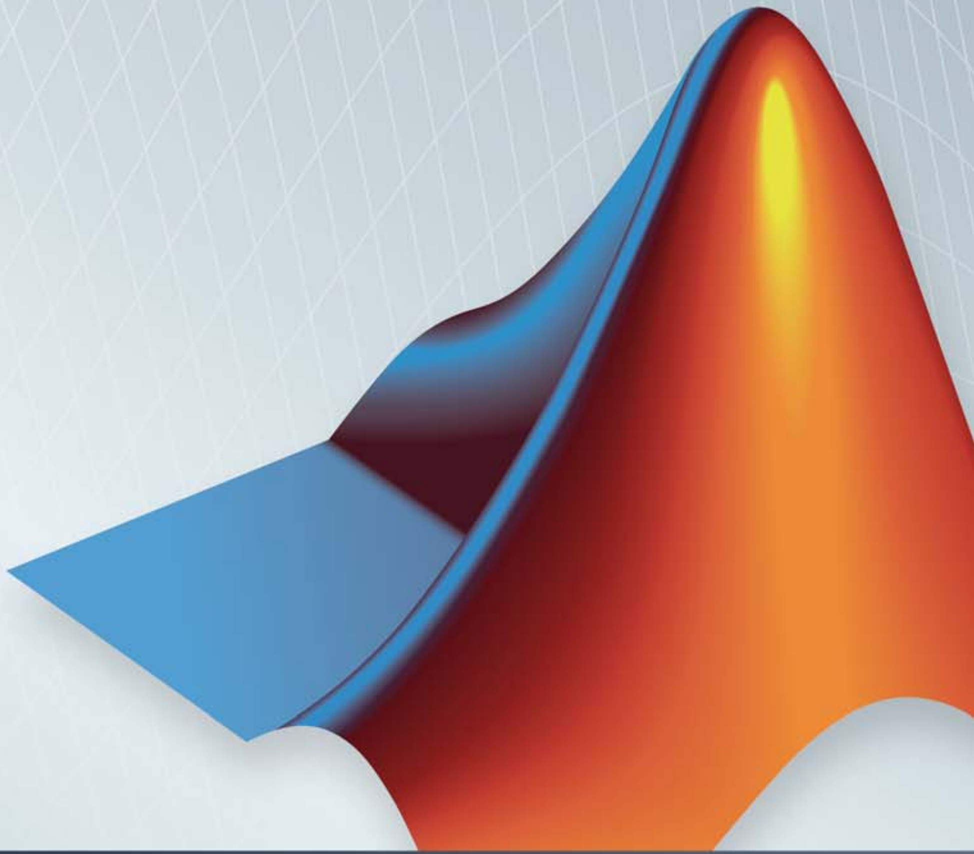
Simulink®

Reference

R2013a



MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

☎ 508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Simulink® Reference

© COPYRIGHT 2002–2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

July 2002	Online only	Revised for Simulink 5 (Release 13)
April 2003	Online only	Revised for Simulink 5.1 (Release 13SP1)
April 2004	Online only	Revised for Simulink 5.1.1 (Release 13SP1+)
June 2004	Online only	Revised for Simulink 6 (Release 14)
October 2004	Online only	Revised for Simulink 6.1 (Release 14SP1)
March 2005	Online only	Revised for Simulink 6.2 (Release 14SP2)
September 2005	Online only	Revised for Simulink 6.3 (Release 14SP3)
March 2006	Online only	Revised for Simulink 6.4 (Release 2006a)
September 2006	Online only	Revised for Simulink 6.5 (Release 2006b)
March 2007	Online only	Revised for Simulink 6.6 (Release 2007a)
September 2007	Online only	Revised for Simulink 7.0 (Release 2007b)
March 2008	Online only	Revised for Simulink 7.1 (Release 2008a)
October 2008	Online only	Revised for Simulink 7.2 (Release 2008b)
March 2009	Online only	Revised for Simulink 7.3 (Release 2009a)
September 2009	Online only	Revised for Simulink 7.4 (Release 2009b)
March 2010	Online only	Revised for Simulink 7.5 (Release 2010a)
September 2010	Online only	Revised for Simulink 7.6 (Release 2010b)
April 2011	Online only	Revised for Simulink 7.7 (Release 2011a)
September 2011	Online only	Revised for Simulink 7.8 (Release 2011b)
March 2012	Online only	Revised for Simulink 7.9 (Release 2012a)
September 2012	Online only	Revised for Simulink 8.0 (Release 2012b)
March 2013	Online only	Revised for Simulink 8.1 (Release 2013a)

Contents

Blocks — Alphabetical List
1

Functions — Alphabetical List
2

Mask Icon Drawing Commands
3

Simulink Debugger Commands
4

Simulink Classes
5

Model and Block Parameters
6

Model Parameters **6-2**
  About Model Parameters **6-2**
  Examples of Setting Model Parameters **6-95**

Common Block Properties **6-96**

About Common Block Properties **6-96**
Examples of Setting Block Properties **6-107**

Block-Specific Parameters **6-109**

Mask Parameters **6-260**
About Mask Parameters **6-260**

7 Simulink Identifier

Simulink Identifier **7-2**

8 Model Advisor Checks

Simulink Checks **8-2**
Simulink Check Overview **8-4**
Identify unconnected lines, input ports, and output ports **8-5**
Check root model Inport block specifications **8-6**
Check optimization settings **8-7**
Check for parameter tunability information ignored for referenced models **8-10**
Check for implicit signal resolution **8-11**
Check for optimal bus virtuality **8-12**
Check for Discrete-Time Integrator blocks with initial condition uncertainty **8-13**
Identify disabled library links **8-14**
Identify parameterized library links **8-15**
Identify unresolved library links **8-16**
Identify model reference variants and variant subsystems that override variant choice **8-17**
Identify configurable subsystem blocks for converting to variant subsystem blocks **8-18**
Check usage of function-call connections **8-18**

Check signal logging save format 8-19
Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues 8-21
Check if read/write diagnostics are enabled for data store blocks 8-23
Check data store block sample times for modeling errors .. 8-25
Check for potential ordering issues involving data store access 8-26
Check for partial structure parameter usage with bus signals 8-28
Check for calls to slDataTypeAndScale 8-30
Check bus usage 8-32
Check for potentially delayed function-call subsystem return values 8-34
Identify block output signals with continuous sample time and non-floating point data type 8-35
Check Merge block usage 8-36
Check consistency of initialization parameters for Outport and Merge blocks 8-37
Check for non-continuous signals driving derivative ports 8-55
Runtime diagnostics for S-functions 8-57
Check file for foreign characters 8-59
Check model for known block upgrade issues 8-64
Check model for known block upgrade issues requiring compile time information 8-65
Check that the model is saved in SLX format 8-68
Check Model History properties 8-70
Analyze model hierarchy for upgrade issues 8-72
Check and update masked blocks in library to use promoted parameters 8-74

9 Performance Advisor Checks

Simulink Performance Advisor Checks 9-2
Simulink Performance Advisor Check Overview 9-2
Baseline 9-3
Checks that require Update Diagram 9-3
Checks that require simulation to run 9-3
Check Accelerator settings 9-3

Create Baseline **9-3**
Identify resource intensive diagnostic settings **9-4**
Check optimization settings **9-4**
Identify inefficient lookup table blocks **9-5**
Identify Interpreted MATLAB Function blocks **9-5**
Check MATLAB Function block debug settings **9-6**
Check Stateflow block debug settings **9-6**
Identify simulation target settings **9-6**
Check model reference rebuild setting **9-7**
Check Model Reference Parallel Build **9-7**
Check Delay block circular buffer setting **9-9**
Check solver type selection **9-9**
Select normal or accelerator simulation mode **9-11**

Simulink Limits

10

Maximum Size Limits of Simulink Models **10-2**

Index

1

Blocks — Alphabetical List

Abs

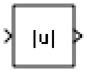
Purpose

Output absolute value of input

Library

Math Operations

Description



The Abs block outputs the absolute value of the input.

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

If you...	The block...	And...
Select this check box	Saturates to the most positive value of the integer data type	• 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.

Data Type Support

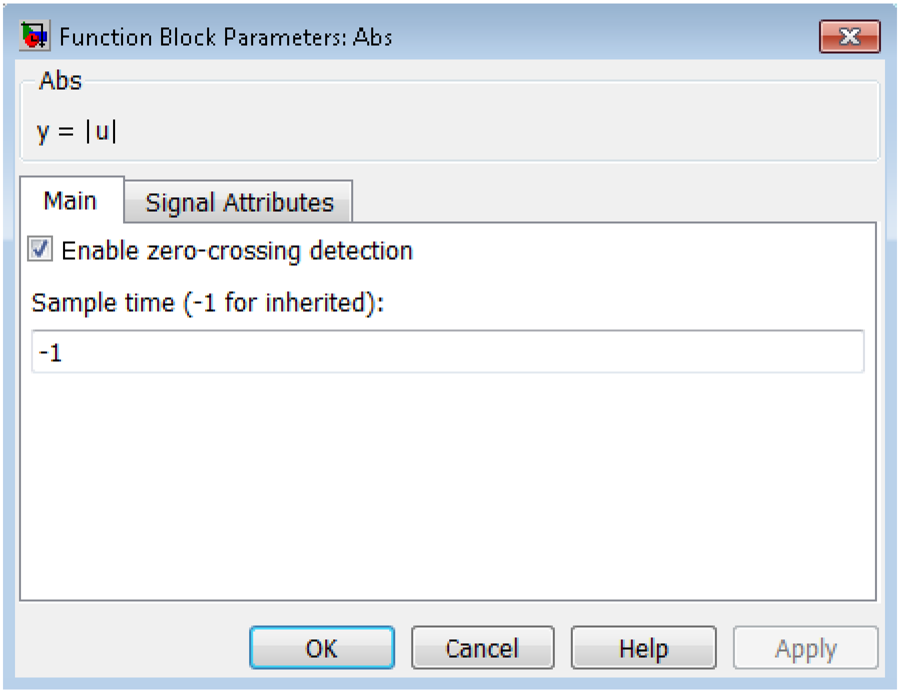
The Abs block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box

The **Main** pane of the Abs block dialog box appears as follows:


Function Block Parameters: Abs

Abs

y = |u|

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

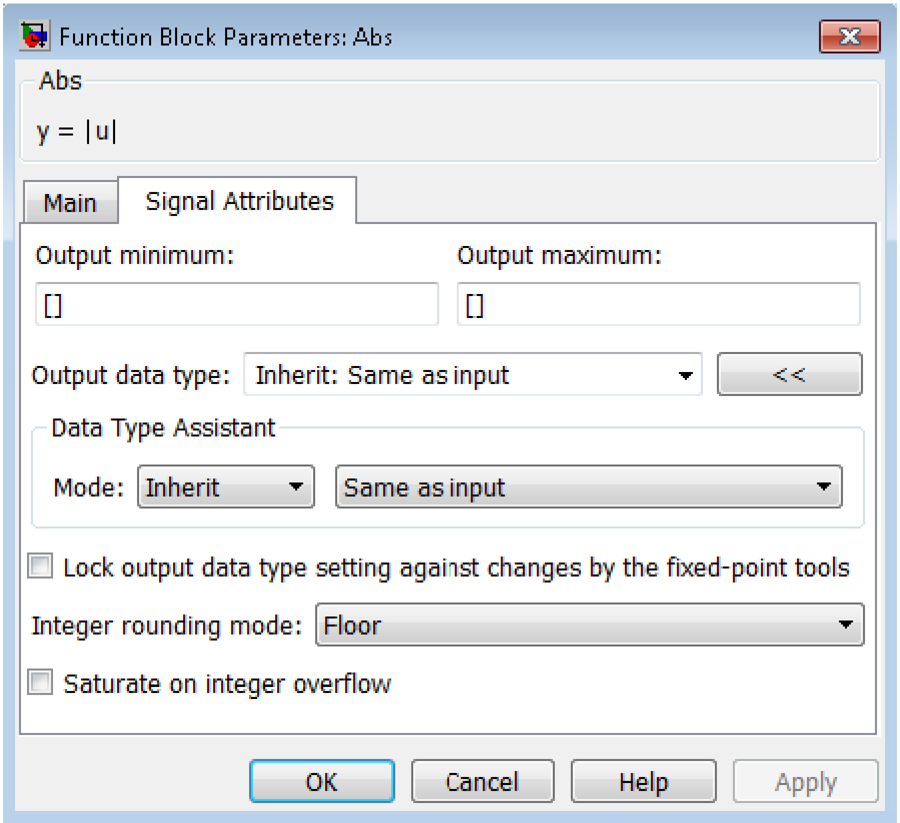
Enable zero-crossing detection

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

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 upon its context within the model. For more information, see “Sample Time”.

The **Signal Attributes** pane of the Abs block dialog box appears as follows:



Function Block Parameters: Abs

Abs

y = |u|

Output minimum: []

Output maximum: []

Output data type: Inherit: Same as input

Data Type Assistant

Mode: Inherit — Same as input

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

Integer rounding mode: Floor

☐ Saturate on integer overflow

Output minimum

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

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

Output maximum

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

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

Output data type

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

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

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

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

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

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

Integer rounding mode

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

Saturate on integer overflow

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

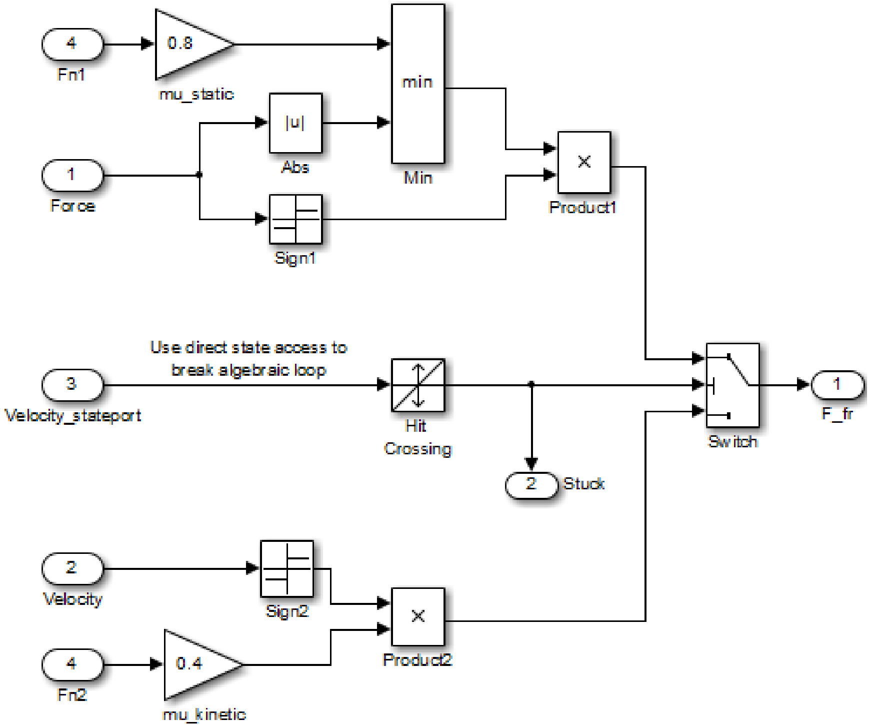
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

Usage as an Input to a MinMax Block

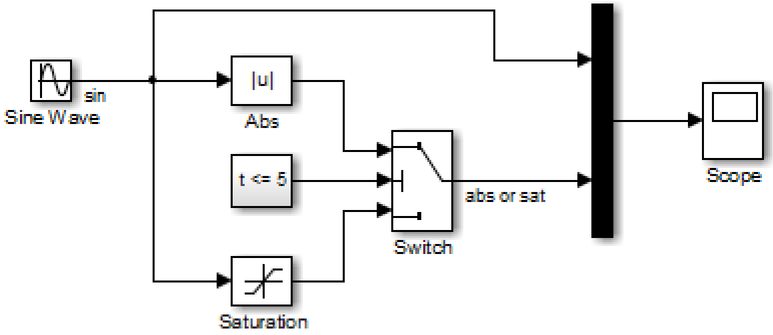
The `sldemo_hardstop` model shows how you can use the Abs block as an input to the MinMax block.

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


4
Fn1
0.8
mu_static
min
Min
|u|
Abs
1
Force
Sign1
×
Product1
Use direct state access to break algebraic loop
3
Velocity_stateport
Hit Crossing
2
Stuck
Switch
1
F_fr
2
Velocity
Sign2
×
Product2
4
Fn2
0.4
mu_kinetic

Usage as an Input to a Switch Block

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


sin
Sine Wave
|u|
Abs
t <= 5
Switch
abs or sat
Saturation
Scope

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled

Action Port

Purpose

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

Library

Ports & Subsystems

Description

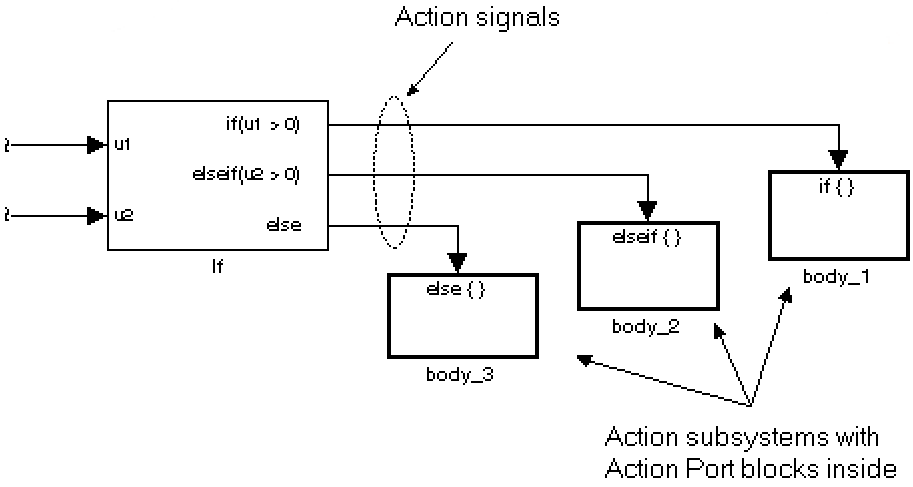
Action Port blocks implement action subsystems used in `if` and `switch` control flow statements. The If Action Subsystem and the Switch Case Action Subsystem blocks each contain an Action Port block.

Convert a Subsystem into an Action Subsystem

Use an Action Port block to convert a subsystem that has no control ports (such as an enable port, trigger port, or a function-call port) into an action subsystem.

1. Open the subsystem you want to convert.
2. Drag an If Action Subsystem or a Switch Case Action Subsystem block from the Ports & Subsystems library into your subsystem.
3. Open the action subsystem and copy the Action Port block.
4. Go up a level to your subsystem and paste the Action Port block.
5. Delete the action subsystem you dragged into your model.

The subsystem icon in your model will change to include an Action port. The resulting action subsystem is atomic and executes in response to the conditional outputs of an If or Switch Case block. The diagram shows an `if-else` control flow statement using action subsystems connected to an If block.


Action signals
if(u1 > 0)
u1
elseif(u2 > 0)
u2
else
If
if { }
body_1
elseif { }
body_2
else { }
body_3
Action subsystems with
Action Port blocks inside

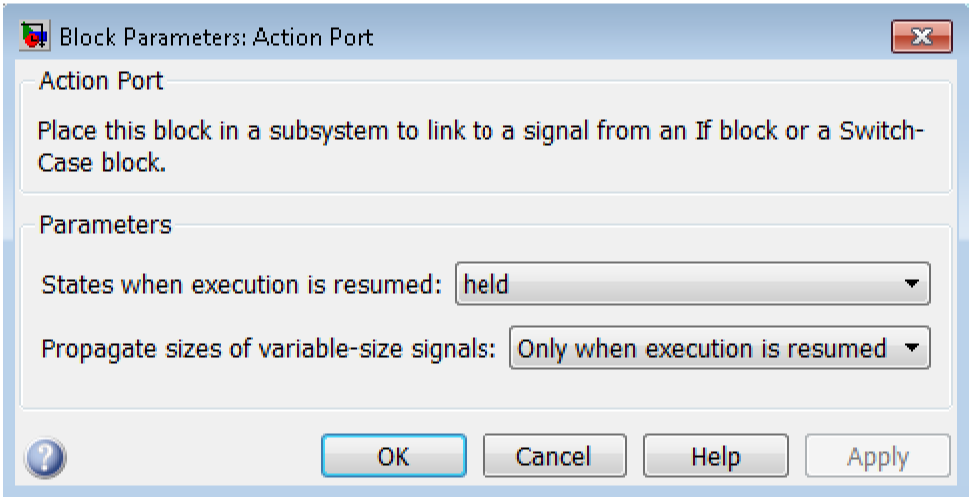
When you connect an If or Switch Case block to an action subsystem, the action port label on the subsystem icon changes to match the output port of the driving block (i.e. `if{ }`, `else{ }`, `elseif{ }`, `case{ }`, or `default{ }`).

Note All blocks in an action subsystem must run at the same rate as the driving If or Switch Case block, or should have their sample time set to `-1` (inherited).

Data Type Support

Action Port blocks do not have data inputs or outputs.

Parameters and Dialog Box



Block Parameters: Action Port

Action Port

Place this block in a subsystem to link to a signal from an If block or a Switch-Case block.

Parameters

States when execution is resumed: held

Propagate sizes of variable-size signals: Only when execution is resumed

- “States when execution is resumed” on page 1-12
- “Propagate sizes of variable-size signals” on page 1-14

States when execution is resumed

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

Settings

Default: `held`

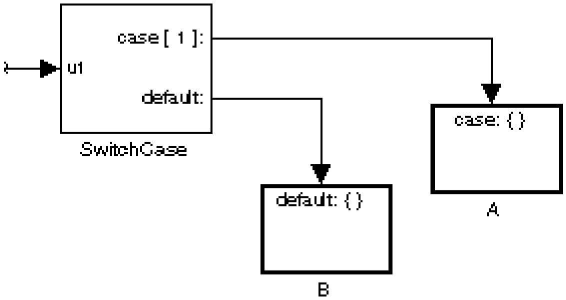
`held`

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

`reset`

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

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



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

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

Command-Line Information

Parameter: `InitializeStates`
Type: string
Value: `'held' | 'reset' |`
Default: `'held'`

Propagate sizes of variable-size signals

Specify when to propagate a variable-size signal.

Settings

Default: `Only when execution is resumed`

`Only when execution is resumed`
: Propagates variable-size signals only when reenabling the subsystem containing the Action Port block.

`During execution`
: Propagates variable-size signals at each time step.

Command-Line Information

Parameter: `PropagateVarSize`
Type: string
Value: `'Only when execution is resumed' | 'During execution'`
Default: `'Only when execution is resumed'`

Characteristics

Sample Time	Inherited from driving If or Switch Case block

See Also

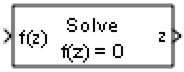
If, If Action Subsystem, Switch Case, and Switch Case Action Subsystem blocks for examples using Action Port blocks

Algebraic Constraint

Purpose Constrain input signal to zero

Library Math Operations

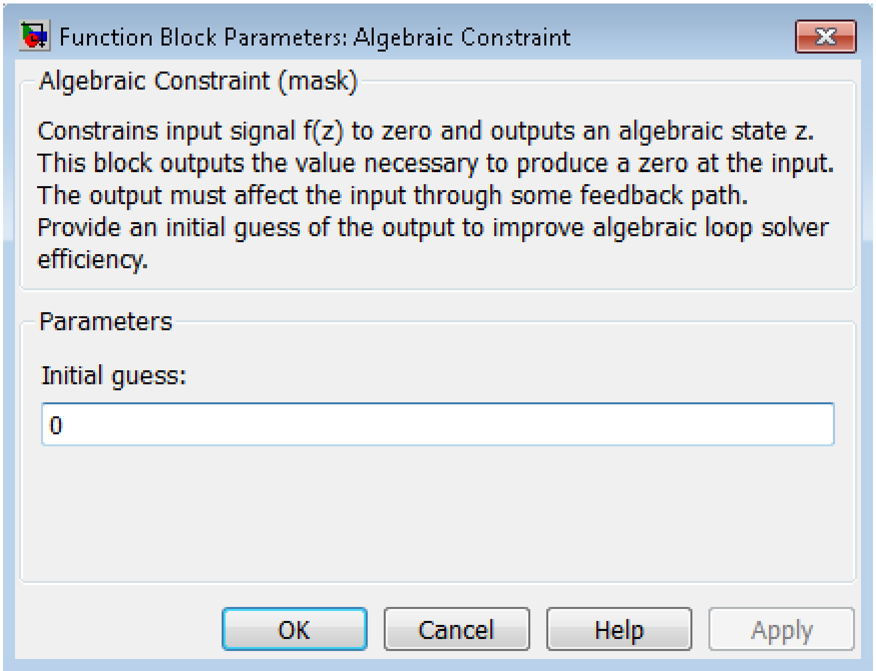
Description



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

Algorithm The Algebraic Constraint block uses a dogleg trust-region algorithm to solve algebraic loops [1], [2].

Data Type Support The Algebraic Constraint block accepts and outputs real values of type `double`.


Function Block Parameters: Algebraic Constraint
Algebraic Constraint (mask)
Parameters
OK
Cancel
Help
Apply

Constrains input signal f(z) to zero and outputs an algebraic state z. This block outputs the value necessary to produce a zero at the input. The output must affect the input through some feedback path. Provide an initial guess of the output to improve algebraic loop solver efficiency.

Initial guess: 0

Parameters and Dialog Box

Initial guess

An initial guess for the solution value. The default is `0`.

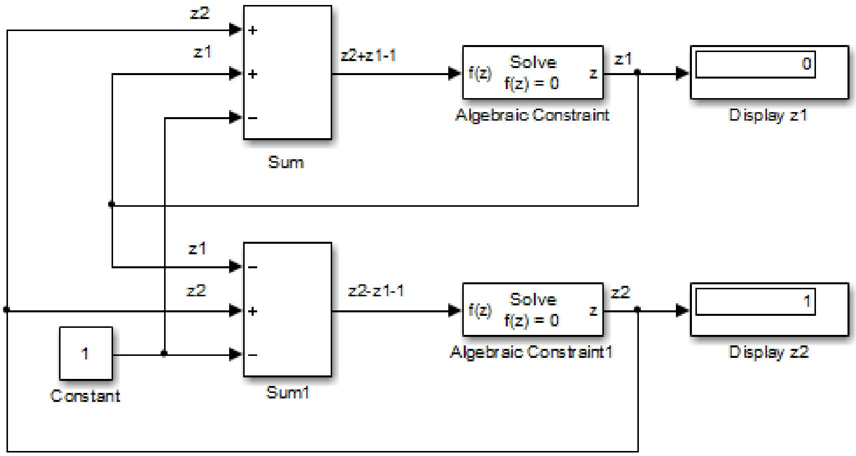
Example

By default, the **Initial guess** parameter is zero. You can improve the efficiency of the algebraic-loop solver by providing an **Initial guess** for the algebraic state z that is close to the solution value.

For example, the following model solves these equations:

```
z2 + z1 = 1
z2 - z1 = 1
```

The solution is `z2 = 1`, `z1 = 0`, as the Display blocks show.


z2
z1
+
+
−
Sum
z2+z1-1
f(z)
Solve
f(z) = 0
z
Algebraic Constraint
z1
0
Display z1
z1
z2
−
+
−
1
Constant
Sum1
z2-z1-1
f(z)
Solve
f(z) = 0
z
Algebraic Constraint1
z2
1
Display z2

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
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, NY: Gordon and Breach, 1970.

Assertion

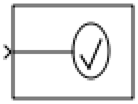
Purpose

Check whether signal is zero

Library

Model Verification

Description



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

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

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

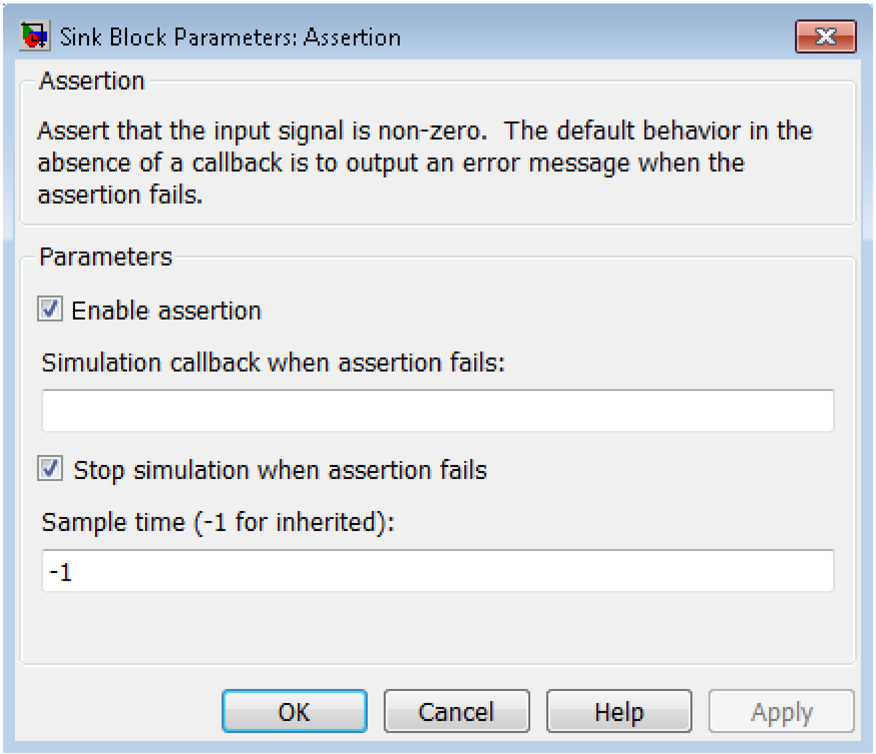
The Assertion block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is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

Note For information about how Simulink Coder™ generated code handles Model Verification blocks, see "Debug".

Data Type Support

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

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



Sink Block Parameters: Assertion

Assertion

Assert that the input signal is non-zero. The default behavior in the absence of a callback is to output an error message when the assertion fails.

Parameters

☑ Enable assertion

Simulation callback when assertion fails:

Stop simulation when assertion fails: ☑

Sample time (-1 for inherited): -1

Parameters and Dialog Box

Enable assertion

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Assertion block to halt the simulation when the block input is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

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 upon its context within the model. For more information, see "Sample Time".

Characteristics

Direct Feedthrough	No
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

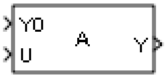
Purpose

Assign values to specified elements of signal

Library

Math Operations

Description



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

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

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

- **1**

  Index Option, select `Assign all`

- **2**

  Index Option, select `Index vector (dialog)`

  Index, enter `[1 3 5]`

- **3**

  Index Option, select `Starting index (dialog)`

  Index, enter `4`

- **4**

  Index Option, select `Starting index (port)`

- **5**

  Index Option, select `Index vector (port)`

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

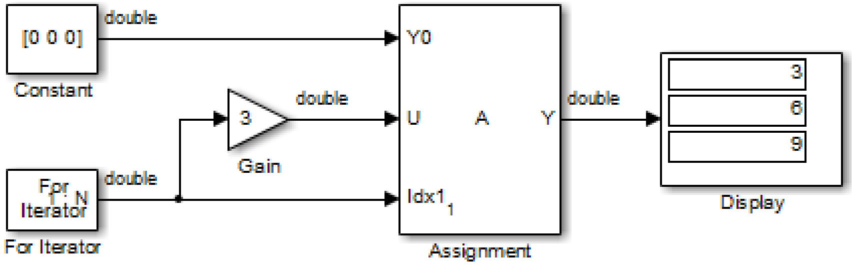
The Assignment block's data port is labeled U. The rest of this section refers to the data port as U to simplify the explanation of the block's usage.

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

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

Iterated Assignment

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

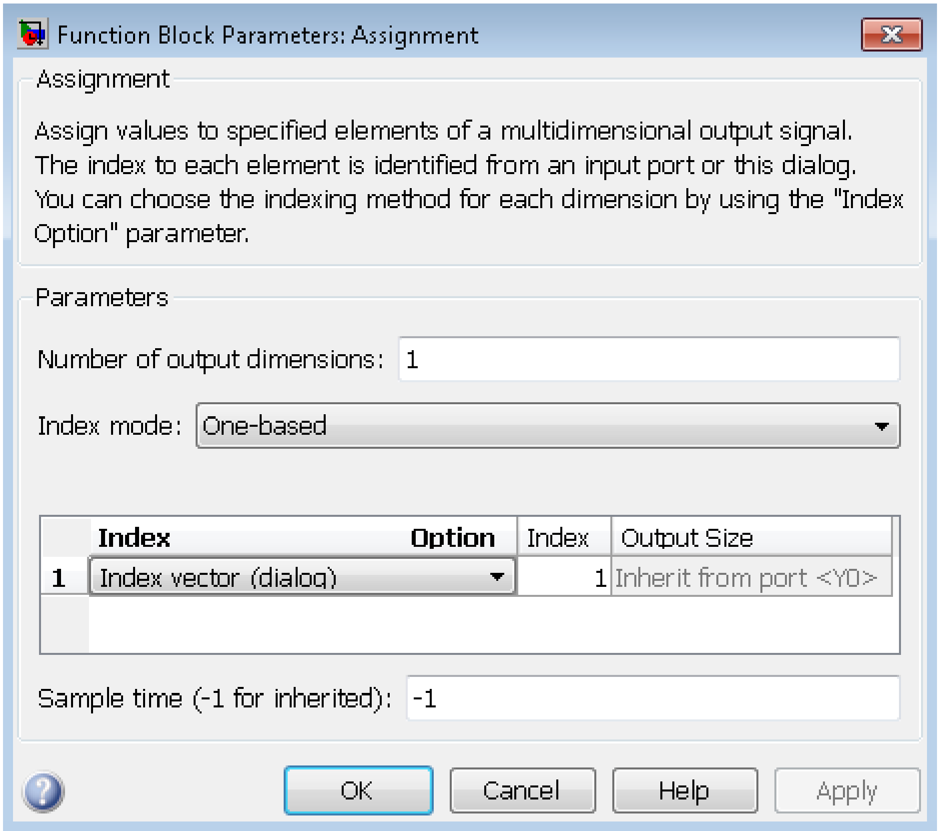

[0 0 0]
Constant
double
For Iterator
1 : N
For Iterator
double
3
Gain
double
Y0
U
A
Y
Idx1
1
Assignment
double
3
6
9
Display

Iterated assignment uses an iterator (For or While) block to generate indices for the Assignment block. On the first iteration of an iterated assignment, the Assignment block copies the first input (Y0) to the output (Y) and assigns the second input (U) to the output $Y(E_1)$. On successive iterations, the Assignment block assigns the current value of U to $Y(E_i)$, that is, without first copying Y0 to Y. These actions occur in a single time step.

Data Type Support

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

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



Assign values to specified elements of a multidimensional output signal. The index to each element is identified from an input port or this dialog. You can choose the indexing method for each dimension by using the "Index Option" parameter.

Number of output dimensions: 1

Index mode: One-based

	Index Option	Index	Output Size
1	Index vector (dialog)	1	Inherit from port <Y0>

Sample time (-1 for inherited): -1

Parameters and Dialog Box

Number of output dimensions

Enter the number of dimensions of the output signal.

Index mode

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

Index Option

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

- `Assign all`

  This is the default. All elements are assigned.

- `Index vector (dialog)`

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

- `Index vector (port)`

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

- `Starting index (dialog)`

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

- `Starting index (port)`

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

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

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

Otherwise, Y0 always initializes output port Y.

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

Index

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

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

Output Size

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

Initialize output (Y)

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

- `Initialize using input port <Y0>`

  The signal at the input port Y0 initializes the output.

- `Specify size for each dimension in table`

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

Action if any output element is not assigned

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

- `Error`
- `Warning`
- `None`

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Backlash

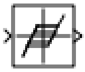
Purpose

Model behavior of system with play

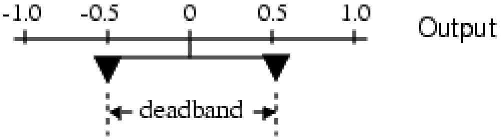
Library

Discontinuities

Description



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



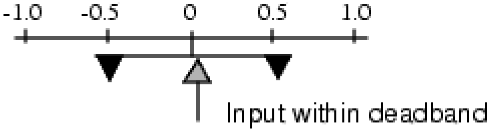
A system with play can be in one of three modes:

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

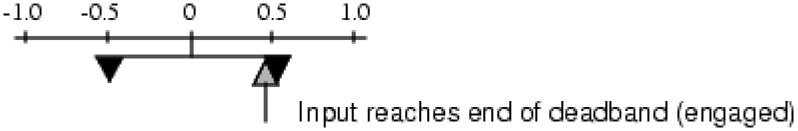
If the initial input is outside the deadband, the **Initial output** parameter value determines whether the block is engaged in a positive or negative direction, and the output at the start of the simulation is the input plus or minus half the deadband width.

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

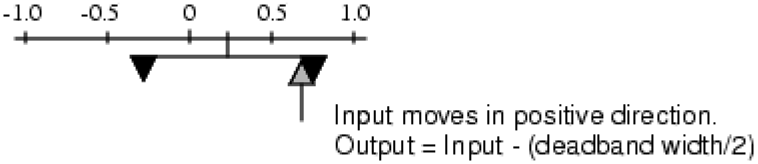
The following figures illustrate the block's operation when the initial input is within the deadband. The first figure shows the relationship between the input and the output while the system is in disengaged mode (and the default parameter values are not changed).


-1.0
-0.5
0
0.5
1.0
Input within deadband

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


-1.0
-0.5
0
0.5
1.0
Input reaches end of deadband (engaged)

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


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

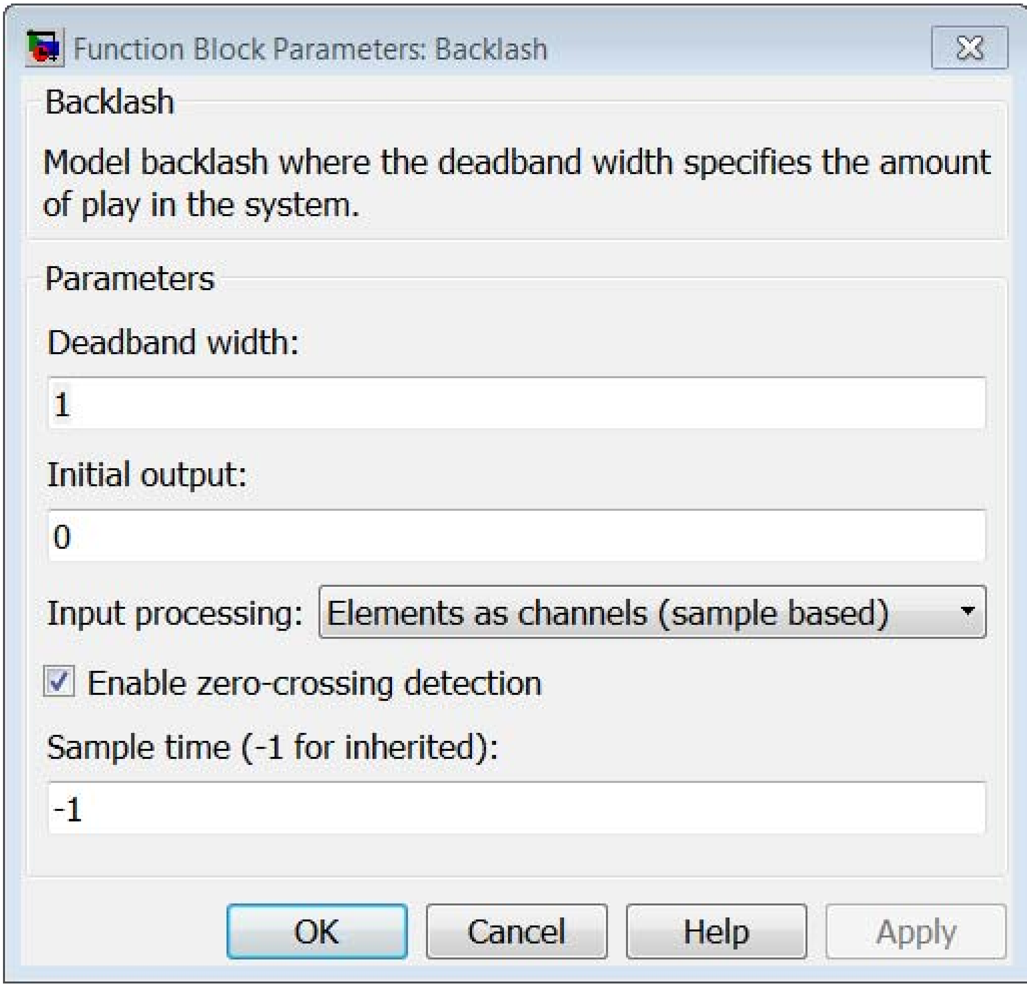
If the input reverses its direction, it disengages from the output. The output remains constant until the input either reaches the opposite end of the deadband or reverses its direction again and engages at the same end of the deadband. Now, as before, movement in the input causes equal movement in the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

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

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

Data Type Support

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


Function Block Parameters: Backlash
Backlash
Parameters
OK
Cancel
Help
Apply

Model backlash where the deadband width specifies the amount of play in the system.

Deadband width: 1

Initial output: 0

Input processing: Elements as channels (sample based)

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

Parameters and Dialog Box

Deadband width

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

Initial output

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

Input processing

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

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

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

For more information, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

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

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

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

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

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

Enable zero-crossing detection

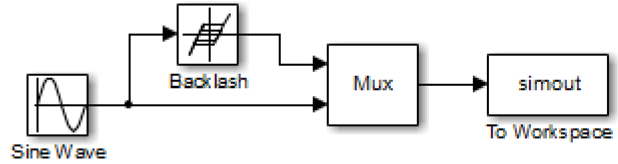
Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Sample time (-1 for inherited)

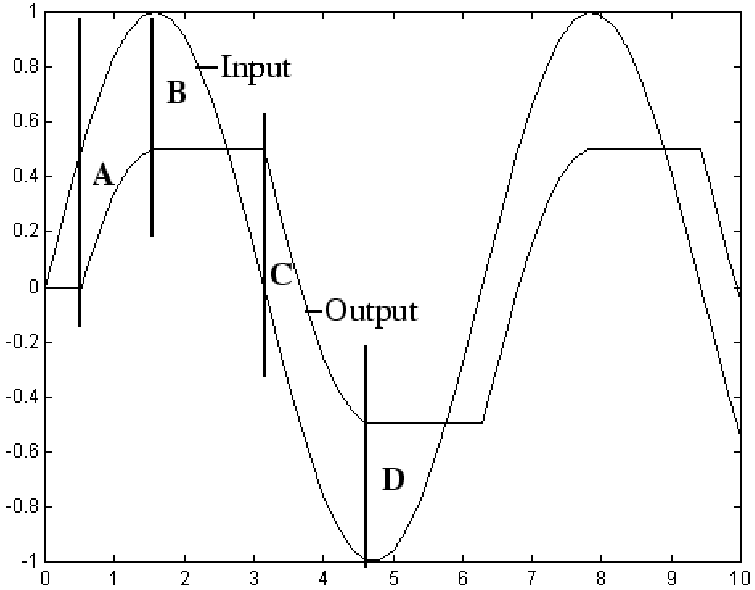
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation.

Examples

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



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



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

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

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

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

Characteristics

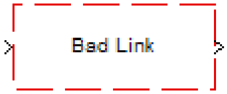
Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled

Bad Link

Purpose

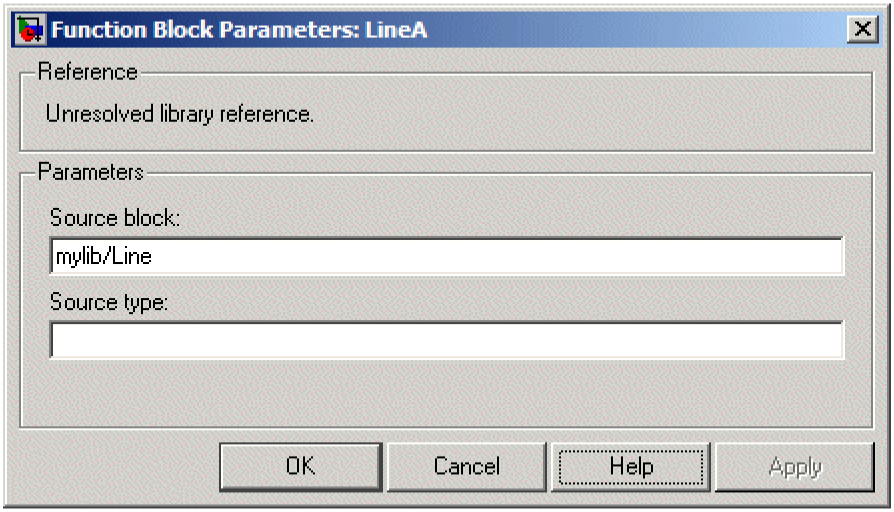
Indicate unresolved reference to library block

Description



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

Parameters and Dialog Box



Function Block Parameters: LineA

Reference

Unresolved library reference.

Parameters

Source block: mylib/Line

Source type:

Source block

Path of the library block that this link represents. To fix a bad link, edit this field to reflect the actual path of the library block. Then select Apply or OK to apply the fix and close the dialog box.

Source type

Type of library block that this link represents.

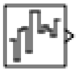
Band-Limited White Noise

Purpose Introduce white noise into continuous system

Library Sources

Description

Simulation of White Noise



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

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

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

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

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

Comparison with the Random Number Block

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

Usage with the Averaging Power Spectral Density Block

The Band-Limited White Noise block specifies a two-sided spectrum, where the units are Hz. The Averaging Power Spectral Density block specifies a one-sided spectrum, where the units are the square of the magnitude per unit radial frequency: 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:

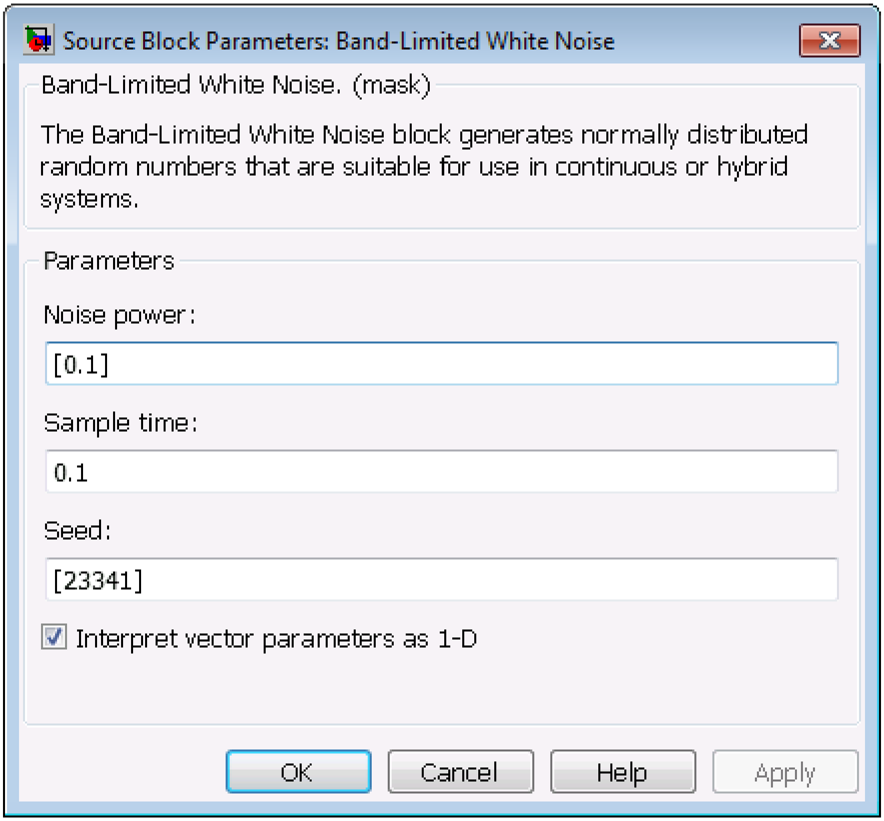
- 1/2 is the factor for converting from a two-sided to one-sided spectrum
- 2π is the factor for converting from Hz to rad/sec

Algorithm

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

Data Type Support

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


Source Block Parameters: Band-Limited White Noise

Band-Limited White Noise. (mask)

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

Parameters

Noise power:

[0.1]

Sample time:

0.1

Seed:

[23341]

☑ Interpret vector parameters as 1-D

OK
Cancel
Help
Apply

Parameters and Dialog Box

Noise power

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

Sample time

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

Seed

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

Interpret vector parameters as 1-D

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

Examples

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

- `slexAircraftExample`
- `sldemo_radar_eml`

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of the **Noise power** and **Seed** parameters and output
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

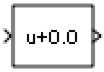
Random Number

Purpose Add bias to input

Library Math Operations

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

$$Y = U + bias,$$

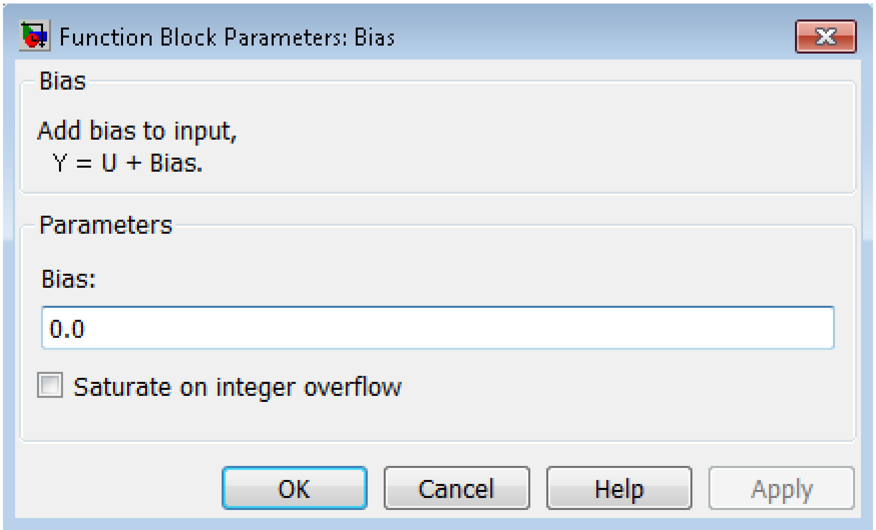


where U is the block input and Y is the output.

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

- Floating point
- Built-in integer
- Fixed point

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



Add bias to input,
Y = U + Bias.

Bias: 0.0

☐ Saturate on integer overflow

Parameters and Dialog Box

Bias

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

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles	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.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
	out-of-range signals. For more information, see "Checking 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. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	Yes
States	0
Dimensionalized	Yes
Zero-Crossing Detection	No

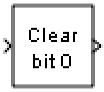
Purpose

Set specified bit of stored integer to zero

Library

Logic and Bit Operations

Description



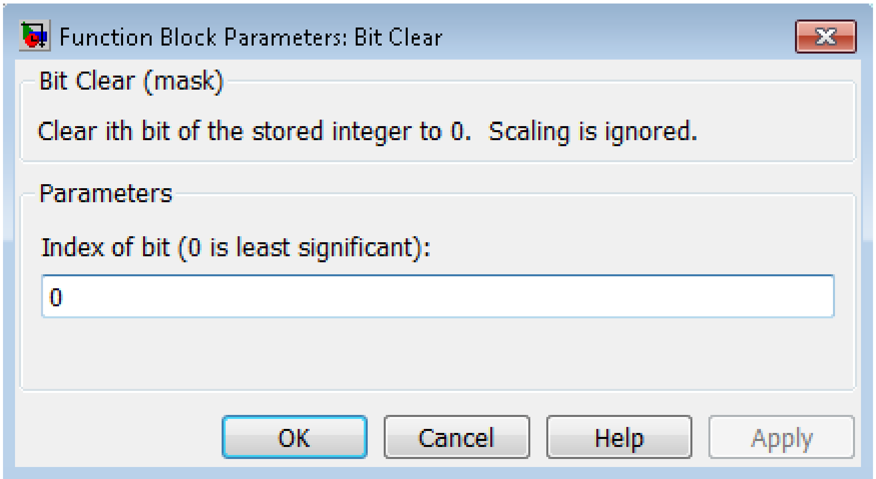
The Bit Clear block sets the specified bit, given by its index, of the stored integer to zero. Scaling is ignored.

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

Data Type Support

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

Parameters and Dialog Box



Function Block Parameters: Bit Clear

Bit Clear (mask)

Clear ith bit of the stored integer to 0. Scaling is ignored.

Parameters

Index of bit (0 is least significant): 0

Index of bit

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

Examples

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

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Bit Set

Bit Set

Purpose

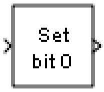
Set specified bit of stored integer to one

Library

Logic and Bit Operations

Description

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

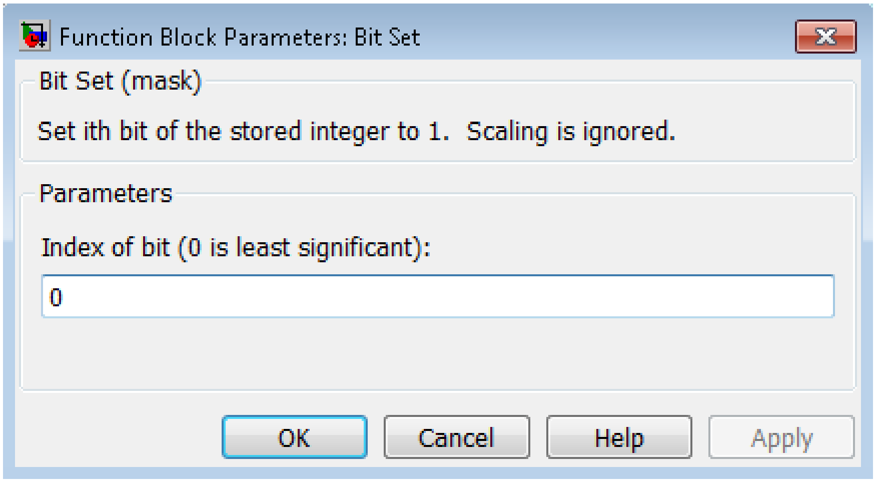


You can specify the bit to be set to one with the **Index of bit** parameter, where bit zero is the least significant bit.

Data Type Support

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

Parameters and Dialog Box



Function Block Parameters: Bit Set

Bit Set (mask)

Set ith bit of the stored integer to 1. Scaling is ignored.

Parameters

Index of bit (0 is least significant):

0

Index of bit

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

Examples

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

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Bit Clear

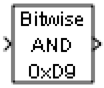
Purpose

Specified bitwise operation on inputs

Library

Logic and Bit Operations

Description



Bitwise Operations

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

You can select one of the following bitwise operations:

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

Restrictions on Block Operations

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

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

Behavior of Inputs and Outputs

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

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

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

Bit Mask Behavior

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

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

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

Bit Set and Bit Clear Operations

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

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

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

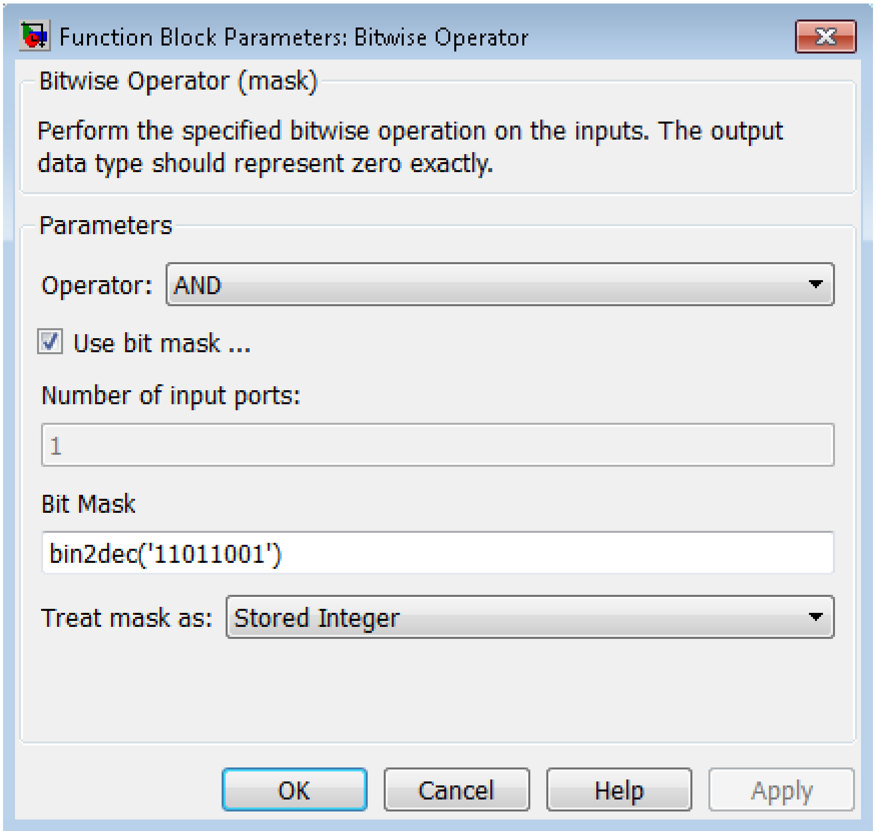
Data Type Support

The Bitwise Operator block supports the following data types:

- Built-in integer
- Fixed point
- Boolean

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

Parameters and Dialog Box



Function Block Parameters: Bitwise Operator

Bitwise Operator (mask)

Perform the specified bitwise operation on the inputs. The output data type should represent zero exactly.

Parameters

Operator: AND

☑ Use bit mask ...

Number of input ports:

1

Bit Mask

bin2dec('11011001')

Treat mask as: Stored Integer

Operator

Specify the bitwise logical operator for the block operands.

Use bit mask

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

Number of input ports

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

Bit Mask

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

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

Treat mask as

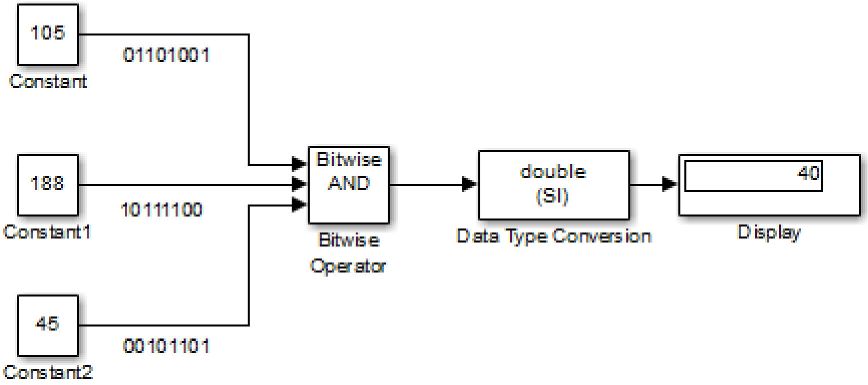
Specify whether to treat the mask as a real-world value or a stored integer. This parameter is available only when you select **Use bit mask**.

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

Examples

Unsigned Inputs for the Bitwise Operator Block

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

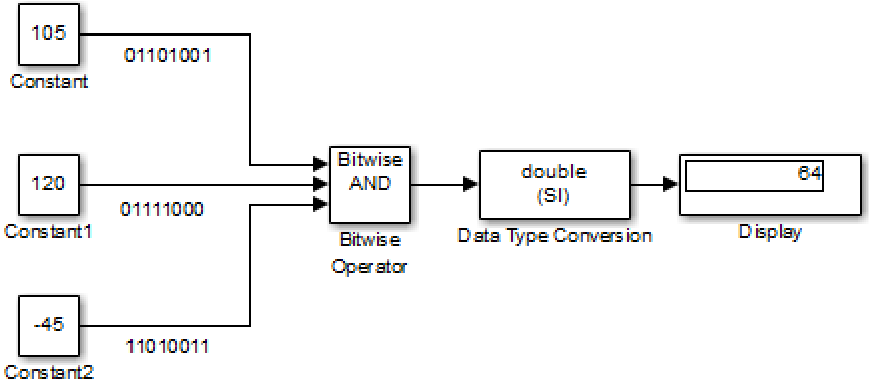

105
Constant
01101001
188
Constant1
10111100
45
Constant2
00101101
Bitwise AND
Bitwise Operator
double (SI)
Data Type Conversion
40
Display

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.


105
Constant
01101001
120
Constant1
01111000
-45
Constant2
11010011
Bitwise AND
Bitwise Operator
double (SI)
Data Type Conversion
64
Display

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

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

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	Yes, of inputs
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Logical Operator

Block Support Table

Purpose

View data type support for Simulink blocks

Library

Model-Wide Utilities

Description

Block Support
Table

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

Data Type Support

Not applicable

Parameters and Dialog Box

Not applicable

Characteristics

Not applicable

Alternatives

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

Purpose

Replace specified bus elements

Library

Signal Routing

Description

The Bus Assignment block assigns signals connected to its Assignment input ports (`:=`) to specified elements of the bus connected to its Bus input port, replacing the signals previously assigned to those elements. The change does not affect the signals themselves, it affects only the composition of the bus. Signals not replaced are unaffected by the replacement of other signals. For information about buses, see:

- "Composite Signals"
- "Create and Access a Bus"

Connect the bus to be changed to the first input port. 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. The signal connected to the assignment port must have the same structure, data type, and numeric type as the bus element to which it corresponds.

You cannot use the Bus Assignment block to replace a bus that is nested within another bus. No element selected in the dialog box for replacement can be a bus, and no signal connected to an Assignment port can be a bus.

All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. All buses and signals input to a Bus Assignment block that modifies a nonvirtual bus must therefore have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus, to allow the signal or bus to be included in a nonvirtual bus. See "Virtual and Nonvirtual Buses" for more information.

By default, Simulink implicitly converts a non-bus signal to a bus signal to support connecting the signal to a Bus Assignment or Bus Selector block. To prevent Simulink from performing that conversion, in the

Model Configuration Parameters > Diagnostics > Connectivity pane, set the "Non-bus signals treated as bus signals" diagnostic to `warning` or `error`.

By default, Simulink repairs broken selections in the Bus Assignment and Bus Selector block parameters dialog boxes that are due to upstream bus hierarchy changes. Simulink generates a warning to highlight that it made changes. 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`.

The following limitations apply to working with arrays of buses, when using the Bus Assignment block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

- You cannot connect an array of buses signal to a Bus Assignment block. To work with an array of buses signal, first 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.
- You cannot assign into a sub-bus that is an array of buses.

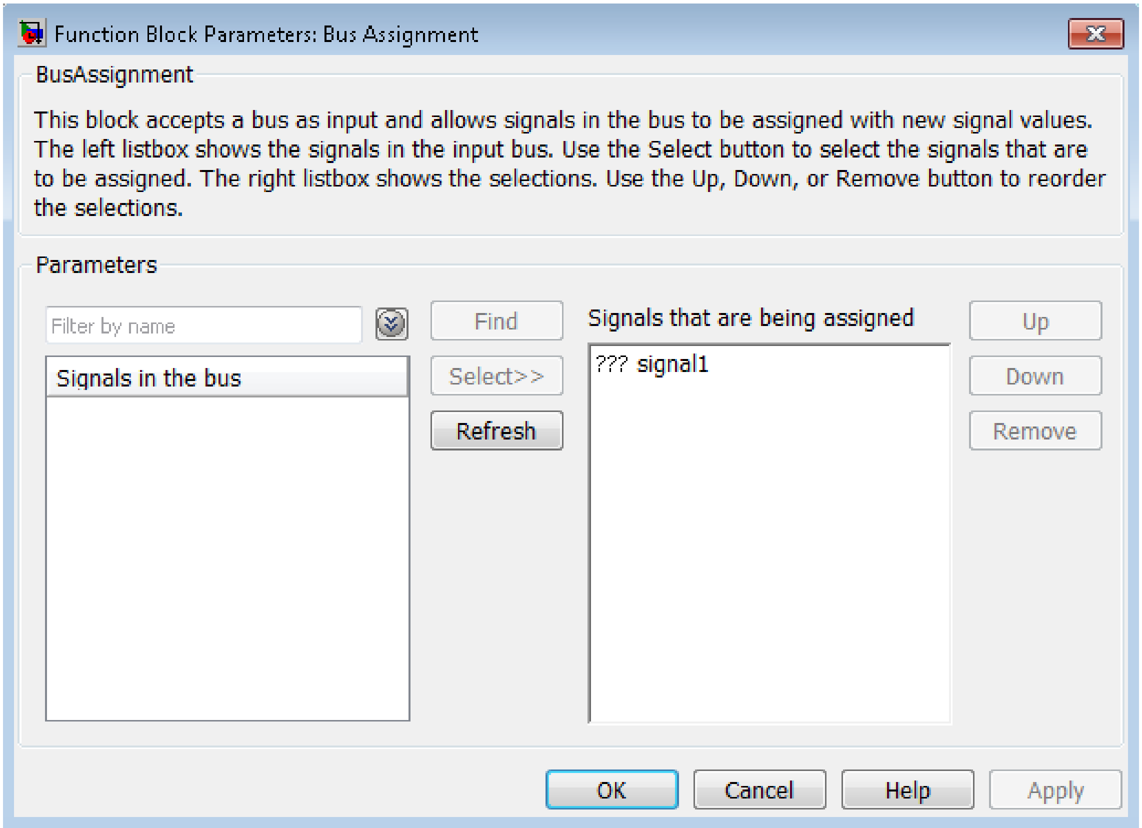
Data Type Support

The bus input port of the Bus Assignment block accepts and outputs real or complex values of any data type that Simulink supports, including fixed-point and enumerated data types. The assignment input ports accept the same data types as the bus elements to which they correspond.

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

Parameters and Dialog Box

The Bus Assignment dialog box appears as follows:



Function Block Parameters: Bus Assignment
BusAssignment

This block accepts a bus as input and allows signals in the bus to be assigned with new signal values. The left listbox shows the signals in the input bus. Use the Select button to select the signals that are to be assigned. The right listbox shows the selections. Use the Up, Down, or Remove button to reorder the selections.

Parameters
Filter by name
Find
Select>>
Refresh
Signals in the bus
Signals that are being assigned
??? signal1
Up
Down
Remove
OK
Cancel
Help
Apply

Signals in the bus

Displays the names of the signals contained by the bus at the block's Bus input port. Click any item in the list to select it. To find the source of the selected signal, click the adjacent **Find** button. Simulink opens the subsystem containing the signal source and highlights the source's icon. Use the **Select>>** button to move the currently selected signal into the adjacent list of signals to be assigned values (see **Signals that are being assigned** below). To refresh the display (e.g., to reflect modifications to the bus connected to the block), click the adjacent **Refresh** button.

Signals that are being assigned

Lists the 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**, and **Remove** buttons. Port connectivity is maintained when the signal order is changed.

Three question marks (`???`) before the name of a bus element indicate that the input bus no longer contains an element of that name, for example, because the bus has changed since the last time you refreshed the Bus Assignment block's input and bus element assignment lists. You can fix the problem either by modifying the bus to include a signal of the specified name or by removing the name from the list of bus elements to be assigned values.

Enable regular expression

To display this parameter, select the **Options** button on the right-hand side of the **Filter by name** edit box (⏬).

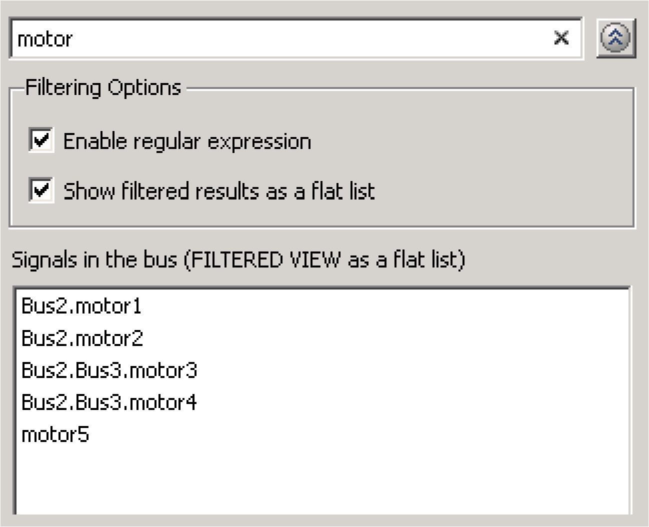
Enables the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see "Regular Expressions".

The default is `On`. If you disable use of MATLAB regular expressions for filtering signal names, filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Show filtered results as a flat list

To display this parameter, select the **Options** button on the right-hand side of the **Filter by name** edit box (⏬).

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.


motor
Filtering Options
☑ Enable regular expression
☑ Show filtered results as a flat list
Signals in the bus (FILTERED VIEW as a flat list)

Bus2.motor1
Bus2.motor2
Bus2.Bus3.motor3
Bus2.Bus3.motor4
motor5

The default is `Off`, which displays the filtered list using a tree format.

Characteristics

Multidimensionalized	Yes
Virtual	Yes, if the input bus is virtual For more information, see "Virtual Blocks" in the Simulink documentation.

See Also

- "Composite Signals"
- "Create and Access a Bus"
- Bus Creator

- Bus Selector
- Bus to Vector
- `Simulink.Bus`
- `Simulink.Bus.cellToObject`
- `Simulink.Bus.createObject`
- `Simulink.BusElement`
- `Simulink.Bus.objectToCell`
- `Simulink.Bus.save`

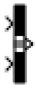
Purpose

Create signal bus

Library

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 ports that you specify. Connect to the resulting input ports those signals that you want to group. For information about buses, see:

- "Composite Signals"
- "Create and Access a Bus"

The signals in the bus are ordered from the top input port to the bottom input port. See "How to Rotate a Block" in for a description of the port order for various block orientations.

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

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

You can use an array of buses as an input signal to a Bus Creator block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

Naming Signals

The Bus Creator block assigns a name to each signal on the bus that it creates. This allows you to refer to signals by name when you are searching for their sources (see "Browsing Bus Signals" on page 1-66) or selecting signals for connection to other blocks.

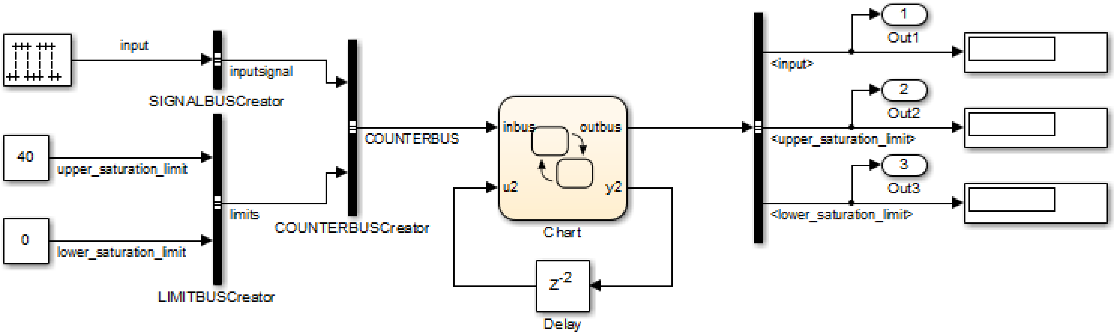
The block offers two bus signal naming options. You can specify that:

- Each signal on the bus inherits the name of the signal connected to the bus (the default).

  Inputs to a Bus Creator block must have unique names. If there are duplicate names, the Bus Creator block appends `(signal#)` to all input signal names, where # is the input port index.

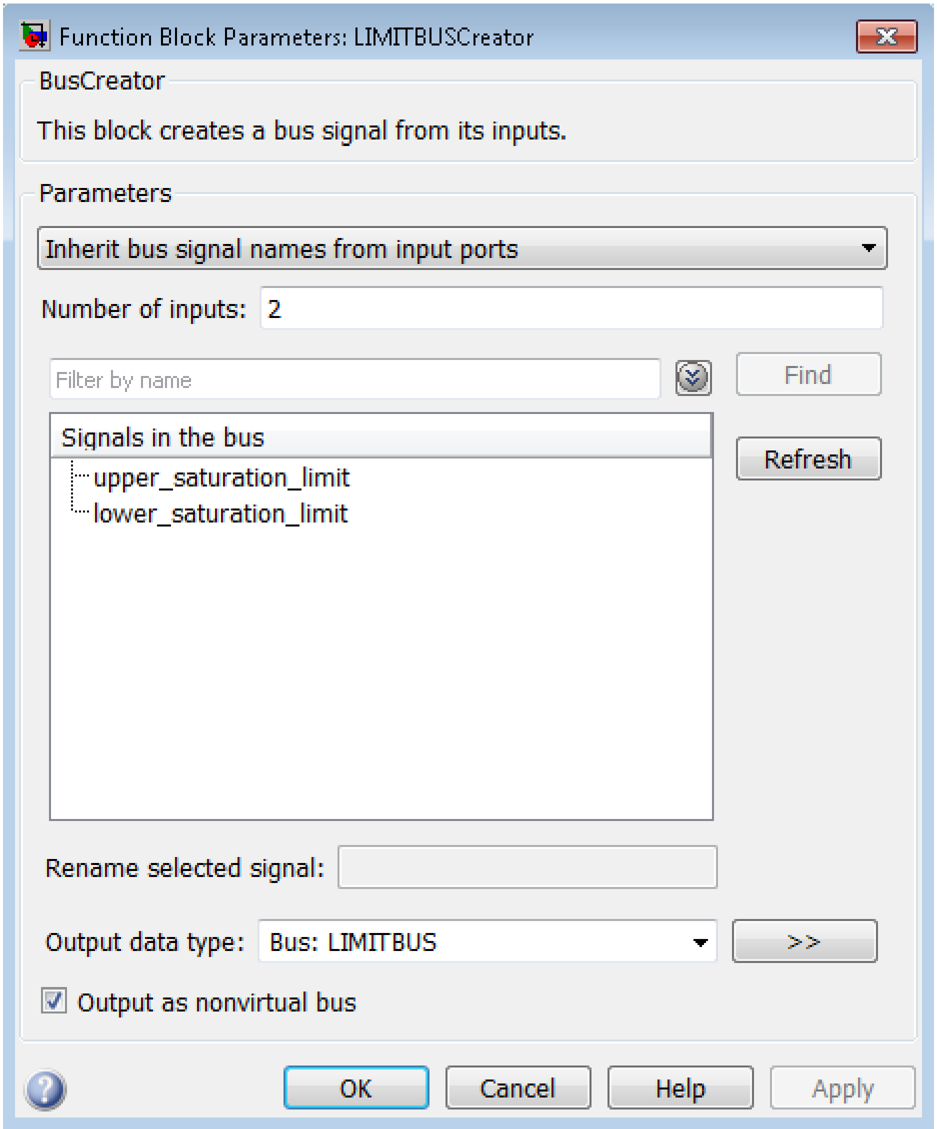
- Each input signal must have a specific name.

To specify that bus signals inherit their names from input ports, select `Inherit bus signal names from input ports` from the list box on the Block Parameters dialog box. The names of the inherited bus signals appear in the **Signals in the bus** list box. For example, suppose that you have the following model:


input
inputsignal
SIGNALBUSCreator
40
upper_saturation_limit
0
lower_saturation_limit
limits
LIMITBUSCreator
COUNTERBUS
COUNTERBUSCreator
inbus
outbus
u2
y2
Chart
z^-2
Delay
<input>
<upper_saturation_limit>
<lower_saturation_limit>
1
Out1
2
Out2
3
Out3

Copyright 2006-2011 The MathWorks, Inc.

In the dialog box for `LIMITBUSCreator`, the signals `upper_saturation_limit` and `lower_saturation_limit` appear in the **Signals in the bus** list box:



Function Block Parameters: LIMITBUSCreator

BusCreator

This block creates a bus signal from its inputs.

Parameters

Inherit bus signal names from input ports

Number of inputs: 2

Signals in the bus
upper_saturation_limit
lower_saturation_limit

Rename selected signal:

Output data type: Bus: LIMITBUS

☑ Output as nonvirtual bus

The Bus Creator block generates names for bus signals whose corresponding inputs do not have names. The names are of the form `signaln`, where `n` is the number of the port to which the input signal is connected.

You can change the name of any signal by editing its name on the block diagram or in the Signal Properties dialog box. If you change the signal name using either approach while the Bus Creator block dialog box is open, you need to update the name in the dialog box. To do so, close and reopen the dialog box or click the **Refresh** button next to the **Signals in the bus** list.

To specify that the bus inputs must have specific names, select `Require input signal names to match signals below` from the list box in the block parameter dialog box. The block parameter dialog box displays the names of the signals currently connected to its inputs, or a generated name (for example, `signal2`) for an anonymous input. Then you can use the parameter dialog box to change the required names of the block inputs.

To change the required signal name, select the signal in the **Signals in the bus** list. The name of the selected signal appears in the **Rename selected signal** field. Edit the name in the field and click **Apply** or **OK**.

Browsing Bus Signals

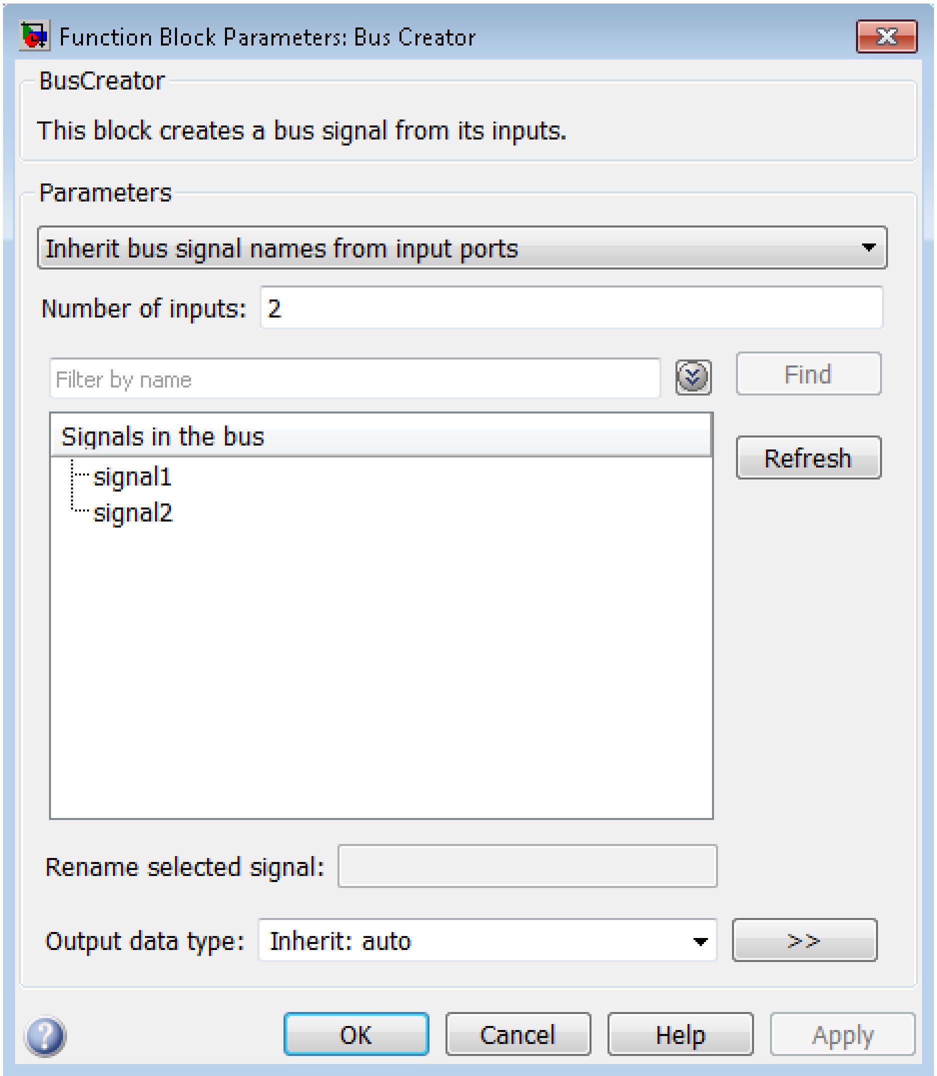
The **Signals in the bus** list on a Bus Creator Block Parameters dialog box displays a list of the signals entering the block. A plus sign (+) next to a signal indicates that the signal is itself a bus. To display the contents of the bus, click the plus sign. If the expanded input includes bus signals, plus signs appear next to the names of those bus signals. You can expand them as well. In this way, you can view all signals entering the block, including those entering via buses.

To find the source of any signal entering the block, select the signal in the **Signals in the bus** list and click the adjacent **Find** button. Simulink opens the subsystem containing the signal source, if necessary, and highlights the source's icon.

Data Type Support

The Bus Creator block accepts and outputs real or complex values of any data type supported by Simulink, including fixed-point and enumerated data types, as well as bus objects.

For a discussion on the data types supported by Simulink, refer to “Data Types Supported by Simulink”.


Filter by name
Find
Refresh
>>
OK
Cancel
Help
Apply

Function Block Parameters: Bus Creator

BusCreator

This block creates a bus signal from its inputs.

Parameters

Inherit bus signal names from input ports

Number of inputs: 2

Signals in the bus
- signal1
- signal2

Rename selected signal:

Output data type: Inherit: auto

Parameters and Dialog Box

- “Signal naming options” on page 1-70
- “Number of inputs” on page 1-71

- “Signals in bus” on page 1-72
- “Enable regular expression” on page 1-86
- “Show filtered results as a flat list” on page 1-87
- “Rename selected signal” on page 1-76
- “Data type” on page 1-77
- “Show data type assistant” on page 1-1672
- “Mode” on page 1-79
- “Output as nonvirtual bus” on page 1-80

Signal naming options

Assign input signal names to the corresponding bus signals.

Settings

Default: `Inherit bus signal names from input ports`

`Inherit bus signal names from input ports`
Assign input signal names to the corresponding bus signals.

`Require input signal names to match signals below`
Inputs must have the names listed in the **Signals in bus** list.

Dependencies

Selecting `Require input signal names to match signals below` enables **Rename selected signal**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Number of inputs

Specify the number of input ports on this block.

Settings

Default: `2`

To bundle a group of signals, enter the number of signals in the group.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Signals in bus

Show the signals in the output bus.

Settings

When you modify the **Number of inputs** parameter, click **Refresh** to update the list of signals.

Tips

- A plus sign (+) next to a signal name indicates that the signal is itself a bus. Click the plus sign to display the subsidiary bus signals.
- Click the **Refresh** button to update the list after editing the name of an input signal.
- Click the **Find** button to highlight the source of the currently selected signal.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Enable regular expression

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”.

Settings

Default: On

☑ On

Allow use of MATLAB regular expressions for filtering signal names.

☐ Off

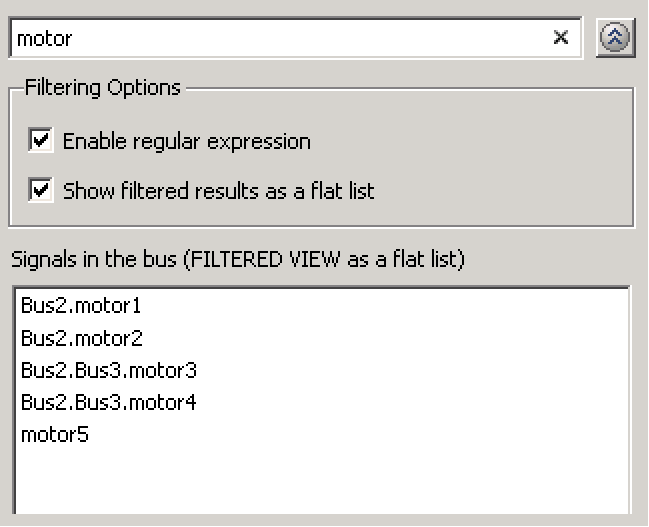
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box (⌄) enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



motor

Filtering Options

☑ Enable regular expression

☑ Show filtered results as a flat list

Signals in the bus (FILTERED VIEW as a flat list)

Bus2.motor1
Bus2.motor2
Bus2.Bus3.motor3
Bus2.Bus3.motor4
motor5

Settings

Default: Off

☑ On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

☐ Off

Display filtered bus hierarchies using a tree format.

Dependencies

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box (⏬) enables this parameter.

Rename selected signal

List the name of the signal currently selected in the **Signals in the bus** list when you select the `Require input signal names to match signals below` option.

Settings

Default: `''`

Edit this field to change the name of the currently selected signal. See "Signal Names" for guidelines for signal names.

Dependencies

Selecting `Require input signal names to match signals below` for **Parameters** and `signal1` or `signal2` for **Signals in the bus** enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Data type

Specify the output data type of the external input.

Settings

Default: `Inherit: auto`

`Inherit: auto`
: A rule that inherits a data type

`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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

Settings

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

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`
: Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right.

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

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

  Do not specify a bus object as the expression.

Tips

At the beginning of a simulation or when you update the model diagram, Simulink checks whether the signals connected to this Bus Creator block have the specified structure. If not, Simulink displays an error message.

Dependency

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant".

Output as nonvirtual bus

Output a nonvirtual bus.

Settings

Default: Off

☑ On

Output a nonvirtual bus.

☐ Off

Output a virtual bus.

Tips

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

Dependencies

The following **Data type** values enable this parameter:

- `Bus: <object name>`
- `<data type expression>` that specifies a bus object

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Examples

For an example of how the Bus Creator block works, see the `sldemo_househeat` model.

Characteristics

Multidimensionalized	Yes
Virtual	Yes, if the output bus is virtual For more information, see "Virtual Blocks" in the Simulink documentation.

See Also

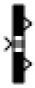
- "Composite Signals"
- "Create and Access a Bus"
- Bus Assignment
- Bus Selector
- Bus to Vector
- `Simulink.Bus`
- `Simulink.Bus.cellToObject`
- `Simulink.Bus.createObject`
- `Simulink.BusElement`
- `Simulink.Bus.objectToCell`
- `Simulink.Bus.save`

Bus Selector

Purpose Select signals from incoming bus

Library 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. For information about buses, see:



- "Composite Signals"
- "Create and Access a Bus"

When the block outputs separate elements, it outputs each element from a separate port from top to bottom of the block. See "How to Rotate a Block" for a description of the port order for various block orientations.

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

Caution

MathWorks recommends not using Bus Selector blocks in library blocks, because such use complicates changing the library blocks and increases the likelihood of errors. See "Buses and Libraries" for more information.

The following limitations apply to working with arrays of buses, when using the Bus Selector block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

- You cannot connect an array of buses signal to a Bus Selector block. To work with an array of buses signal, first 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.

- You cannot assign into a sub-bus that is an array of buses.

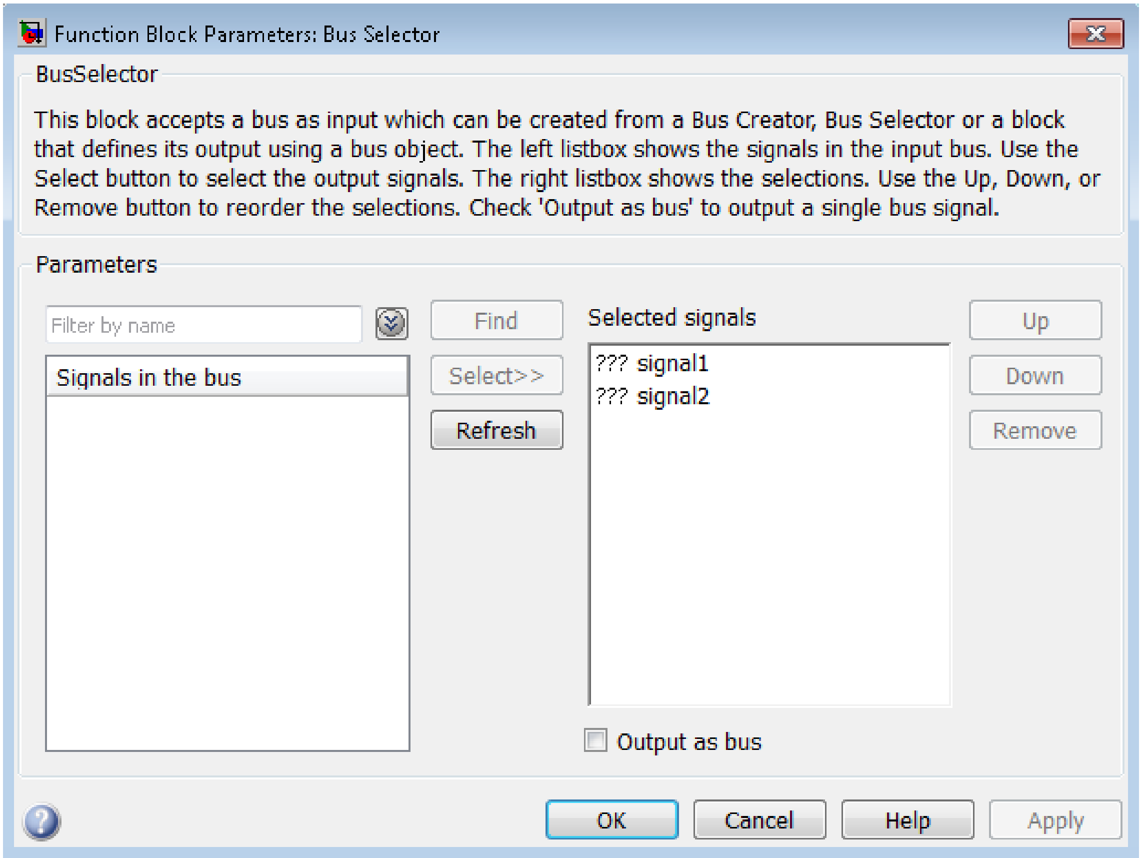
Data Type Support

A Bus Selector block accepts and outputs real or complex values of any data type supported by Simulink software, including fixed-point and enumerated data types.

For a discussion on the data types supported by Simulink software, see “Data Types Supported by Simulink” in the “Working with Data” chapter of the Simulink documentation.

Parameters and Dialog Box

The Bus Selector dialog box appears as follows:



Function Block Parameters: Bus Selector
BusSelector

This block accepts a bus as input which can be created from a Bus Creator, Bus Selector or a block that defines its output using a bus object. The left listbox shows the signals in the input bus. Use the Select button to select the output signals. The right listbox shows the selections. Use the Up, Down, or Remove button to reorder the selections. Check 'Output as bus' to output a single bus signal.

Parameters
Filter by name
Find
Select>>
Refresh
Signals in the bus
Selected signals
Up
Down
Remove
OK
Cancel
Help
Apply

Selected signals:

??? signal1
??? signal2

☐ Output as bus

Signals in the bus

Shows the signals in the input bus.

Settings

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

Tips

- Use **Select>>** to select signals to output.
- To find the source of any signal entering the block, select the signal in the list and click **Find**. The Simulink software opens the subsystem containing the signal source, and highlights the source's icon.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Enable regular expression

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”.

Settings

Default: On

☑ On

Allow use of MATLAB regular expressions for filtering signal names.

☐ Off

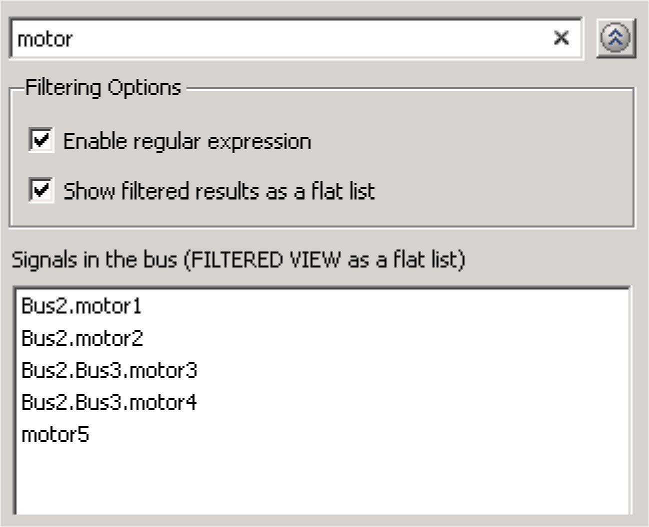
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box (⊙) enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



motor

Filtering Options

☑ Enable regular expression

☑ Show filtered results as a flat list

Signals in the bus (FILTERED VIEW as a flat list)

```
Bus2.motor1
Bus2.motor2
Bus2.Bus3.motor3
Bus2.Bus3.motor4
motor5
```

Settings

Default: Off

☑ On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

☐ Off

Display filtered bus hierarchies using a tree format.

Dependencies

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box (⏬) enables this parameter.

Selected signals

Shows the signals to be output.

Settings

Default: `signal1,signal2`

You can change the list by using the **Up**, **Down**, and **Remove** buttons.

Tips

- Port connectivity is maintained when the signal order is changed.
- If an output signal listed in the **Selected signals** list box is not an input to the Bus Selector block, the signal name is preceded by three question marks (`???`).

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output as bus

Output the selected elements as a bus.

Settings

Default: Off

☑ On

Output the selected elements as a bus.

☐ Off

Output the selected elements as standalone signals, each from an output port that is labeled with the corresponding element's name.

Tips

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 **Data type** parameter to use a `Simulink.Bus` bus object. For an example, see the Signal Conversion documentation.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Examples

For an example of how the Bus Selector block works, see the `sldemo_fuelsys` model. The Bus Selector block appears in the following subsystems:

- `fuel_rate_control/airflow_calc`
- `fuel_rate_control/validate_sample_time`

Characteristics

Multidimensionalized	Yes
Virtual	Yes, if the input bus is virtual For more information, see "Virtual Blocks" in the Simulink documentation.

See Also

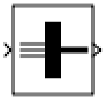
- “Composite Signals”
- “Create and Access a Bus”
- Bus Assignment
- Bus Creator
- Bus to Vector
- Signal Conversion
- `Simulink.Bus`
- `Simulink.Bus.cellToObject`
- `Simulink.Bus.createObject`
- `Simulink.BusElement`
- `Simulink.Bus.objectToCell`
- `Simulink.Bus.save`

Bus to Vector

Purpose Convert virtual bus to vector

Library Signal Attributes

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

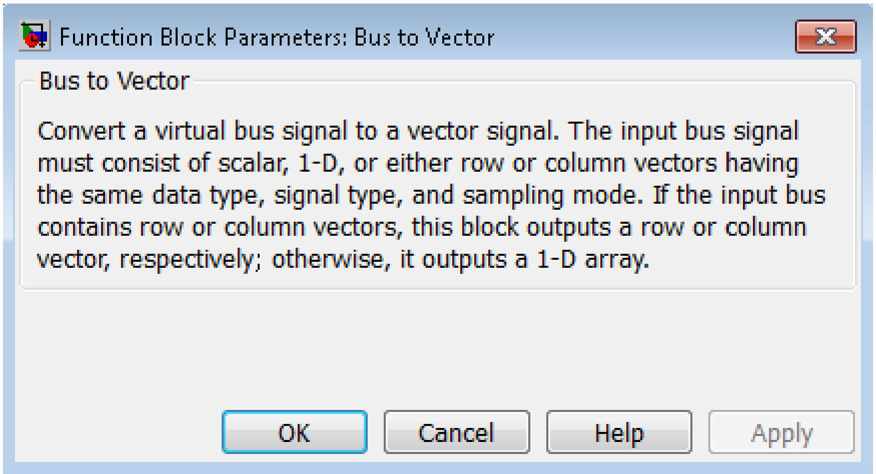


Use the Bus to Vector block only to replace an implicit bus-to-vector conversion with an equivalent explicit conversion. See “Bus signal treated as vector” and “Correcting Buses Used as Muxes” for more information.

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

Data Type Support The Bus to Vector block accepts and outputs real or complex values of any data type that Simulink supports, including fixed-point and enumerated data types.

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



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

Parameters and Dialog Box

This block has no user-accessible parameters.

Characteristics

Multidimensionalized	Yes

See Also

- "Composite Signals"
- "Create and Access a Bus"
- Avoiding Mux/Bus Mixtures
- Bus Assignment
- Bus Creator
- Bus Selector
- `Simulink.BlockDiagram.addBusToVector`
- `Simulink.Bus`
- `Simulink.Bus.cellToObject`
- `Simulink.Bus.createObject`

- `Simulink.BusElement`
- `Simulink.Bus.objectToCell`
- `Simulink.Bus.save`

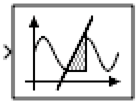
Purpose

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

Library

Model Verification

Description



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

The **Model Verification block enabling** setting under **Debugging** on the **Data Validity** diagnostics pane of the Configuration Parameters dialog box lets you enable or disable all model verification blocks, including Check Discrete Gradient blocks, in a model.

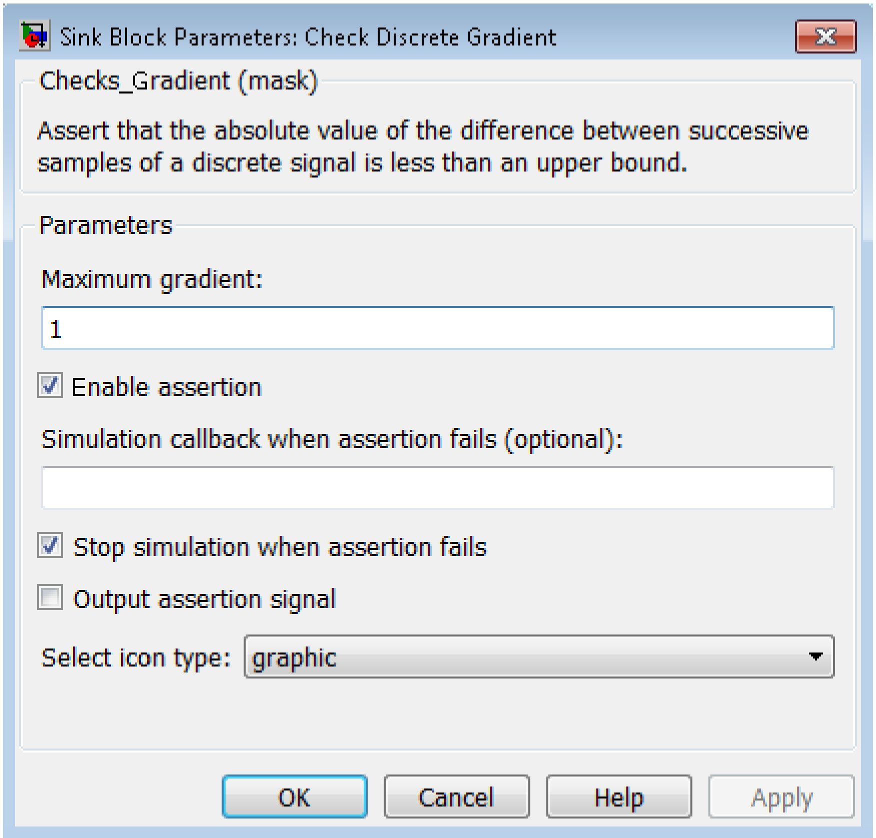
The Check Discrete Gradient block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied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

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

Data Type Support

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

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



Sink Block Parameters: Check Discrete Gradient

Checks_Gradient (mask)

Assert that the absolute value of the difference between successive samples of a discrete signal is less than an upper bound.

Parameters

Maximum gradient: 1

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Parameters and Dialog Box

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

Enable assertion
Clearing this check box disables the Check Discrete Gradient block, that is, causes the model to behave as if the block did

not exist. The **Model Verification block enabling** setting under **Debugging** on the **Data Validity** diagnostics pane of the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Discrete Gradient blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Discrete Gradient block to halt the simulation when the block's output is zero and display an error message in the Simulink Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

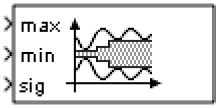
Purpose

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

Library

Model Verification

Description



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

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

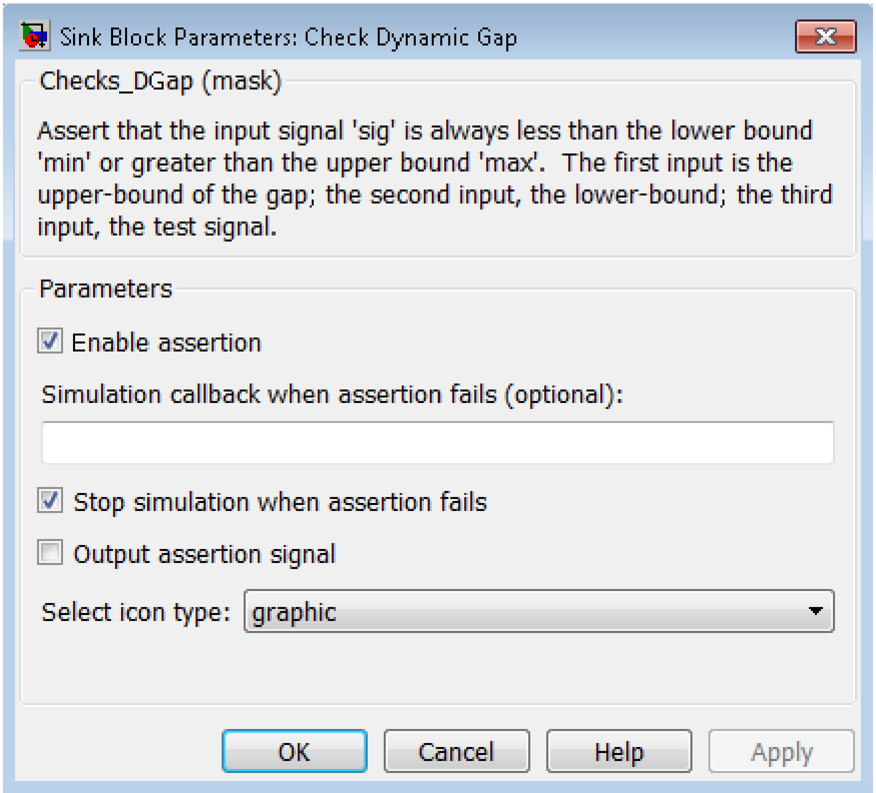
Note For information about how Simulink Coder generated code handles Model Verification blocks, see "Debug".

Data Type Support

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

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

Parameters and Dialog Box


Sink Block Parameters: Check Dynamic Gap

Checks_DGap (mask)

Assert that the input signal 'sig' is always less than the lower bound 'min' or greater than the upper bound 'max'. The first input is the upper-bound of the gap; the second input, the lower-bound; the third input, the test signal.

Parameters

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Enable assertion

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Gap block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

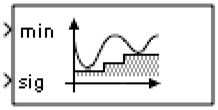
Purpose

Check that one signal is always less than another signal

Library

Model Verification

Description



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

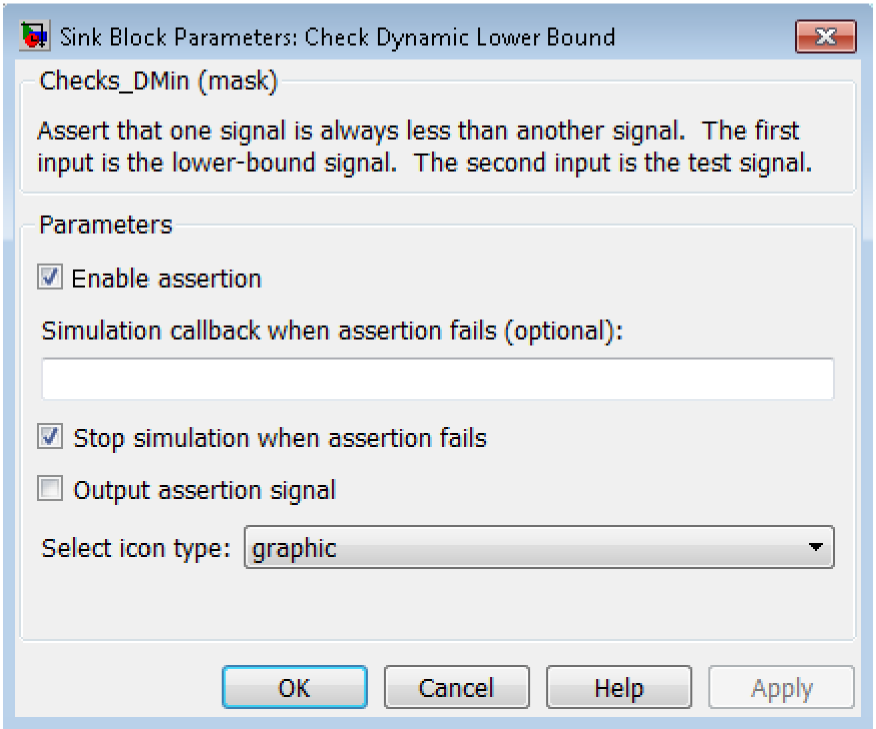
The Check Dynamic Lower Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied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

Note For information about how Simulink Coder generated code handles Model Verification blocks, see "Debug".

Data Type Support

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

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



Sink Block Parameters: Check Dynamic Lower Bound

Checks_DMin (mask)

Assert that one signal is always less than another signal. The first input is the lower-bound signal. The second input is the test signal.

Parameters

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Parameters and Dialog Box

Enable assertion

Clearing this check box disables the Check Dynamic Lower Bound block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting under **Debugging** on the **Data Validity** diagnostics pane of the Configuration Parameters dialog box allows you to enable or disable all model verification blocks, including Check Dynamic

Lower Bound blocks, in a model regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Lower Bound block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving block

Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Check Dynamic Range

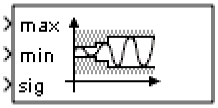
Purpose

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

Library

Model Verification

Description



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

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

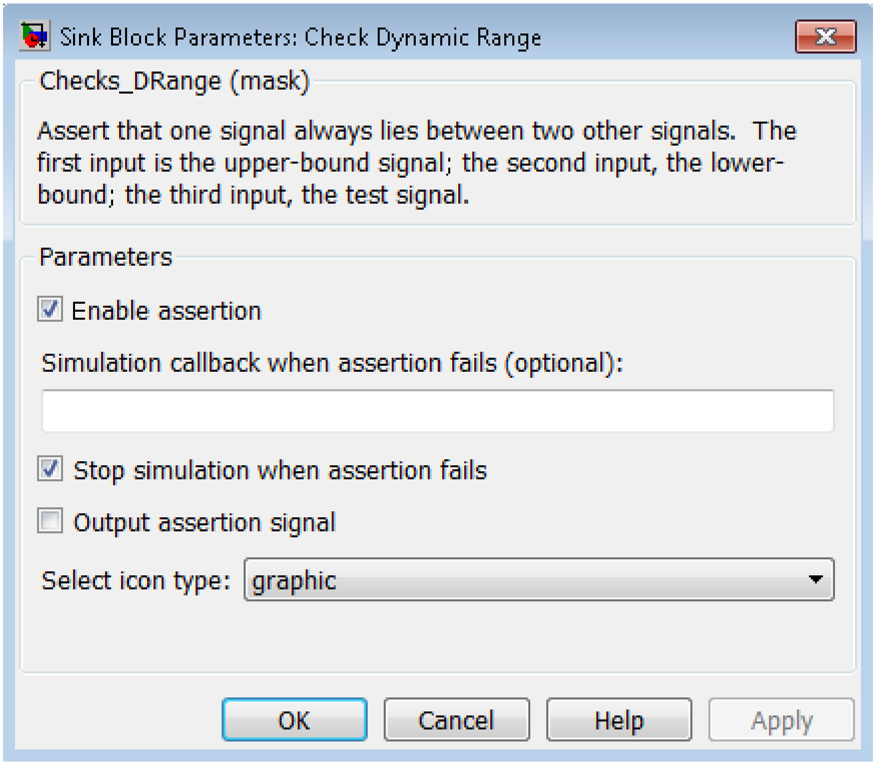
Note For information about how Simulink Coder generated code handles Model Verification blocks, see "Debug".

Data Type Support

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

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

Parameters and Dialog Box



Sink Block Parameters: Check Dynamic Range

Checks_DRange (mask)

Assert that one signal always lies between two other signals. The first input is the upper-bound signal; the second input, the lower-bound; the third input, the test signal.

Parameters

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Enable assertion

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Range block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

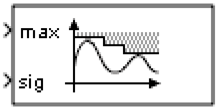
Purpose

Check that one signal is always greater than another signal

Library

Model Verification

Description



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

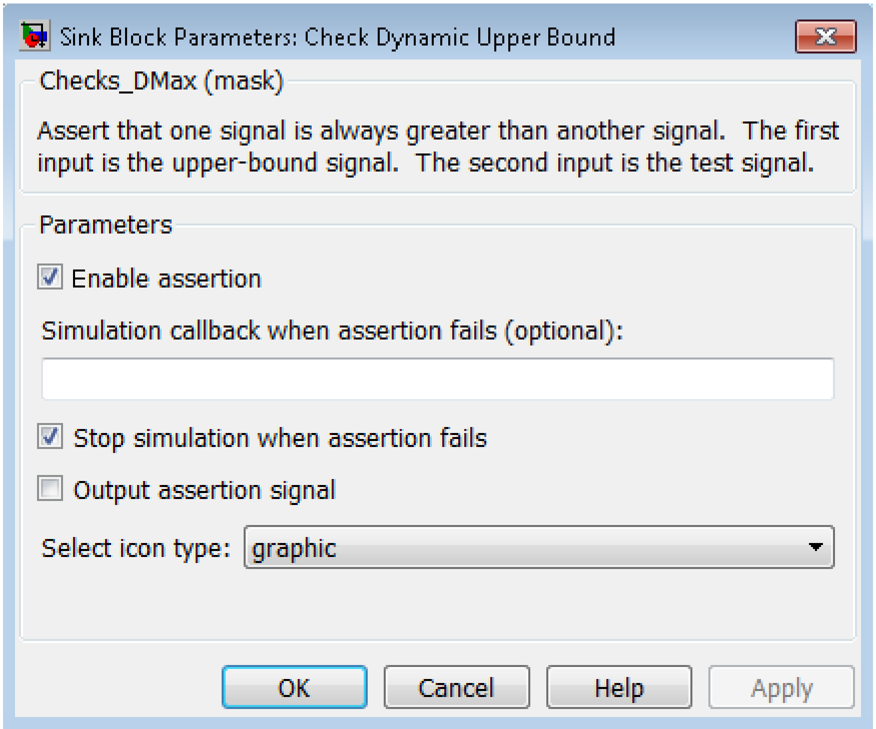
The Check Dynamic Upper Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied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

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

Data Type Support

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

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



Sink Block Parameters: Check Dynamic Upper Bound

Checks_DMax (mask)

Assert that one signal is always greater than another signal. The first input is the upper-bound signal. The second input is the test signal.

Parameters

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Parameters and Dialog Box

Enable assertion

Clearing this check box disables the Check Dynamic Upper Bound block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting under **Debugging** on the **Data Validity** diagnostics pane of the Configuration Parameters dialog box allows you to enable or disable all model verification blocks, including Check Dynamic

Upper Bound blocks, in a model regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Upper Bound block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving block

Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

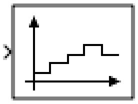
Purpose

Check that input signal has specified resolution

Library

Model Verification

Description



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

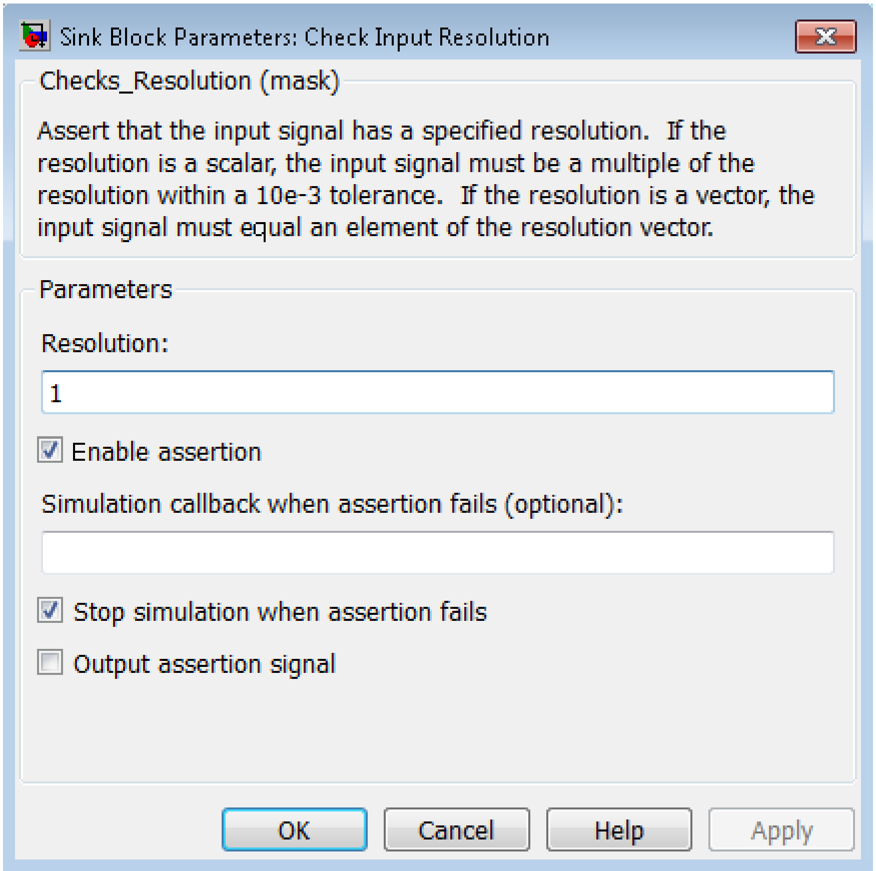
The Check Input Resolution block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that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

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

Data Type Support

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

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



Sink Block Parameters: Check Input Resolution

Checks_Resolution (mask)

Assert that the input signal has a specified 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.

Parameters

Resolution: 1

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Parameters and Dialog Box

Resolution

Specify the resolution that the input signal must have.

Enable assertion

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Input Resolution block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

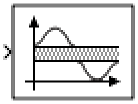
Purpose

Check that gap exists in signal's range of amplitudes

Library

Model Verification

Description



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

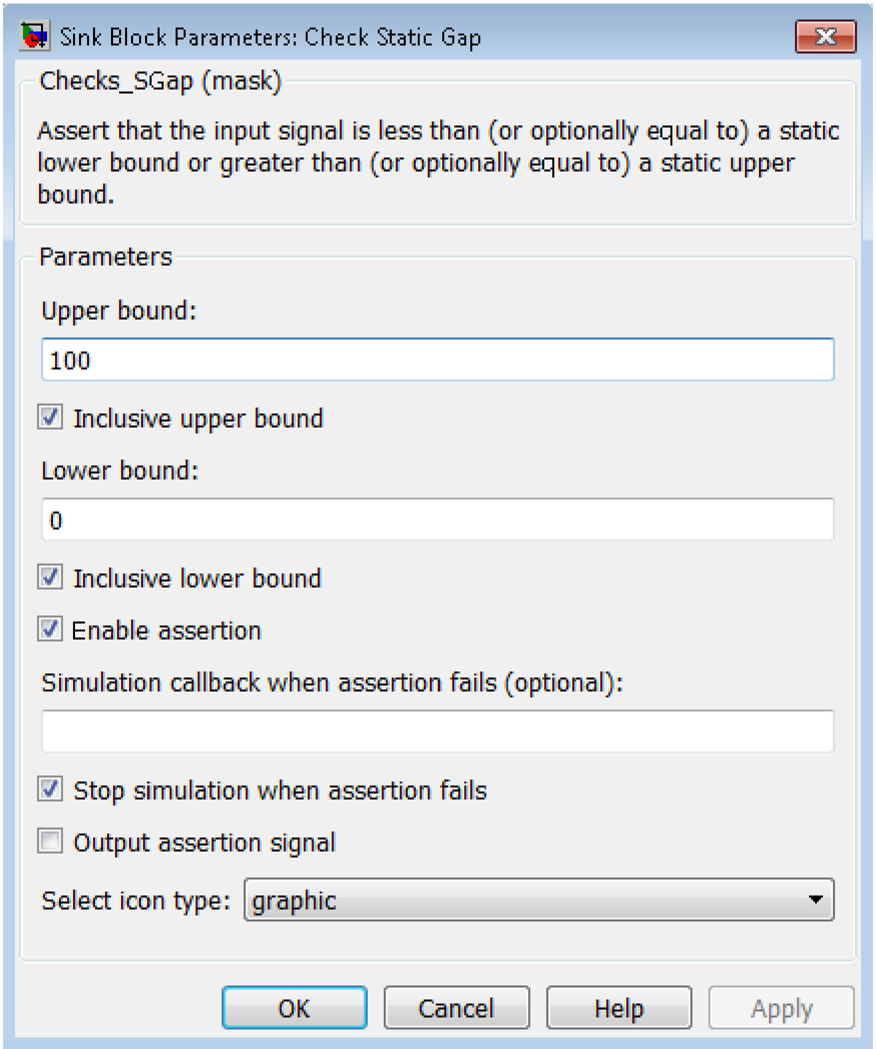
The Check Static Gap block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a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

Note For information about how Simulink Coder generated code handles Model Verification blocks, see "Debug".

Data Type Support

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

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



Sink Block Parameters: Check Static Gap

Checks_SGap (mask)

Assert that 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.

Parameters

Upper bound: 100

☑ Inclusive upper bound

Lower bound: 0

☑ Inclusive lower bound

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Parameters and Dialog Box

Upper bound

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

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

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

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

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

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

Stop simulation when assertion fails
Selecting this check box causes the Check Static Gap block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

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

Optimization pane of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Check Static Lower Bound

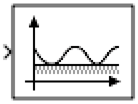
Purpose

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

Library

Model Verification

Description



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

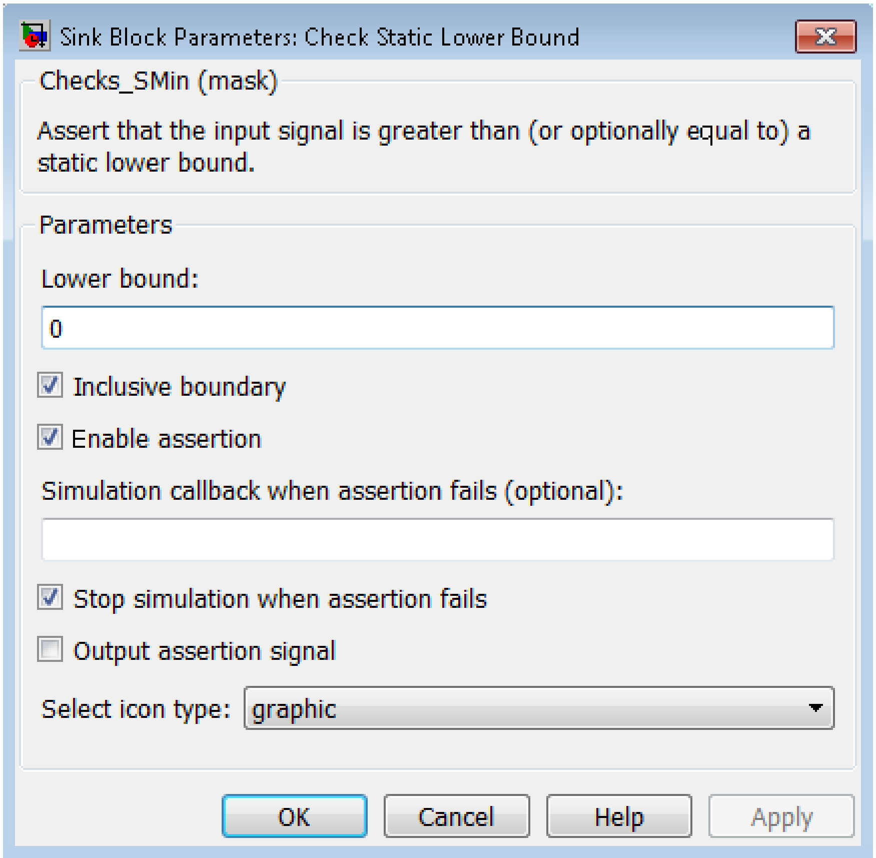
The Check Static Lower Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied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

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

Data Type Support

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

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



Sink Block Parameters: Check Static Lower Bound

Checks_SMin (mask)

Assert that the input signal is greater than (or optionally equal to) a static lower bound.

Parameters

Lower bound:

0

☑ Inclusive boundary

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Parameters and Dialog Box

Lower bound

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

Inclusive boundary

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

Enable assertion

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Static Lower Bound block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Check Static Range

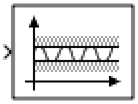
Purpose

Check that signal falls inside fixed range of amplitudes

Library

Model Verification

Description



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

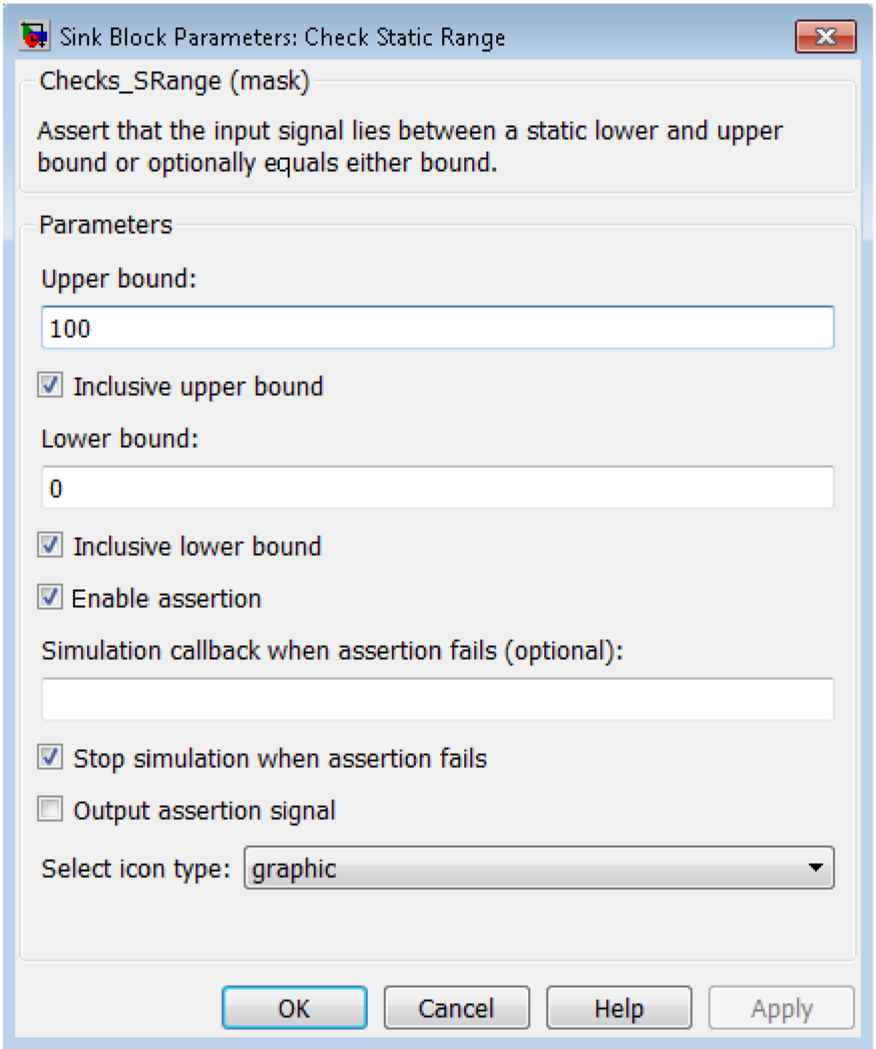
The Check Static Range block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that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

Note For information about how Simulink Coder generated code handles Model Verification blocks, see "Debug".

Data Type Support

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

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



Sink Block Parameters: Check Static Range

Checks_SRange (mask)

Assert that the input signal lies between a static lower and upper bound or optionally equals either bound.

Parameters

Upper bound: 100

☑ Inclusive upper bound

Lower bound: 0

☑ Inclusive lower bound

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Parameters and Dialog Box

Upper bound

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

Inclusive upper bound

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

Lower bound

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

Inclusive lower bound

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

Enable assertion

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Static Range block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Optimization pane of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

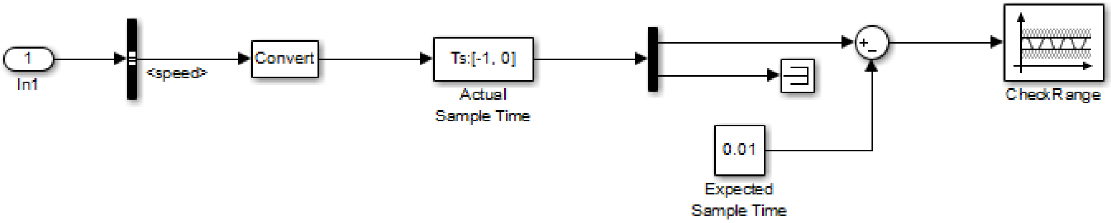
Select icon type

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

Examples

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

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



Characteristics

Direct Feedthrough	No
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	Yes

Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

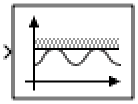
Check that signal is less than (or optionally equal to) static upper bound

Library

Model Verification

Description

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



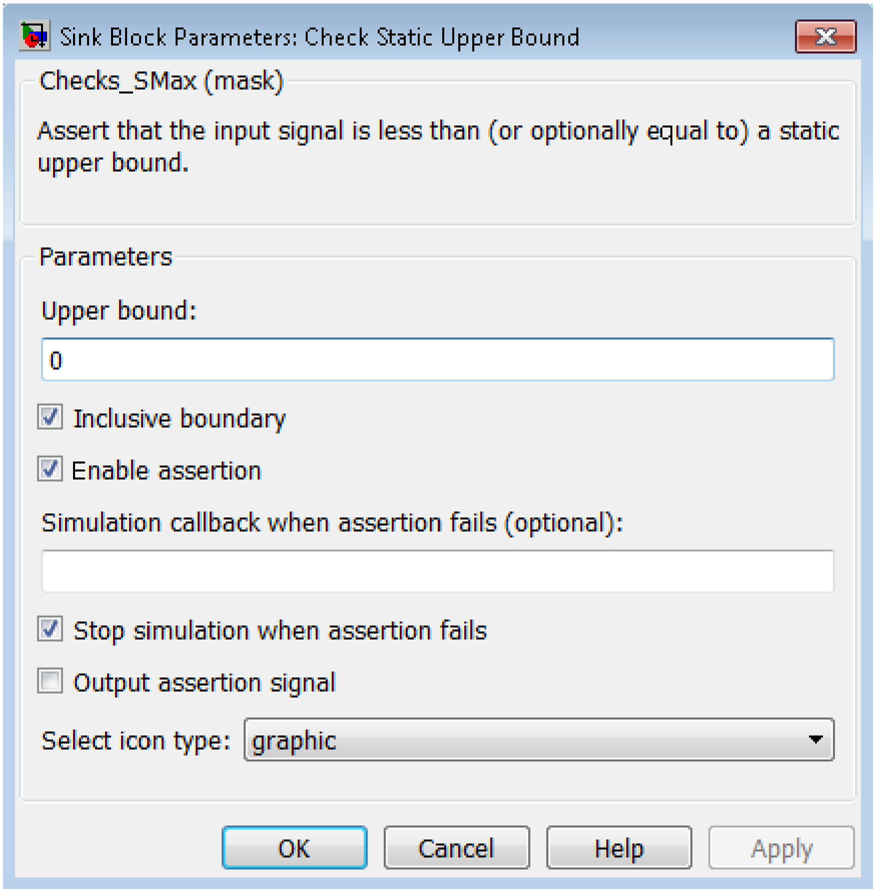
The Check Static Upper Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied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

Note For information about how Simulink Coder generated code handles Model Verification blocks, see "Debug".

Data Type Support

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

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



Sink Block Parameters: Check Static Upper Bound

Checks_SMax (mask)

Assert that the input signal is less than (or optionally equal to) a static upper bound.

Parameters

Upper bound: 0

☑ Inclusive boundary

☑ Enable assertion

Simulation callback when assertion fails (optional):

☑ Stop simulation when assertion fails

☐ Output assertion signal

Select icon type: graphic

Parameters and Dialog Box

Upper bound

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

Inclusive boundary

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

Enable assertion

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Static Upper Bound block to halt the simulation when the block's output is zero and display an error message in the Simulation Diagnostics Viewer. Otherwise, the block displays a warning message in the MATLAB Command Window and continues the simulation.

Output assertion signal

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

Select icon type

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

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

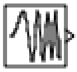
Purpose

Generate sine wave with increasing frequency

Library

Sources

Description



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

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

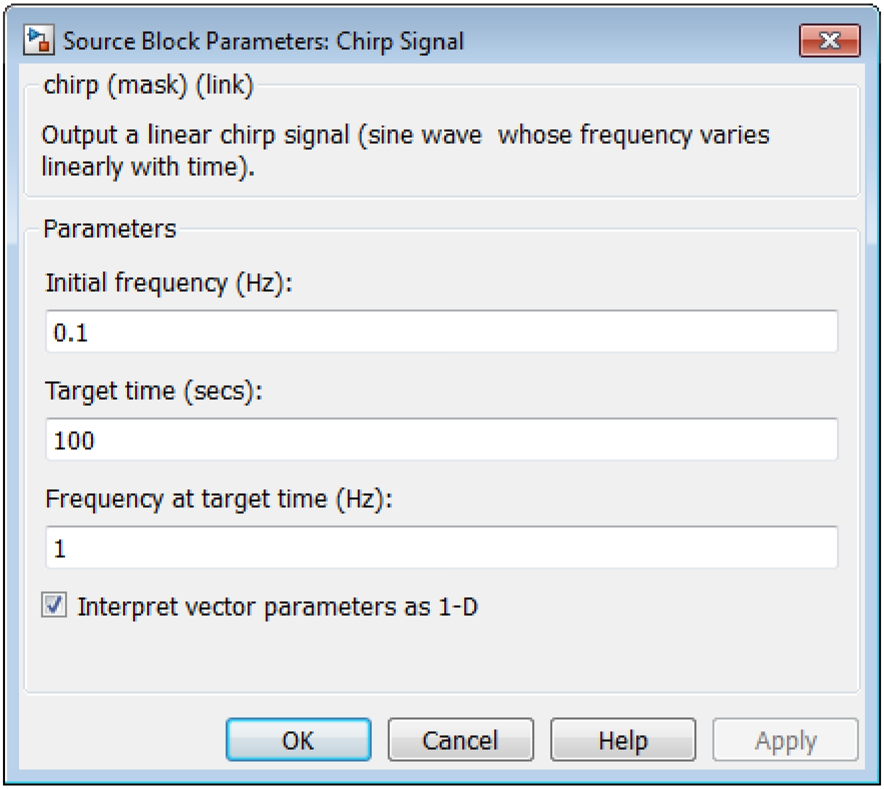
The following limitations apply to the Chirp Signal block:

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

Data Type Support

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

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



chirp (mask) (link)

Output a linear chirp signal (sine wave whose frequency varies linearly with time).

Parameters

Initial frequency (Hz): 0.1

Target time (secs): 100

Frequency at target time (Hz): 1

☑ Interpret vector parameters as 1-D

Parameters and Dialog Box

Initial frequency

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

Target time

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

Frequency at target time

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

Interpret vector parameters as 1-D

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

Characteristics

Sample Time	Continuous
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes
Zero-Crossing Detection	No

Clock

Purpose

Display and provide simulation time

Library

Sources

Description

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

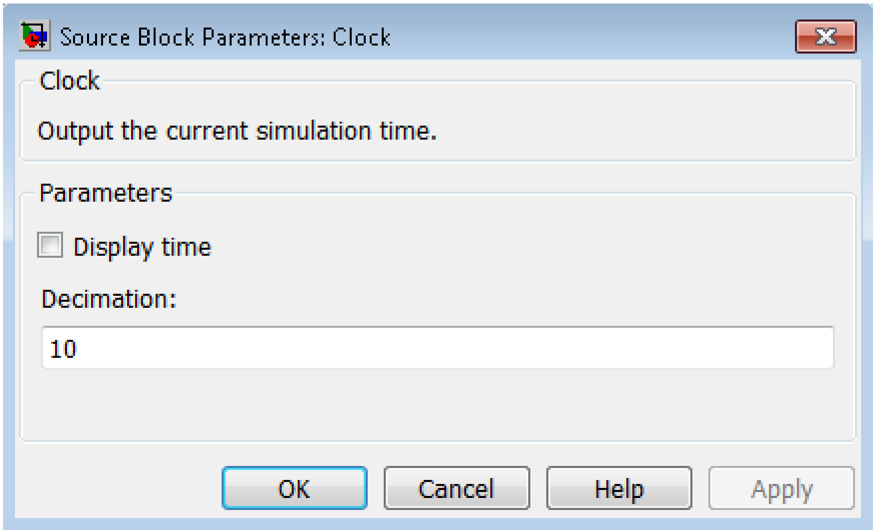
When you need the current time within a discrete system, use the Digital Clock block.

Data Type Support

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

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

Parameters and Dialog Box



Source Block Parameters: Clock

Clock

Output the current simulation time.

Parameters

☐ Display time

Decimation:

10

Display time

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

Decimation

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

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

Examples

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

- `sldemo_tonegen_fixpt`
- `penddemo`

Characteristics

Sample Time	Continuous
Scalar Expansion	N/A
Dimensionalized	No
Zero-Crossing Detection	No

See Also

Digital Clock

Combinatorial Logic

Purpose

Implement truth table

Library

Logic and Bit Operations

Description

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

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

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

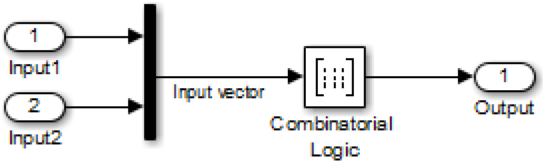
```
number of rows = 2 ^ (number of inputs)
```

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:

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

Two-Input AND Logic

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



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

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

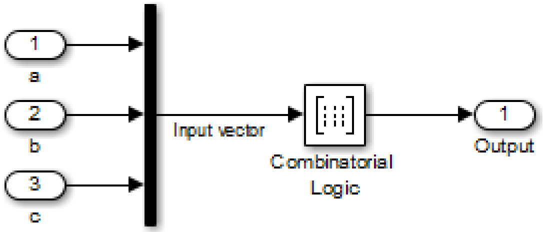
```
(2^1*1 + 1)
```

The output value is 0.

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

Circuit Logic

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



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

Inputs			Outputs	
a	**b**	**c**	**c’**	**s**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

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

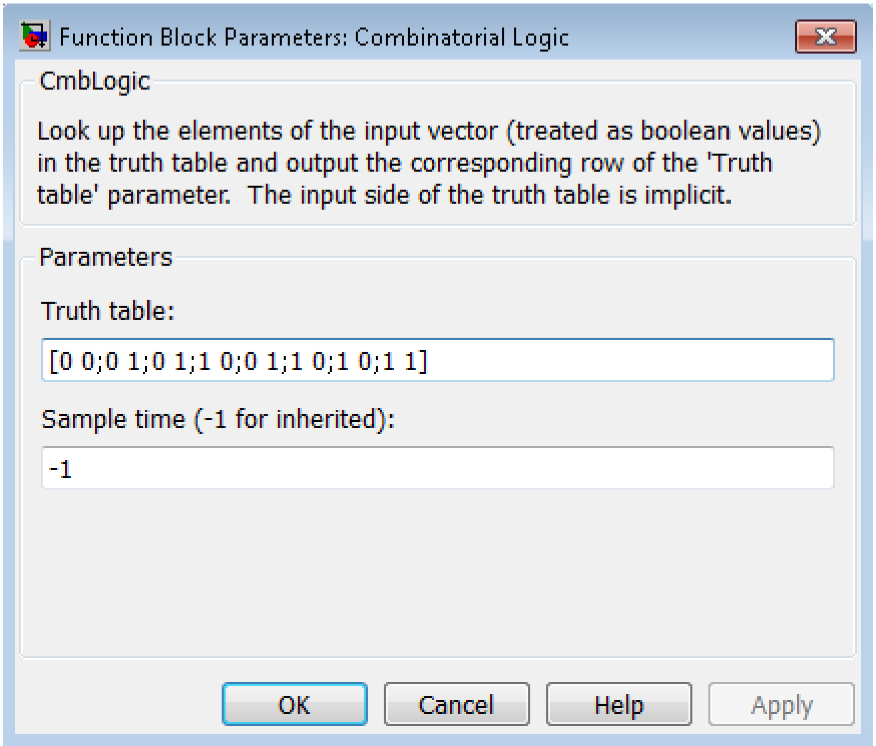
Data Type Support

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

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

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

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


Function Block Parameters: Combinatorial Logic
CmbLogic
Parameters
OK
Cancel
Help
Apply

Look up the elements of the input vector (treated as boolean values) in the truth table and output the corresponding row of the 'Truth table' parameter. The input side of the truth table is implicit.

Truth table:

[0 0;0 1;0 1;1 0;0 1;1 0;1 0;1 1]

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Truth table

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

Sample time (-1 for inherited)

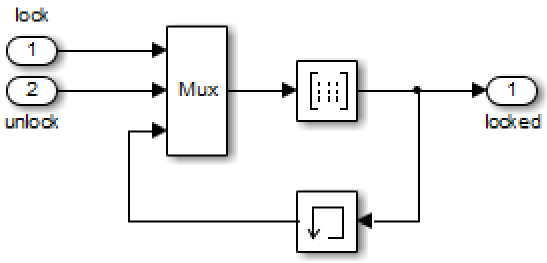
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the Simulink documentation.

Examples

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

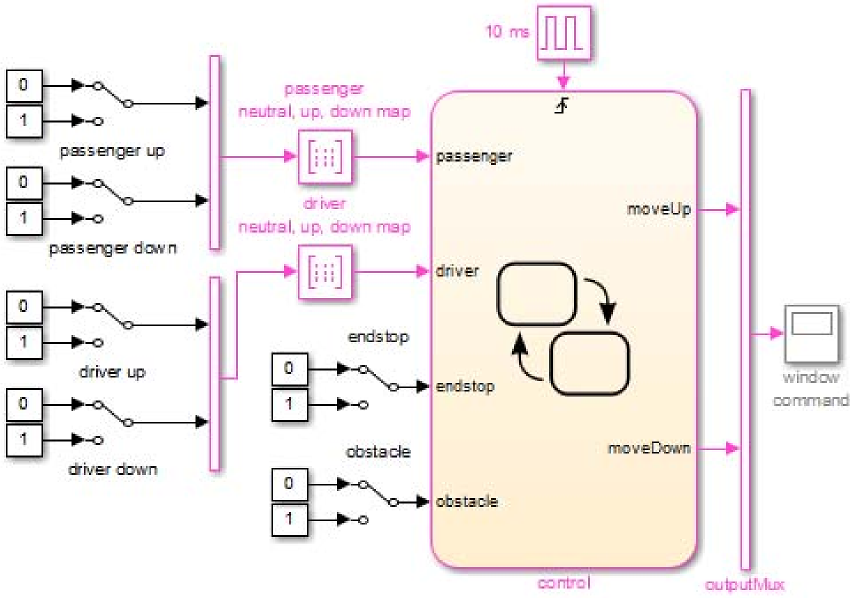
The `sldemo_clutch` model shows how you can use the Combinatorial Logic block with the Memory block to implement a finite-state machine.

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



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

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


10 ms
0
1
passenger up
0
1
passenger down
0
1
driver up
0
1
driver down
passenger
neutral, up, down map
driver
neutral, up, down map
endstop
0
1
obstacle
0
1
passenger
driver
endstop
obstacle
moveUp
moveDown
control
outputMux
window
command

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	Yes, the output width is the number of columns of the **Truth table** parameter
Zero-Crossing Detection	No

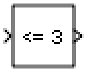
Purpose

Determine how signal compares to specified constant

Library

Logic and Bit Operations

Description



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

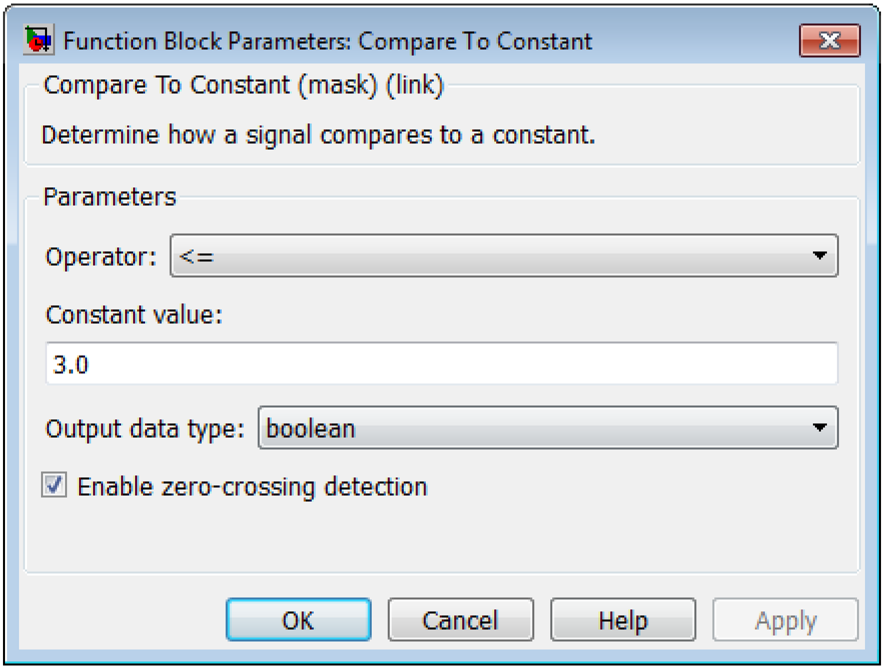
- == — Determine whether the input is equal to the specified constant.
- ~= — Determine whether the input is not equal to the specified constant.
- < — Determine whether the input is less than the specified constant.
- <= — Determine whether the input is less than or equal to the specified constant.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

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

Data Type Support

The Compare To Constant block accepts inputs of any data type that Simulink supports, including fixed-point and enumerated data types. The block first converts its **Constant value** parameter to the input data type, and then performs the specified operation. The block output is `uint8` or `boolean` as specified by the **Output data type** parameter.

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


Function Block Parameters: Compare To Constant
Compare To Constant (mask) (link)

Determine how a signal compares to a constant.

Parameters

Operator: <=

Constant value:

3.0

Output data type: boolean

☑ Enable zero-crossing detection

OK
Cancel
Help
Apply

Parameters and Dialog Box

Operator

Specify how the input is compared to the constant value, as discussed in Description.

Constant value

Specify the constant value to which the input is compared.

Output data type

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

Enable zero-crossing detection

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

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled.

See Also

Compare To Zero

Compare To Zero

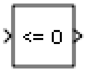
Purpose

Determine how signal compares to zero

Library

Logic and Bit Operations

Description



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

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

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

Data Type Support

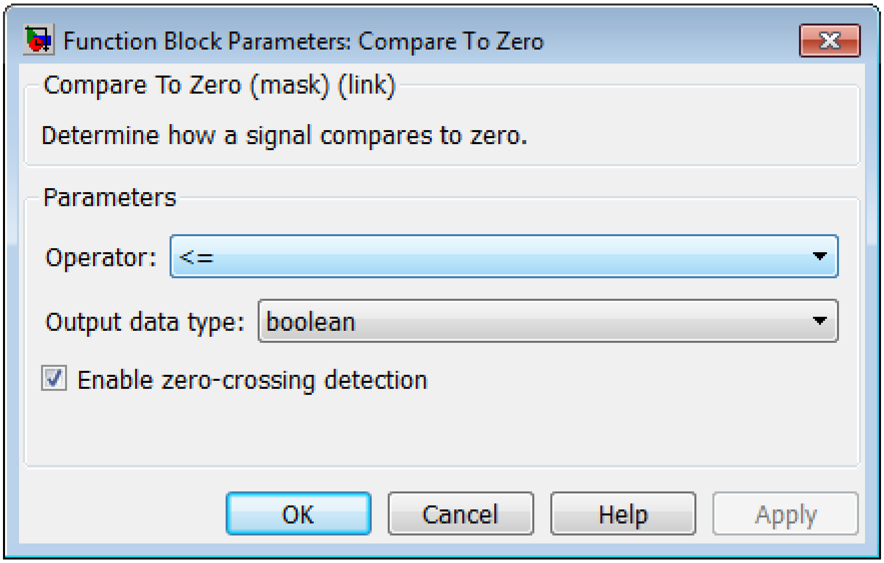
The Compare To Zero block accepts inputs of the following data types:

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

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

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

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


Function Block Parameters: Compare To Zero

Compare To Zero (mask) (link)

Determine how a signal compares to zero.

Parameters

Operator: <=

Output data type: boolean

☑ Enable zero-crossing detection

Parameters and Dialog Box

Operator
Specify how the input is compared to zero, as discussed in Description.

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

Enable zero-crossing detection

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

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled.

See Also

Compare To Constant

Complex to Magnitude-Angle

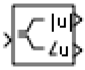
Purpose

Compute magnitude and/or phase angle of complex signal

Library

Math Operations

Description

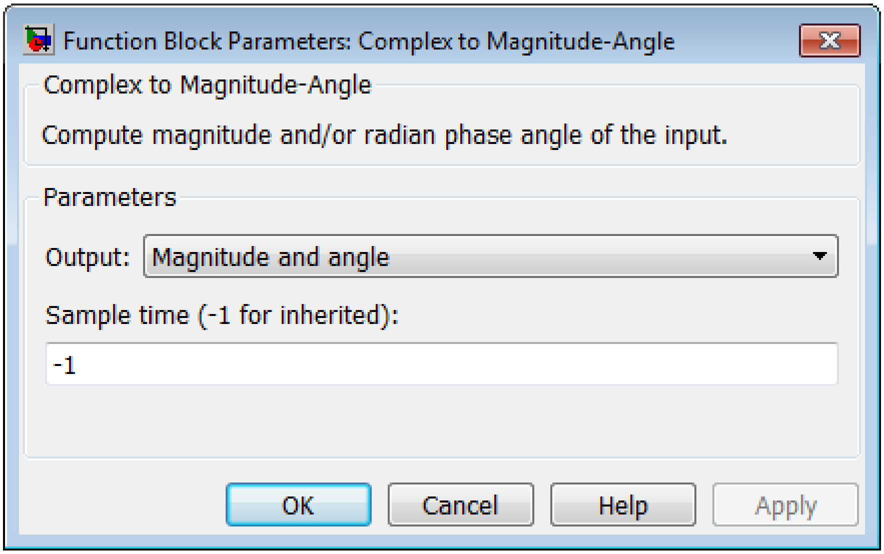


The Complex to Magnitude-Angle block accepts a complex-valued signal of type `double` or `single`. It outputs the magnitude and/or phase angle of the input signal, depending on the setting of the **Output** parameter. The outputs are real values of the same data type as the block input. The input can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

Data Type Support

See the preceding description.

Parameters and Dialog Box



Function Block Parameters: Complex to Magnitude-Angle

Complex to Magnitude-Angle

Compute magnitude and/or radian phase angle of the input.

Parameters

Output: Magnitude and angle

Sample time (-1 for inherited):

-1

Output

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

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see "Specify Sample Time" in the Simulink documentation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified by the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes

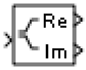
Multidimensionalized	Yes
Zero-Crossing Detection	No

Complex to Real-Imag

Purpose Output real and imaginary parts of complex input signal

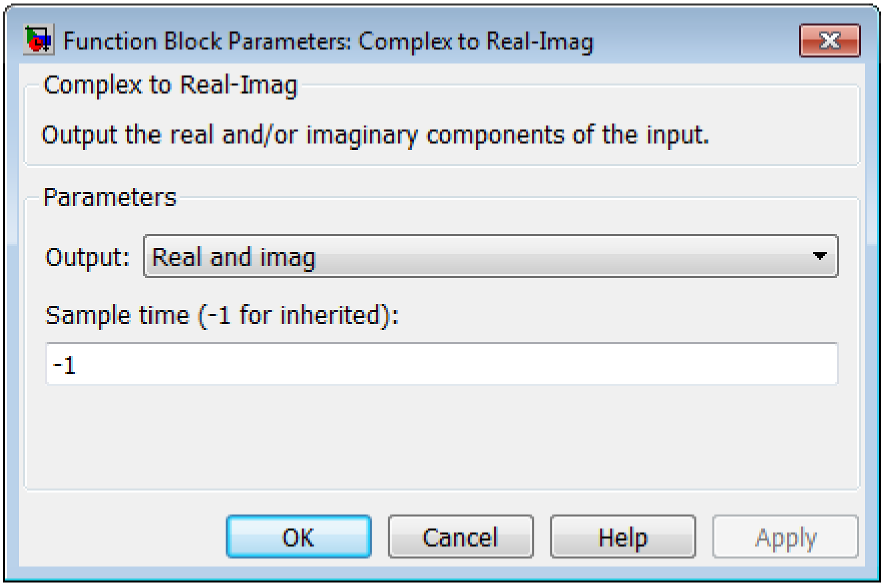
Library Math Operations

Description



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

Data Type Support See the preceding description. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Function Block Parameters: Complex to Real-Imag

Complex to Real-Imag

Output the real and/or imaginary components of the input.

Parameters

Output: Real and imag

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Output

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

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see "Specify Sample Time" in the Simulink documentation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified by the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes

Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

Represent any block selected from user-specified library of blocks

Library

Ports & Subsystems

Description

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

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

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

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

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

Creating a Master Configurable Subsystem

To create a master configurable subsystem:

1. Create a library of blocks representing the various configurations of the configurable subsystem.

2. Save the library.

3. Create an instance of the Configurable Subsystem block in the library.

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

4. Display the Configurable Subsystem block dialog by double-clicking it. The dialog displays a list of the other blocks in the library.

5. Under **List of block choices** in the dialog box, select the blocks that represent the various configurations of the configurable subsystems you are creating.

6. Click the **OK** button to apply the changes and close the dialog box.

7. Select **Block Choice** from the Configurable Subsystem block's context menu.

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

8. Select the block that you want the subsystem to represent by default.

9. Save the library.

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

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

- Change the default block choice to another block in the subsystem, then change the default block choice back to the original block.
- Recreate the configurable subsystem block, including the selection of the updated block as the default block choice.

Creating an Instance of a Configurable Subsystem

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

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

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

Setting Instance Block Parameters

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

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

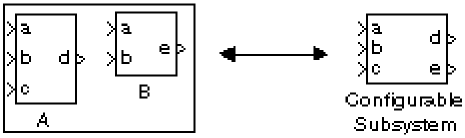
Mapping I/O Ports

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

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

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

For example, suppose that a library contains two blocks A and B and that block A has input ports labeled `a`, `b`, and `c` and an output port labeled `d` and that block B has input ports labeled `a` and `b` and an output port labeled `e`. A Configurable Subsystem block based on this library would have three input ports labeled `a`, `b`, and `c`, respectively, and two output ports labeled `d` and `e`, respectively, as illustrated in the following figure.



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

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

Convert to Variable Subsystem

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

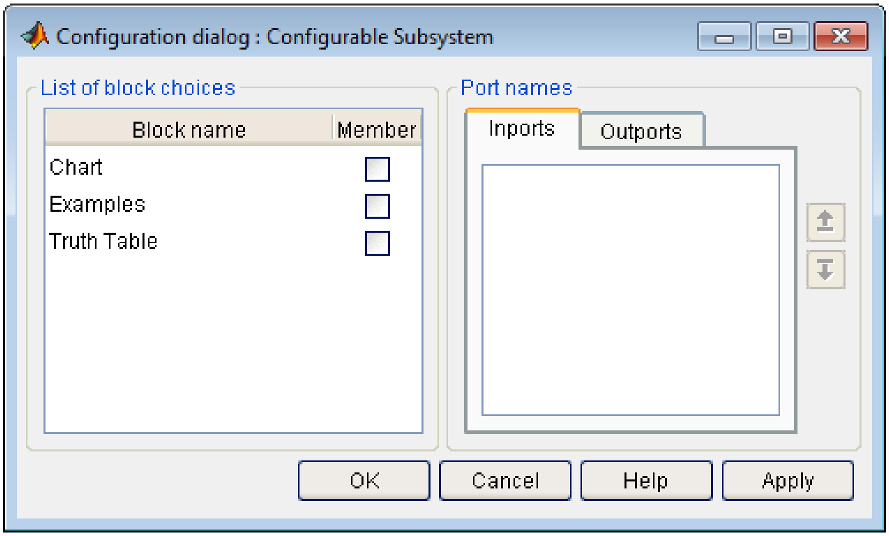
Simulink copies the block choices of the configurable subsystem to a new variant subsystem and adds the appropriate number of inports and outports to the variant subsystem. The current block choice of the configurable subsystem is made the active variant selection.

See Variant Subsystem for more information on variant choices.

Data Type Support

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

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


Configuration dialog : Configurable Subsystem
List of block choices
Block name
Member
Chart ☐
Examples ☐
Truth Table ☐
Port names
Inports
Outports
OK
Cancel
Help
Apply

Parameters and Dialog Box

List of block choices

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

Port names

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

Characteristics

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

Purpose

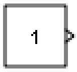
Generate constant value

Library

Sources

Description

The Constant block generates a real or complex constant value.



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

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

Also, the block can generate either a sample-based or frame-based signal, depending on the setting of the **Sampling mode**.

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

Data Type Support

By default, the Constant block outputs a signal whose data type and complexity are the same as those of the **Constant value** parameter. However, you can specify the output to be any data type that Simulink supports, including fixed-point and enumerated data types. The Enumerated Constant block can be more convenient than the Constant block for outputting a constant enumerated value. You can also use a bus object as the output data type, which can help to simplify a model (see “Bus Support” on page 1-187 for details).

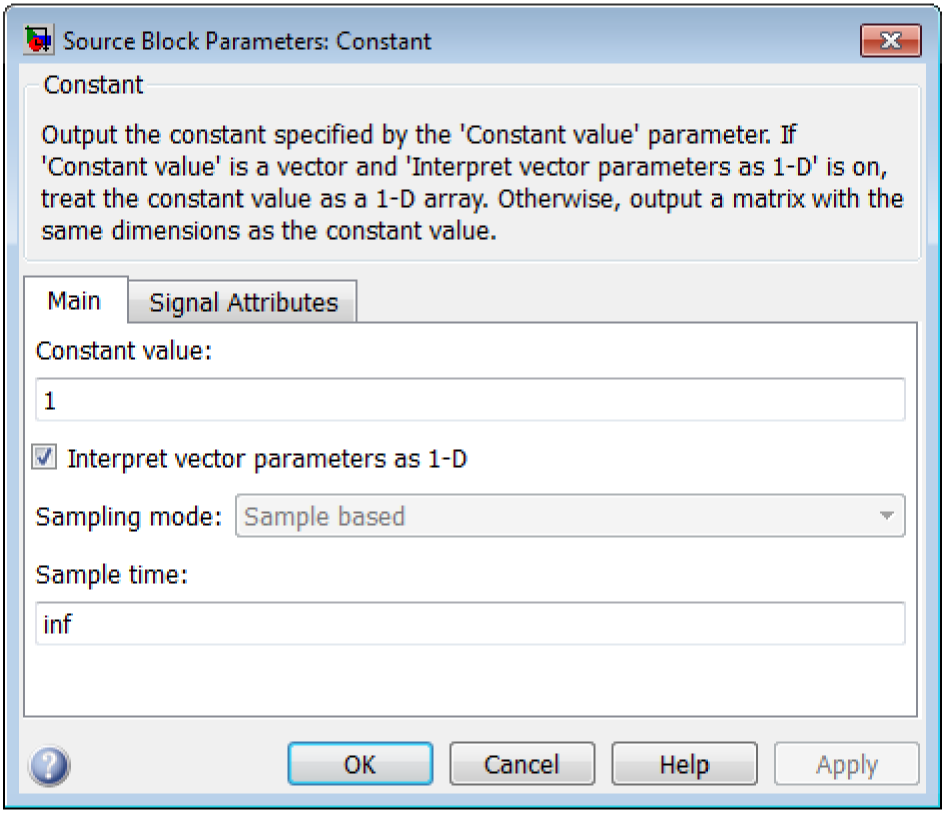
Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite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

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

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

Parameters and Dialog Box

The **Main** pane of the Constant block dialog box appears as follows:



Source Block Parameters: Constant

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Main
Signal Attributes

Constant value: 1

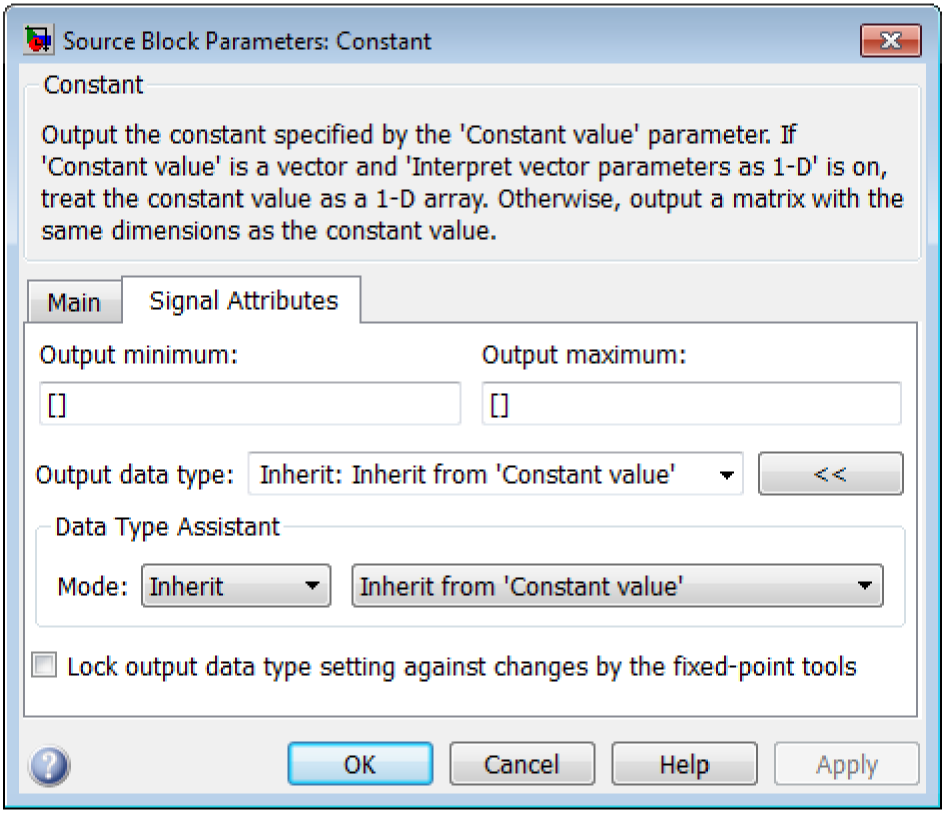
☑ Interpret vector parameters as 1-D

Sampling mode: Sample based

Sample time: inf

OK
Cancel
Help
Apply

The **Signal Attributes** pane of the Constant block dialog appears as follows:


Source Block Parameters: Constant

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Main
Signal Attributes

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit from 'Constant value'

<<

Data Type Assistant

Mode: Inherit — Inherit from 'Constant value'

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

OK
Cancel
Help
Apply

Constant value

Specify the constant value output of the block.

Settings

Default: 1

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

- You can enter any expression that MATLAB evaluates as a matrix, including the Boolean keywords `true` and `false`.
- If you set the **Output data type** to be a bus object, you can specify either:
  - A full MATLAB structure corresponding to the bus object
  - `0` to indicate a structure corresponding to the ground value of the bus object

  For details, see “Bus Support” on page 1-187.
- For non-bus data types, Simulink converts this parameter from its value data type to the specified output data type offline, using round toward nearest and saturation.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Interpret vector parameters as 1-D

Select this check box to output a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector.

Settings

Default: On

☑ On

Outputs a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector. For example, the block outputs a matrix of dimension `1-by-N` or `N-by-1`.

☐ Off

Does not output a vector of length `N` if the **Constant value** parameter evaluates to an `N`-element row or column vector.

If you clear this check box, you can interact with the **Sampling mode** parameter.

Dependencies

This parameter enables **Sampling mode**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sampling mode

Specify whether the output signal is `Sample based` or `Frame based`.

Settings

Default: `Sample based`

`Sample based`
    The output signal is sample-based.

`Frame based`
    The output signal is frame-based.

Tip

If you specify the **Data type** to be a bus object, do not specify a frame-based sampling mode.

Dependencies

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

  For more information, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

- Clearing **Interpret vector parameters as 1-D** enables this parameter.
- Selecting `Sample based` enables the following parameter:
  - **Sample time**
- Selecting `Frame based` enables the following parameter:
  - **Frame period**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time

Specify the interval between times that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

Settings

Default: `inf`

This setting indicates that the block output can never change. This setting speeds simulation and generated code by avoiding the need to recompute the block output.

See "Specify Sample Time" in the online documentation for more information.

Dependency

Sampling mode enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the minimum to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Scaling

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

Settings

Default: `Best precision`

`Binary point`
: Specify binary point location.

`Slope and bias`
: Enter slope and bias.

`Best precision`
: Specify best-precision values.

Dependencies

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

Selecting `Binary point` enables:

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

Selecting `Slope and bias` enables:

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Inherit from 'Constant value'`

`Inherit:  Inherit from 'Constant value'`
Use data type of **Constant value**.

`Inherit:  Inherit via back propagation`
Use data type of the driving block.

`double`
Output data type is `double`.

`single`
Output data type is `single`.

`int8`
Output data type is `int8`.

`uint8`
Output data type is `uint8`.

`int16`
Output data type is `int16`.

`uint16`
Output data type is `uint16`.

`int32`
Output data type is `int32`.

`uint32`
Output data type is `uint32`.

`boolean`
Output data type is `boolean`.

`fixdt(1,16)`
Output data type is fixed point `fixdt(1,16)`.

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

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

`Enum: <class name>`
Use an enumerated data type, for example, `Enum: BasicColors`.

`Bus: <object name>`
Data type is a bus object.

`<data type expression>`
Data type is data type object, for example `Simulink.NumericType`.

Do not specify a bus object as the expression.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Block Output Data Types” for more information.

Show data type assistant

Display the **Data Type Assistant**.

Settings

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

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit from 'Constant value'` (default)
- `Inherit via back propagation`

`Built in`

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

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

`Fixed point`

Fixed-point data types.

`Enumerated`

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

`Bus`

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

`Expression`

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

Do not specify a bus object as the expression.

Frame period

Specify the interval between frames that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

Settings

Default: `inf`

Dependency

Sampling mode enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Dependency

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant".

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`

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

`Off`

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

Tip

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

Dependency

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

Signedness

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

Settings

Default: `Signed`

`Signed`
: Specify the fixed-point data as signed.

`Unsigned`
: Specify the fixed-point data as unsigned.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Word length

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

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

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

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

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Bus Support

Using Bus Objects as the Output Data Type

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

Using Structures for the Constant Value

The structure you specify must contain a value for every element of the bus represented by the bus object.

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.

Sampling Mode

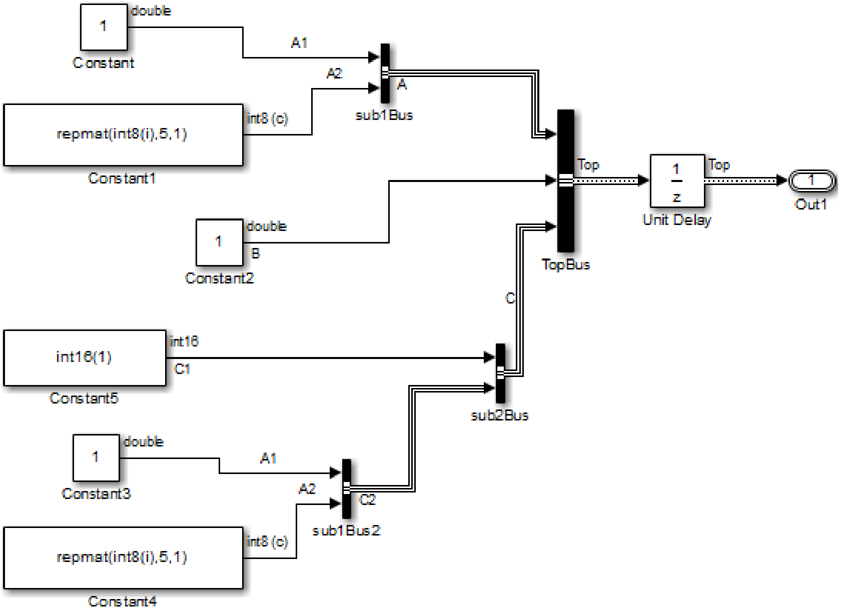
Do not specify a frame-based sampling mode if you specify the **Data type** to be a bus object.

Example of Using a Bus Object for a Constant Block

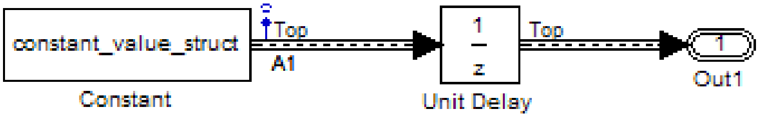
The following example illustrates how using a bus object as an output data type for a Constant block can help to simplify a model.

1. Open the `busic_example` model and update it.

   This model uses six Constant blocks. For details about the model, see “Examples of Partial Structures”.


1
double
Constant
A1
A2
A
sub1Bus
repmat(int8(i),5,1)
int8 (c)
Constant1
Top
1
z
Top
1
Out1
Unit Delay
1
double
B
Constant2
TopBus
C
int16(1)
int16
C1
Constant5
sub2Bus
1
double
A1
Constant3
A2
C2
sub1Bus2
repmat(int8(i),5,1)
int8 (c)
Constant4

2. Open the `constantbus_example` model and update it. This model uses one Constant block that replaces the six Constant blocks in the `busic_example` model.



3 Simulate the `constantbus_example` model. To verify that the output from the Constant block reflects the values from `constant_value_struct`, perform the next two steps.

4 At the MATLAB command line, examine the `constant_value_struct` structure that the Constant block uses for its **Constant value** parameter.

```
constant_value_struct

constant_value_struct =

    A: [1x1 struct]
    B: 5
    C: [1x1 struct]
```

5 Examine the logged data in the `logsout` variable, focusing on the `B` element of the `A1` bus signal. The `constant_value_struct` structure sets the `B` element to `5`.

```
logsout.A1.B.Data
```

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 the following diagnostics as indicated:

- In the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box, set “Mux blocks used to create bus signals” to `error`.
- In the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box, set “Underspecified initialization detection” to `simplified`.

The documentation for these diagnostics explains how to convert your model to handle error messages the diagnostics generate.

Arrays of Buses Not Supported

You cannot use an array of buses with a Constant block.

Examples

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

- `sldemo_auto_climatecontrol`
- `sldemo_boiler`
- `sldemo_bounce`

Characteristics

Direct Feedthrough	N/A
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Enumerated Constant

Purpose

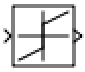
Model discontinuity at zero, with linear gain elsewhere

Library

Discontinuities

Description

The Coulomb and Viscous Friction block models Coulomb (static) and viscous (dynamic) friction. The block models a discontinuity at zero and a linear gain otherwise.



The block output matches the MATLAB result for:

```
y = 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.

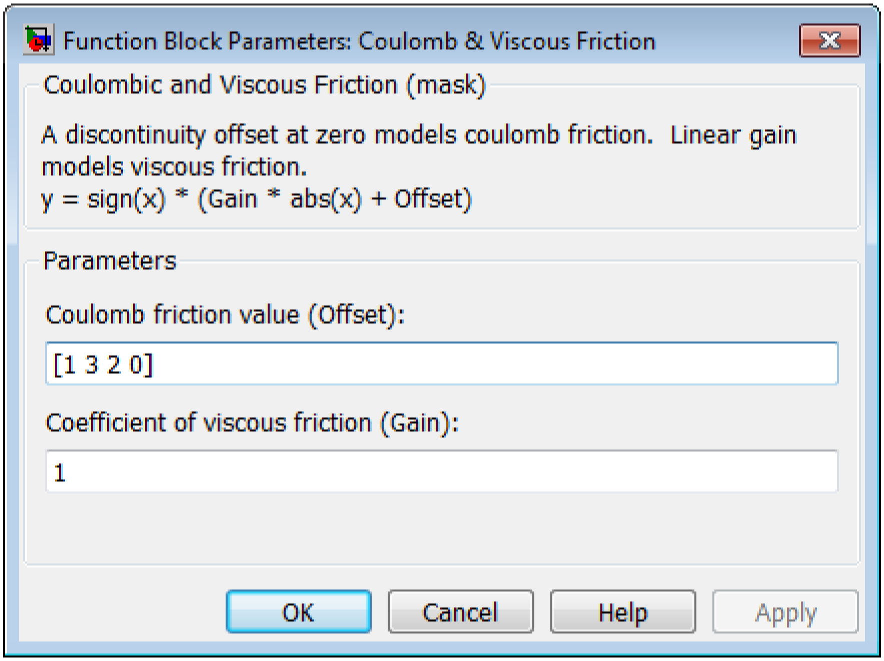
Data Type Support

The Coulomb and Viscous Friction block supports real inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point

The block supports complex inputs only for floating-point data types, `double` and `single`. The output uses the same data type as the input.

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



Function Block Parameters: Coulomb & Viscous Friction

Coulombic and Viscous Friction (mask)

A discontinuity offset at zero models coulomb friction. Linear gain models viscous friction.
y = sign(x) * (Gain * abs(x) + Offset)

Parameters

Coulomb friction value (Offset): [1 3 2 0]

Coefficient of viscous friction (Gain): 1

Parameters and Dialog Box

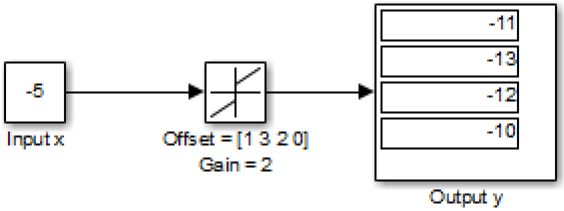
Coulomb friction value
Specify the offset that applies to all input values.

Coefficient of viscous friction
Specify the signal gain for nonzero input values.

Examples

Scalar Input

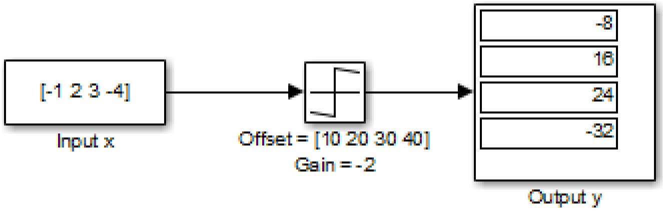
Suppose that you have the following model:


-5
Input x
Offset = [1 3 2 0]
Gain = 2
-11
-13
-12
-10
Output y

In this model, block input `x` and `Gain` are scalar values, but `Offset` is a vector. Therefore, the block uses element-wise scalar expansion to compute the output.

Vector Input

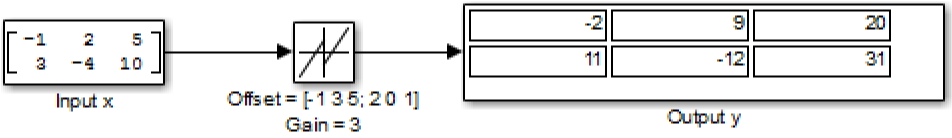
Suppose that you have the following model:


[-1 2 3 -4]
Input x
Offset = [10 20 30 40]
Gain = -2
-8
16
24
-32
Output y

In this model, vector dimensions for block input `x` and `Offset` are the same.

Matrix Input

Suppose that you have the following model:


-1 2 5
3 -4 10
Input x
Offset = [-1 3 5; 2 0 1]
Gain = 3
-2
9
20
11
-12
31
Output y

In this model, matrix dimensions for block input `x` and `Offset` are the same.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	Yes
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	No

Counter Free-Running

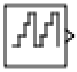
Purpose

Count up and overflow back to zero after reaching maximum value for specified number of bits

Library

Sources

Description



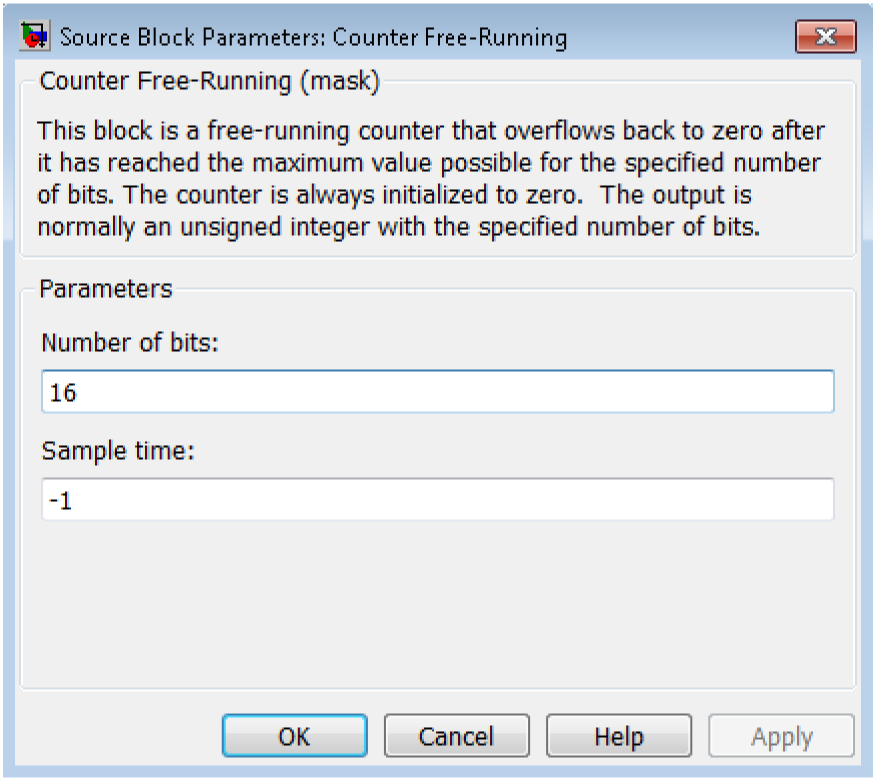
The Counter Free-Running block counts up until reaching the maximum value, $2^{Nbits} - 1$, where $Nbits$ is the number of bits. Then the counter overflows to zero and begins counting up again.

After overflow, the counter always initializes to zero. However, if you select the global doubles override, the Counter Free-Running block does not wrap back to zero.

Data Type Support

The Counter Free-Running block outputs an unsigned integer.

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



Source Block Parameters: Counter Free-Running

Counter Free-Running (mask)

This block is a free-running counter that overflows back to zero after it has reached the maximum value possible for the specified number of bits. The counter is always initialized to zero. The output is normally an unsigned integer with the specified number of bits.

Parameters

Number of bits: 16

Sample time: -1

OK
Cancel
Help
Apply

Parameters and Dialog Box

Number of Bits

Specify the number of bits.

When you use...	**Such as...**	**The block counts up to...**	**Which is...**
A positive integer	8	$2^8 - 1$	255
An unsigned integer expression	`uint8(8)`	$\mathrm{uint8}(2^{\mathrm{uint8}(8)} - 1)$	254

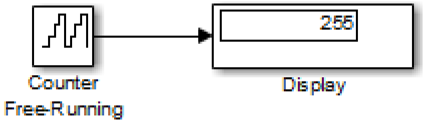
Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation.

Examples

Bit Specification Using a Positive Integer

Suppose that you have the following model:



The block parameters are:

Parameter	Setting
Number of Bits	`8`
Sample time	`-1`

The solver options for the model are:

Parameter	Setting
Stop time	`255`
Type	`Fixed-step`
Solver	`discrete (no continuous states)`
Fixed-step size	`1`

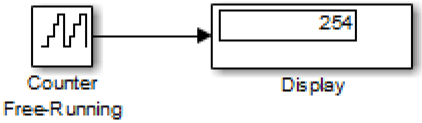
At t = 255, the counter reaches the maximum value:

$$2^8 - 1$$

If you change the stop time of the simulation to 256, the counter wraps to zero.

Bit Specification Using an Unsigned Integer Expression

Suppose that you have the following model:



The block parameters are:

Parameter	Setting
Number of Bits	`uint8(8)`
Sample time	`-1`

The solver options for the model are:

Parameter	Setting
Stop time	`254`
Type	`Fixed-step`
Solver	`discrete (no continuous states)`
Fixed-step size	`1`

At t = 254, the counter reaches the maximum value:

$$\mathrm{uint8}(2^{\mathrm{uint8}(8)} - 1)$$

If you change the stop time of the simulation to 255, the counter wraps to zero.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Zero-Crossing Detection	No

See Also

Counter Limited

Counter Limited

Purpose

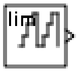
Count up and wrap back to zero after outputting specified upper limit

Library

Sources

Description

The Counter Limited block counts up until the specified upper limit is reached. Then the counter wraps back to zero, and restarts counting up. The counter always initializes to zero.



You can specify the upper limit with the **Upper limit** parameter.

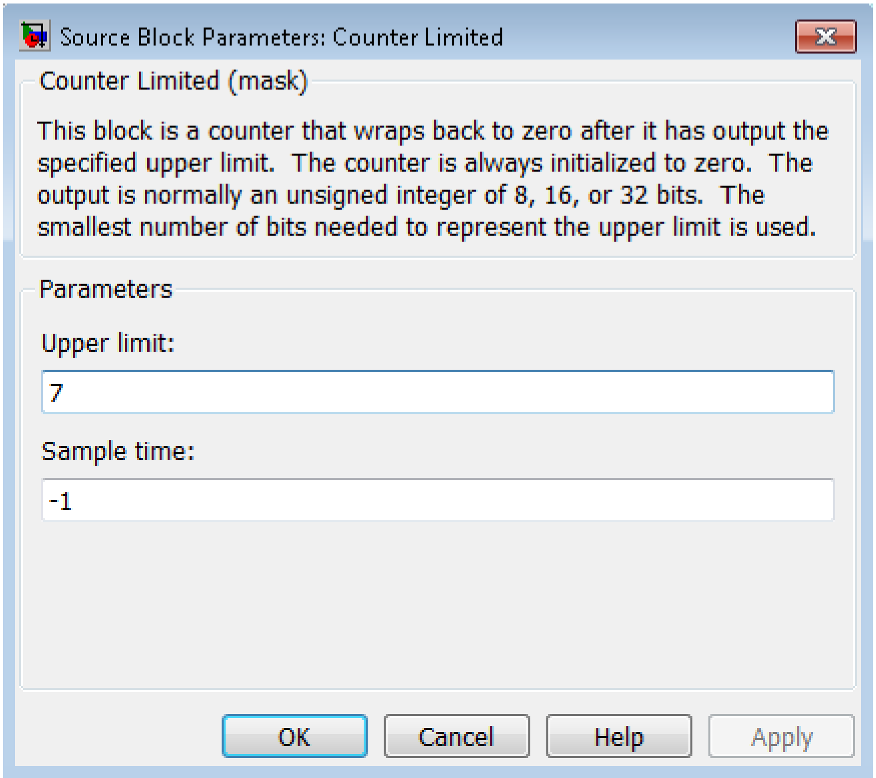
You can specify the sample time with the **Sample time** parameter. A **Sample time** of `-1` means that the sample time is inherited.

The output is an unsigned integer of 8, 16, or 32 bits, with the smallest number of bits needed to represent the upper limit.

Data Type Support

The Counter Limited block outputs an unsigned integer.

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



Counter Limited (mask)

This block is a counter that wraps back to zero after it has output the specified upper limit. The counter is always initialized to zero. The output is normally an unsigned integer of 8, 16, or 32 bits. The smallest number of bits needed to represent the upper limit is used.

Parameters

Upper limit: 7

Sample time: -1

Parameters and Dialog Box

Upper limit

Specify the upper limit.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation.

Examples

The following Simulink examples show how to use the Counter Limited block:

- `sldemo_tonegen_fixpt`

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No

See Also

Counter Free-Running

Purpose

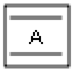
Define data store

Library

Signal Routing

Description

The Data Store Memory block defines and initializes a named shared data store, which is a memory region usable by Data Store Read and Data Store Write blocks that specify the same data store name.



The location of the Data Store Memory block that defines a data store determines which Data Store Read and Data Store Write blocks can access the data store:

- If the Data Store Memory block is in the *top-level system*, Data Store Read and Data Store Write blocks anywhere in the model can access the data store.
- If the Data Store Memory block is in a *subsystem*, Data Store Read and Data Store Write blocks in the same subsystem or in any subsystem below it in the model hierarchy can access the data store.

Data Store Read or Data Store Write blocks cannot access a Data Store Memory block that is either in a model that contains a Model block or in a referenced model.

Do not include a Data Store Memory block in a subsystem that a For Each Subsystem block represents.

Use the **Initial value** parameter to initialize the data store. Specify a scalar value or an array of values in the **Initial value** parameter. The dimensions of the array determine the dimensionality of the data store. Any data written to the data store must have the dimensions designated by the **Initial value** parameter. Otherwise, an error occurs.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. For details, see:

- “Order Data Store Access”
- “Data Store Diagnostics”

- “Log Data Stores”

You can use `Simulink.Signal` objects in addition to, or instead of, Data Store Memory blocks to define data stores. A data store defined in the *base* workspace with a signal object is a *global* data store. Global data stores are accessible to every model, including all referenced models. See “Data Stores” for more information.

Data Type Support

The Data Store Memory block stores real or complex signals of any data type that Simulink supports, including fixed-point, bus object, and enumerated data types.

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

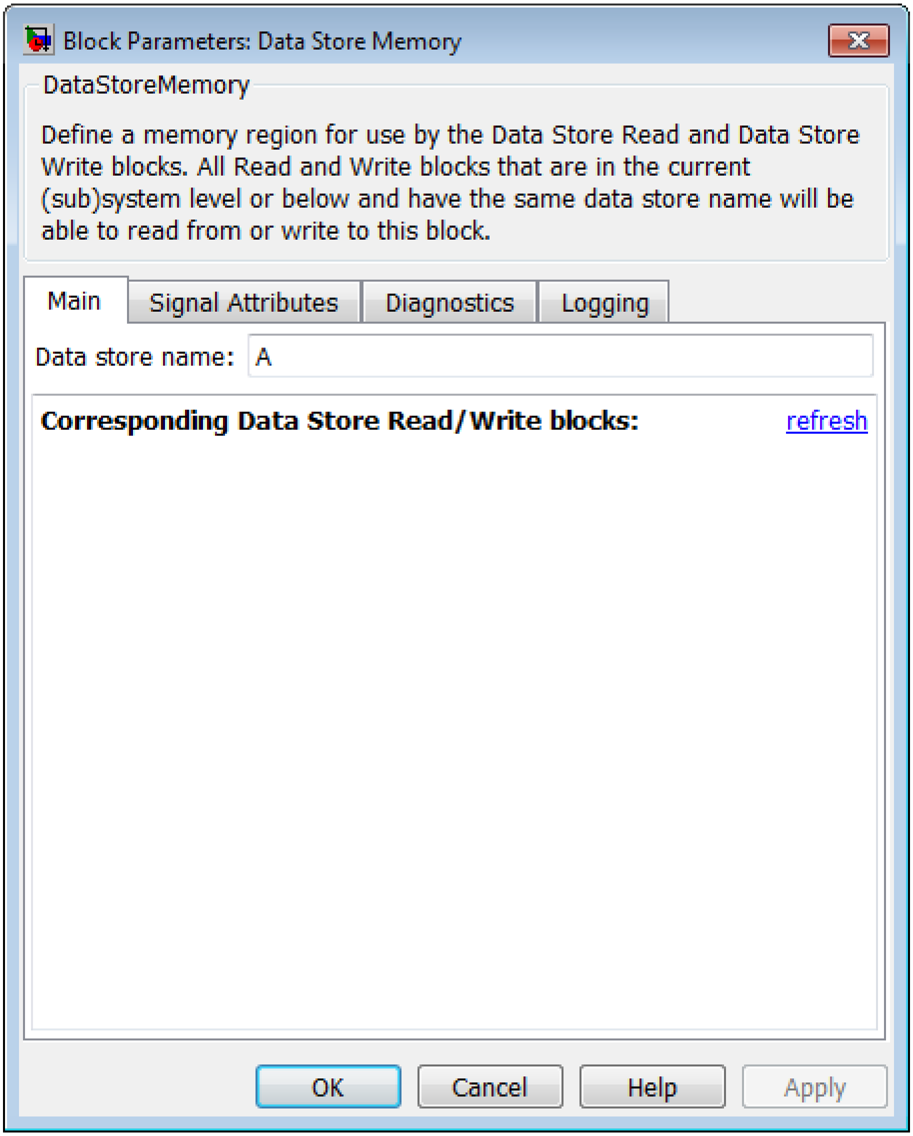
For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

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

You can use arrays of buses with a Data Store Memory block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Parameters and Dialog Box

The **Main** pane of the Data Store Memory block dialog box appears as follows:



Block Parameters: Data Store Memory

DataStoreMemory

Define a memory region for use by the Data Store Read and Data Store Write blocks. All Read and Write blocks that are in the current (sub)system level or below and have the same data store name will be able to read from or write to this block.

Data store name: A

Corresponding Data Store Read/Write blocks:

Main
Signal Attributes
Diagnostics
Logging
refresh
OK
Cancel
Help
Apply

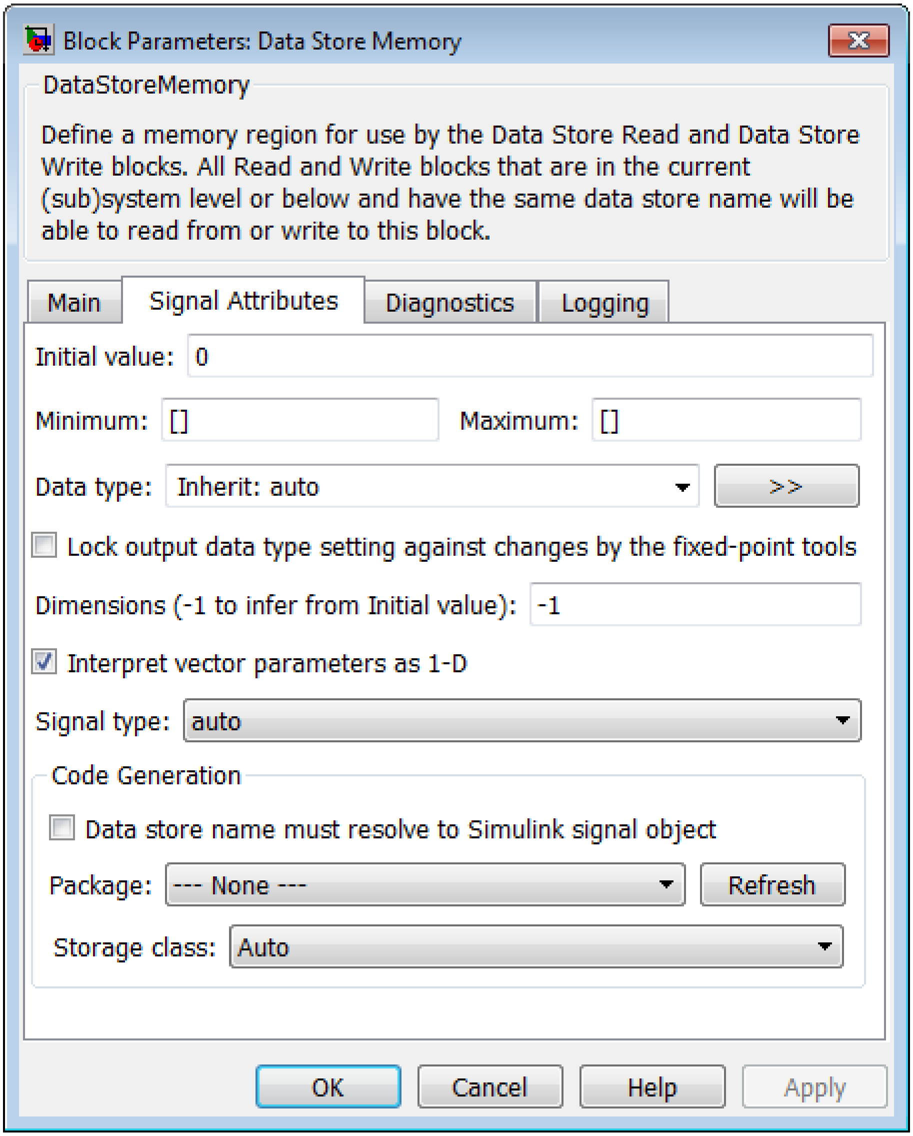
Data store name

Specify a name for the data store you are defining with this block. Data Store Read and Data Store Write blocks with the same name can read from, and write to, the data store initialized by this block. The name can represent a Data Store Memory block or a sign object defined to be a data store.

Corresponding Data Store Read/Write blocks

List all the Data Store Read and Data Store Write blocks that have the same data store name as the current block, and that are in the current system or in any subsystem below it in the model hierarchy. Double-click a block in this list to highlight the block and bring it to the foreground.

The **Signal Attributes** pane of the Data Store Memory block dialog box appears as follows:



Block Parameters: Data Store Memory

DataStoreMemory

Define a memory region for use by the Data Store Read and Data Store Write blocks. All Read and Write blocks that are in the current (sub)system level or below and have the same data store name will be able to read from or write to this block.

Main
Signal Attributes
Diagnostics
Logging

Initial value: 0

Minimum: []

Maximum: []

Data type: Inherit: auto

>>

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

Dimensions (-1 to infer from Initial value): -1

☑ Interpret vector parameters as 1-D

Signal type: auto

Code Generation

☐ Data store name must resolve to Simulink signal object

Package: --- None ---

Storage class: Auto

OK
Cancel
Help
Apply

Initial value

Specify the initial value or values of the data store. The dimensions of this value determine the dimensions of data that may be written to the data store. The **Minimum** parameter specifies the minimum value for this parameter, and the **Maximum** parameter specifies the maximum value.

Initial value dimensions must match the dimensions that you specify in the **Signal Attributes > Dimensions** parameter, unless the initial value is a MATLAB structure.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). This number must be a finite real double scalar value.

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

Simulink uses the minimum value to perform:

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

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). This number must be a finite real double scalar value.

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

Simulink uses the maximum value to perform:

- Parameter range checking (see "Check Parameter Values")
- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Data type

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

- A rule that inherits a data type (for example, `Inherit: auto`)
- The name of a built-in data type (for example, `single`)
- The name of a data type object (for example, a `Simulink.NumericType` object)
- An expression that evaluates to a data type (for example, `fixdt(1,16,0)`). Do not specify a bus object as the data type in an expression; use `Bus: <object name>` to specify a bus data type.
- `Bus: <object name>`; enter the name of a bus object that you want to use to define the structure of the bus. The bus must be a nonvirtual bus. If you need to create or change a bus object, click the **Show data type assistant** button and then click the **Edit** button to the right of the **Bus object** field to open the Simulink Bus Editor. For details about the Bus Editor, see "Manage Bus Objects with the Bus Editor"

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

See “Specify Block Output Data Types”.

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

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

Signal type

Specify the numeric type, real or complex, of the values in the data store.

Dimensions (-1 to infer from Initial value)

Specify dimensions that match the dimensions of the **Initial value** dimensions, unless you specify a MATLAB structure for the initial value. For example, if you use a MATLAB structure for the initial value, then you need to specify dimensions to initialize an array of buses with this MATLAB structure.

Interpret vector parameters as 1-D

If you enable this option and specify the **Initial value** parameter as a column or row matrix, Simulink initializes the data store to a 1-D array whose elements are equal to the elements of the row or column vector. See “Determining the Output Dimensions of Source Blocks”.

Data store must resolve to Simulink signal object

Specify that Simulink software, when compiling the model, searches the model and base workspace for a `Simulink.Signal` object having the same name, as described in “Symbol Resolution”. If Simulink does not find such an object, the compilation stops, with an error. Otherwise, Simulink compares the attributes of the signal object to the corresponding attributes of the Data Store Memory block. If the block and the object attributes are inconsistent, Simulink halts model compilation and displays an error.

Package

Select a package that defines the custom storage class you want to apply. If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Storage class

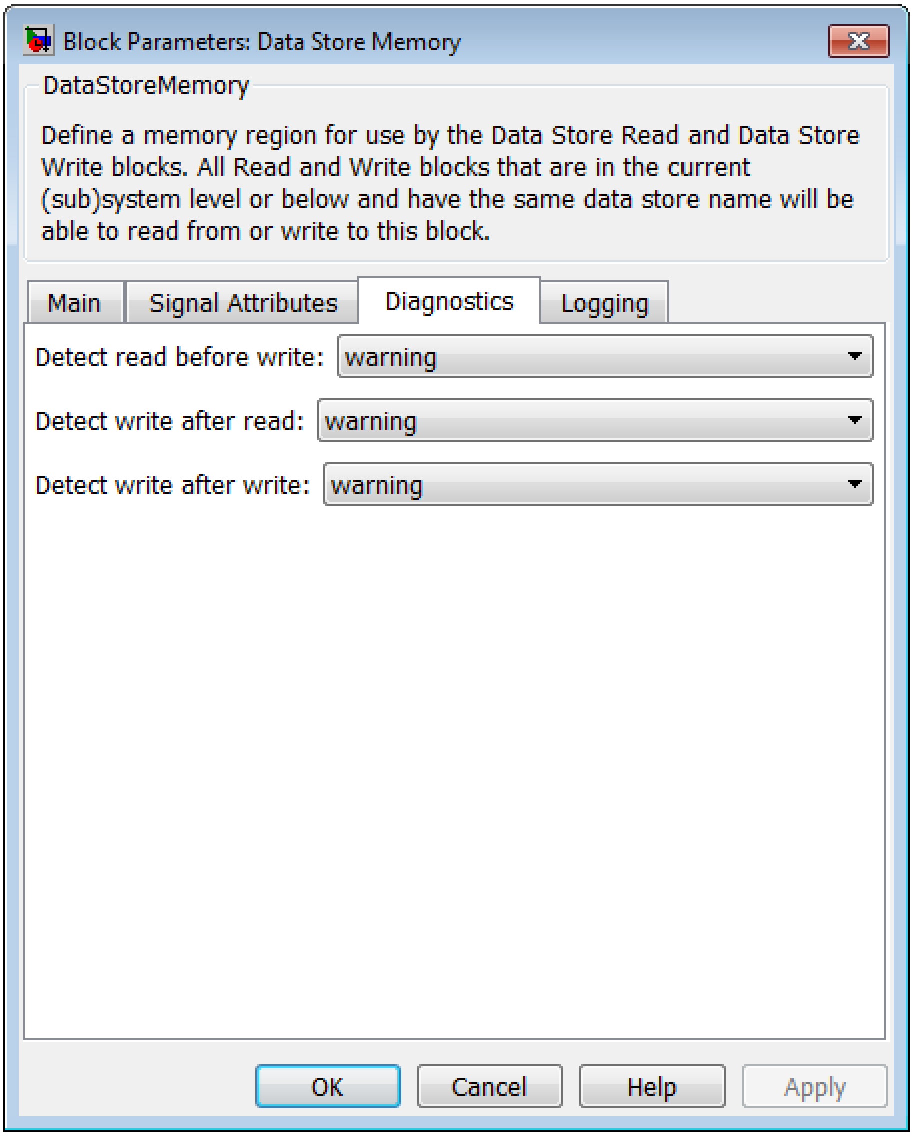
Select a custom storage class for the signal object. When no **Package** is selected, setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Storage type qualifier**.

Storage type qualifier

Specify the Simulink Coder storage type qualifier.

See “States” in the Simulink Coder documentation for more information.

The **Diagnostics** pane of the Data Store Memory block dialog box appears as follows:


Block Parameters: Data Store Memory
Main
Signal Attributes
Diagnostics
Logging
OK
Cancel
Help
Apply

DataStoreMemory

Define a memory region for use by the Data Store Read and Data Store Write blocks. All Read and Write blocks that are in the current (sub)system level or below and have the same data store name will be able to read from or write to this block.

Detect read before write: warning

Detect write after read: warning

Detect write after write: warning

Detect read before write

Select the diagnostic action to take if the model attempts to read data from a data store to which it has not written data in this time step. See also the "Detect read before write" diagnostic in the **Data Store Memory Block** section of the **Model Configuration Parameters > Diagnostics > Data Validity** pane.

Default: `warning`

`none`
: Take no action.

`warning`
: Display a warning.

`error`
: Terminate the simulation and display an error message.

Detect write after read

Select the diagnostic action to take if the model attempts to write data to the data store after previously reading data from it in the current time step. See also the "Detect write after read" diagnostic in the **Data Store Memory Block** section of the **Model Configuration Parameters > Diagnostics > Data Validity** pane.

Default: `warning`

`none`
: Take no action.

`warning`
: Display a warning.

`error`
: Terminate the simulation and display an error message.

Detect write after write

Select the diagnostic action to take if the model attempts to write data to the data store twice in succession in the current time step. See also the "Detect write after write" diagnostic in the **Data Store Memory Block** section of the **Model Configuration Parameters > Diagnostics > Data Validity** pane.

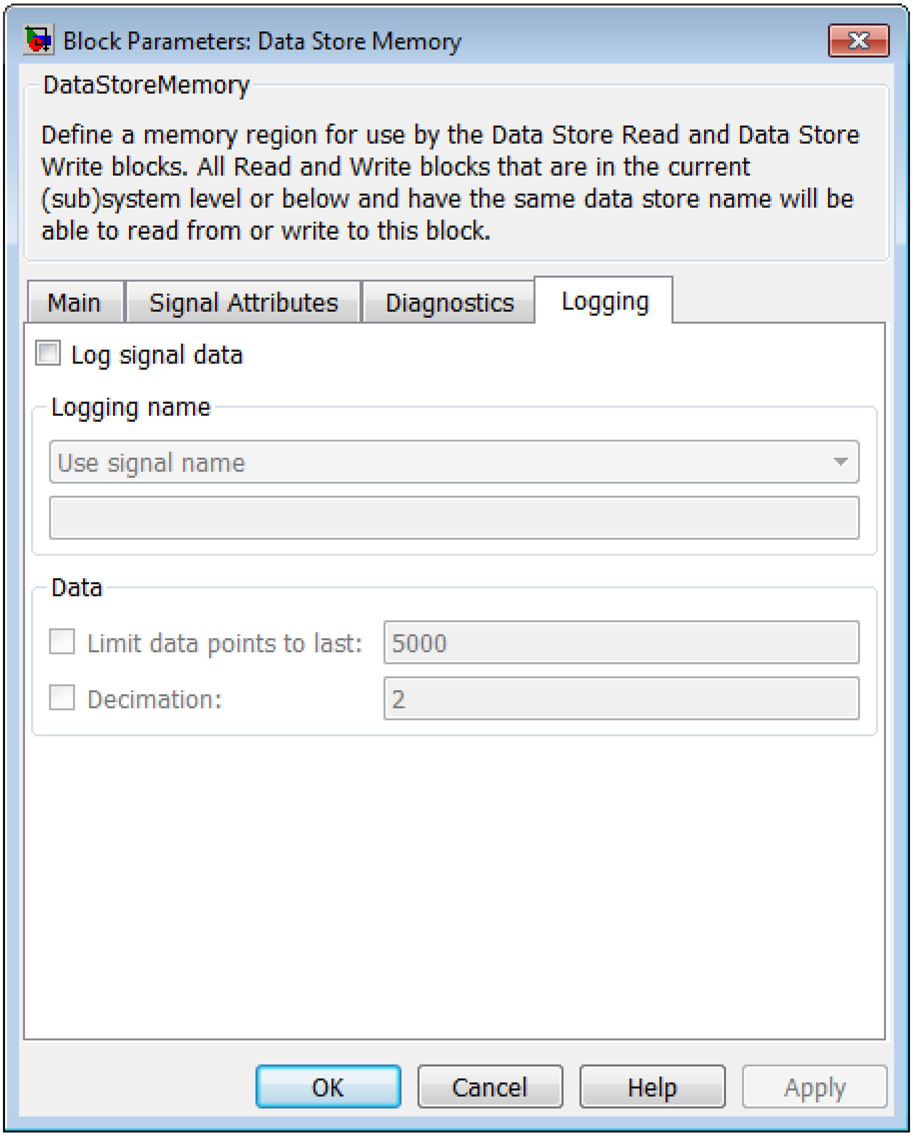
Default: `warning`

`none`
: Take no action.

`warning`
: Display a warning.

`error`
: Terminate the simulation and display an error message.

The **Logging** pane of the Data Store Memory block dialog box appears as follows:



Block Parameters: Data Store Memory

DataStoreMemory

Define a memory region for use by the Data Store Read and Data Store Write blocks. All Read and Write blocks that are in the current (sub)system level or below and have the same data store name will be able to read from or write to this block.

Main
Signal Attributes
Diagnostics
Logging

☐ Log signal data

Logging name

Use signal name

Data

☐ Limit data points to last: 5000

☐ Decimation: 2

OK
Cancel
Help
Apply

Log signal data

Select this option to save the values of this signal to the MATLAB workspace during simulation. See “Signal Logging” for details.

Logging name

Use this pair of controls, consisting of a list box and an edit field, to specify the name associated with logged signal data.

Simulink uses the signal name as its logging name by default. To specify a custom logging name, select `Custom` from the list box and enter the custom name in the adjacent edit field.

Data

Use this group of controls to limit the amount of data that Simulink logs for this signal.

- **Limit data points to last**: Discard all but the last `N` data points, where `N` is the number that you enter in the adjacent edit field.
- **Decimation**: Log every `N`th data point, where `N` is the number that you enter in the adjacent edit field. For example, suppose that your model uses a fixed-step solver with a step size of `0.1` s. If you select this option and accept the default decimation value (2), Simulink records data points for this signal at times `0.0`, `0.2`, `0.4`, and so on.

For more information, see “Log Data Stores”

Characteristics

Sample Time	N/A
Dimensionalized	Yes
Multidimensionalized	Yes

See Also

- “Data Stores”
- “Access Data Stores with Simulink Blocks”
- Data Store Read

- Data Store Write
- “Log Data Stores”

Data Store Read

Purpose

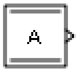
Read data from data store

Library

Signal Routing

Description

The Data Store Read block copies data from the named data store to its output. More than one Data Store Read block can read from the same data store.



The data store from which the data is read is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see "Data Stores" and Data Store Memory.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. See "Order Data Store Access" and "Data Store Diagnostics" for details.

Data Type Support

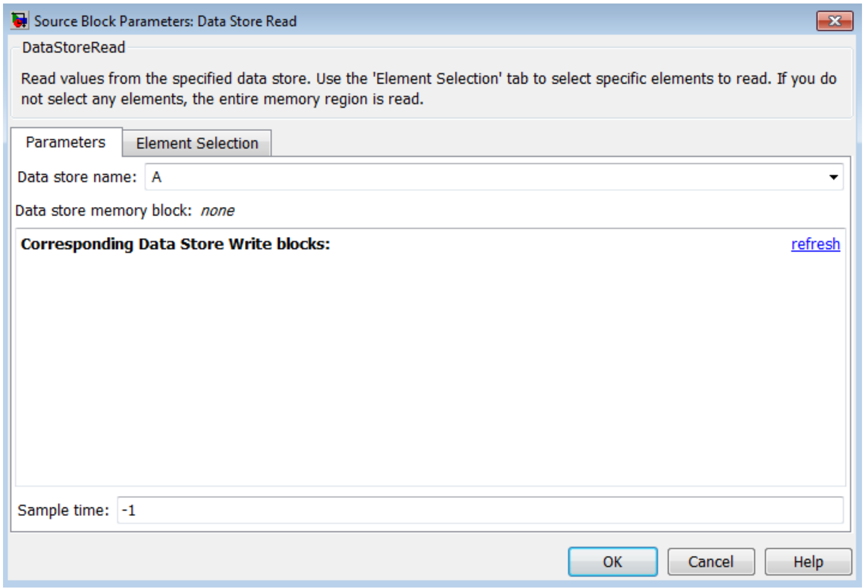
The Data Store Read block can output a real or complex signal of any data type that Simulink supports, including fixed-point, bus object, and enumerated data types.

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

You can use arrays of buses with a Data Store Read block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

Parameters and Dialog Box

The **Parameters** pane of the Data Store Read block dialog box appears as follows:



Read values from the specified data store. Use the 'Element Selection' tab to select specific elements to read. If you do not select any elements, the entire memory region is read.

Data store name: A

Data store memory block: *none*

Corresponding Data Store Write blocks:

Sample time: -1

Data store name

Specifies the name of the data store from which this block reads data. The adjacent pull-down list lists the names of Data Store Memory blocks that exist at the same level in the model as the Data Store Read block or at higher levels. The pulldown list also includes all `Simulink.Signal` objects in the base and model workspaces. To change the name, select a name from the pull-down list or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink software searches the model upwards from this block's level for a Data Store Memory block having the specified data store name. If Simulink software does not find such a block, it searches the model workspace and the MATLAB workspace for a `Simulink.Signal` object having the same name. See “Symbol Resolution” for more information about the search path.

If Simulink software finds the signal object, it creates a hidden Data Store Memory block at the model’s root level having the properties specified by the signal object and an initial value of `0`. If Simulink software finds neither the Data Store Memory block nor the signal object, it halts the compilation and displays an error.

Data store memory block

This field lists the Data Store Memory block that initialized the store from which this block reads.

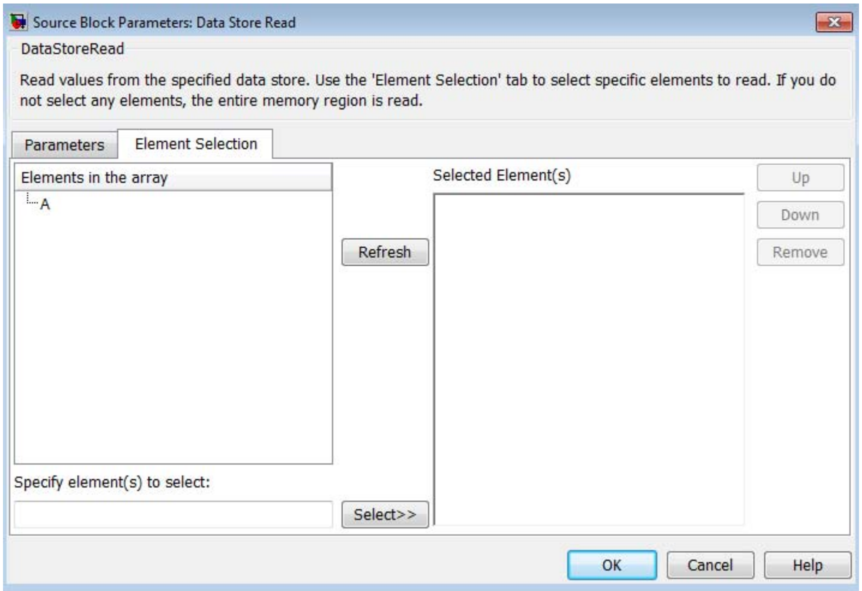
Data store write blocks

This parameter lists all the Data Store Write blocks with the same data store name as this block that are in the same (sub)system or in any subsystem below it in the model hierarchy. Double-click any entry on this list to highlight the block and bring it to the foreground.

Sample time

The sample time, which controls when the block reads from the data store. A value of `-1` indicates that the sample time is inherited. See “Specify Sample Time” for more information.

The **Element Selection** pane of the Data Store Read block dialog box appears as follows:


Source Block Parameters: Data Store Read
DataStoreRead
Parameters
Element Selection
Elements in the array
A
Refresh
Selected Element(s)
Up
Down
Remove
Specify element(s) to select:
Select>>
OK
Cancel
Help

Read values from the specified data store. Use the 'Element Selection' tab to select specific elements to read. If you do not select any elements, the entire memory region is read.

Use the **Element Selection** pane to select a subset of the bus or matrix elements defined for the associated data store. The Data Store Read block icon reflects the elements that you specify. For details, see "Accessing Specific Bus and Matrix Elements".

Elements in the array or **Signals in the bus** (Prompt is specific to the type of data.)

For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

For data stores with a bus data type, you can expand the tree to view the bus elements. For data stores with arrays, you can read the whole data store, or you can specify one or more elements of the whole data store.

You can select an element and then use one of the following approaches:

- Click **Select>>** to display that element (and all its subelements) in the **Selected element(s)** list.
- Use the **Specify element(s) to select** edit box to specify the bus or matrix elements that you want to select for reading. Then click **Select>>**.

To refresh the display to reflect modifications to the bus or matrix used in the data store, click **Refresh**.

Specify element(s) to select

Enter a MATLAB expression to define the specific element that you want to read. For example, for a data store named `DSM` that has maximum dimensions of `[3,5]`, you could enter expressions such as `DSM(2, 4)` or `DSM([1 3], 2)` in the edit box and then click **Select>>**.

To apply the element selection, click **OK**.

Selected Element(s)

Displays the elements that you select. The Data Store Read block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click **Up** or **Down**. Changing the order of the elements in the list changes the order of the ports. To remove an element, click **Remove**.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes
Multidimensionalized	Yes

See Also

- “Data Stores”
- “Access Data Stores with Simulink Blocks”
- Data Store Memory

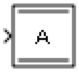
- Data Store Write

Data Store Write

Purpose Write data to data store

Library Signal Routing

Description The Data Store Write block copies the value at its input to the named data store. Each write operation performed by a Data Store Write block writes over the data store, replacing the previous contents.



The data store to which this block writes is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see “Data Stores” and Data Store Memory. The size of the data store is set by the signal object or the Data Store Memory block that defines and initializes the data store. Each Data Store Write block that writes to that data store must write the same amount of data.

More than one Data Store Write block can write to the same data store. However, if two Data Store Write blocks attempt to write to the same data store during the same simulation step, results are unpredictable.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. For details, see “Order Data Store Access” and “Data Store Diagnostics”.

You can log the values of a local or global data store data variable for all the steps in a simulation. For details, see “Log Data Stores”.

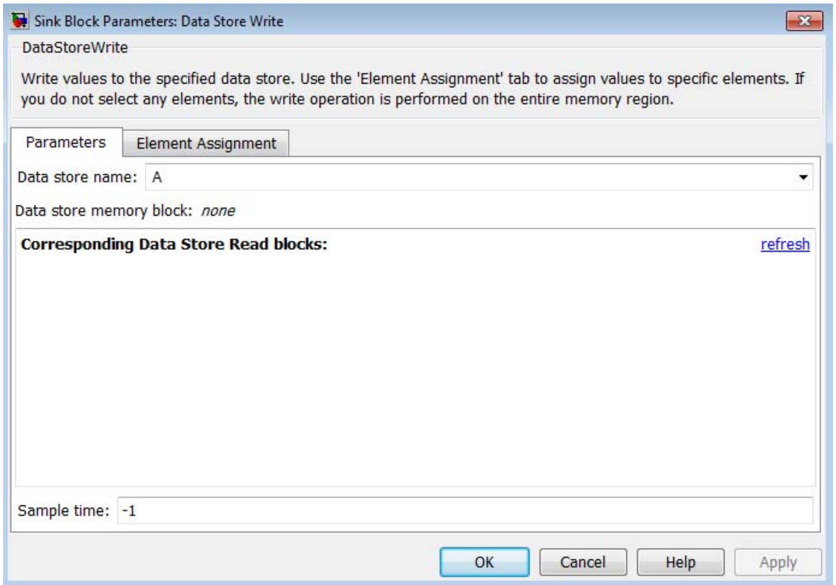
Data Type Support The Data Store Write block accepts a real or complex signal of any data type that Simulink supports, including fixed-point, bus object, and enumerated data types.

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

You can use an array of buses with a Data Store Write block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Parameters and Dialog Box

The **Parameters** pane of the Data Store Write block dialog box appears as follows:



Sink Block Parameters: Data Store Write

DataStoreWrite

Write values to the specified data store. Use the 'Element Assignment' tab to assign values to specific elements. If you do not select any elements, the write operation is performed on the entire memory region.

Data store name: A

Data store memory block: *none*

Corresponding Data Store Read blocks:

Sample time: -1

Data store name

Specifies the name of the data store to which this block writes data. The adjacent pull-down list lists the names of Data Store Memory blocks that exist at the same level in the model as the Data Store Write block or at higher levels. The pulldown list also includes all `Simulink.Signal` objects in the base and model workspaces. To change the name, select a name from the pull-down list or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink software searches the model upwards from this block's level for a Data Store Memory block having the specified data store name. If Simulink does not find such a block, it searches the model workspace and the MATLAB workspace for a `Simulink.Signal` object having the same name. If Simulink software finds neither the Data Store Memory block nor the signal object, it halts the compilation and displays an error. See “Symbol Resolution” for more information about the search path.

If Simulink finds a signal object, it creates a hidden Data Store Memory block at the model’s root level having the properties specified by the signal object and an initial value set to a matrix of zeros. The dimensions of that matrix are inherited from the `Dimensions` property of the signal object.

Data store memory block

This field lists the Data Store Memory block that initialized the store to which this block writes.

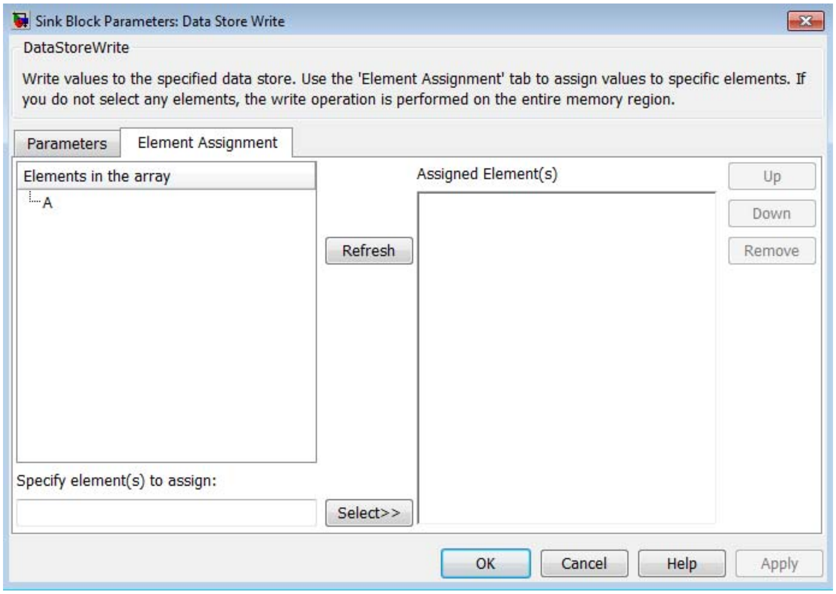
Data store read blocks

This parameter lists all the Data Store Read blocks with the same data store name as this block that are in the same (sub)system or in any subsystem below it in the model hierarchy. Double-click any entry on this list to highlight the block and bring it to the foreground.

Sample time

Specify the sample time that controls when the block writes to the data store. A value of `-1` indicates that the sample time is inherited. See “Specify Sample Time” for more information.

The **Element Assignment** pane of the Data Store Write block dialog box appears as follows:


Sink Block Parameters: Data Store Write
DataStoreWrite
Parameters
Element Assignment
Elements in the array
A
Refresh
Assigned Element(s)
Up
Down
Remove
Specify element(s) to assign:
Select>>
OK
Cancel
Help
Apply

Write values to the specified data store. Use the 'Element Assignment' tab to assign values to specific elements. If you do not select any elements, the write operation is performed on the entire memory region.

Use the **Element Assignment** pane to assign a subset of the bus or matrix elements defined for writing to the associated data store. The Data Store Write block icon reflects the elements that you specify. For details, see “Accessing Specific Bus and Matrix Elements”.

Elements in the array or **Signals in the bus** (Prompt is specific to the type of data.)

For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

For data stores with a bus data type, you can expand the tree to view the bus elements. For data stores with arrays, you can write the whole data store, or you can assign one or more elements to the whole data store.

You can select an element and then use one of the following approaches:

- Click **Select>>** to display that element (and all its subelements) in the **Assigned element(s)** list.
- Use the **Specify element(s) to assign** edit box to specify the bus or matrix elements that you want to select for reading. Then click **Select>>**.

To refresh the display to reflect modifications to the bus or matrix used in the data store, click **Refresh**.

Specify element(s) to assign

Enter a MATLAB expression to define the specific element that you want to write. For example, for a data store named `DSM` that has maximum dimensions of `[3,5]`, you could enter expressions such as `DSM(2, 4)` or `DSM([1 3], 2)` in the edit box. Then click **Select>>**.

To apply the element selection, click **OK**.

Assigned Element(s)

Displays the elements that you selected for assignment. The Data Store Write block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click **Up** or **Down**. Changing the order of the elements in the list changes the order of the ports. To remove an element, click **Remove**.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes
Multidimensionalized	Yes

See Also

- “Data Stores”
- “Access Data Stores with Simulink Blocks”

- Data Store Memory
- Data Store Read
- “Log Data Stores”

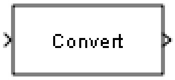
Data Type Conversion

Purpose

Convert input signal to specified data type

Library

Signal Attributes



Description

The Data Type Conversion block converts an input signal of any Simulink data type to the data type you specify for the **Output data type** parameter. The input can be any real- or complex-valued signal. If the input is real, the output is real. If the input is complex, the output is complex.

Note This block requires that you specify the data type and scaling for the conversion. If you want to inherit the data type or scaling from an input signal, use the Data Type Conversion Inherited block.

The **Input and output to have equal** parameter controls how the block handles the input in the context of fixed point data representation. For more information, see "Fixed-Point Numbers".

- Select `Real World Value (RWV)` to treat the input as $V = SQ + B$, where S is the slope and B is the bias. V is used to produce $Q = (V - B)/S$, which is stored in the output.
- Select `Stored Integer (SI)` to treat the input as a stored integer, Q. The value of Q is directly used to produce the output. In this mode, the input and output are identical, except that the input is a raw integer lacking proper scaling information. Selecting `Stored Integer` might be useful in the following cases:
  - Generating code for a fixed-point processor

  The resulting code uses only integers and does not use floating-point operations.

  - Partitioning your model based on hardware characteristics

    For example, part of your model might involve simulating hardware that produces integers as output.

For more information on scaling of fixed point numbers, see “Scaling”.

Casting 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 should 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 should match the underlying value of an enumerated value; otherwise, an error occurs during simulation.

   You can enable the block’s **Saturate on integer overflow** parameter so that Simulink uses the default value of the enumerated type when the value input to the block does not match the underlying value of an enumerated value. See “Type Casting for Enumerations”.

You cannot use a Data Type Conversion block in the following cases.

- To cast a non-integer numeric signal to an enumerated signal
- To cast a complex signal to an enumerated signal, regardless of the data types of the complex signal’s real and imaginary parts.

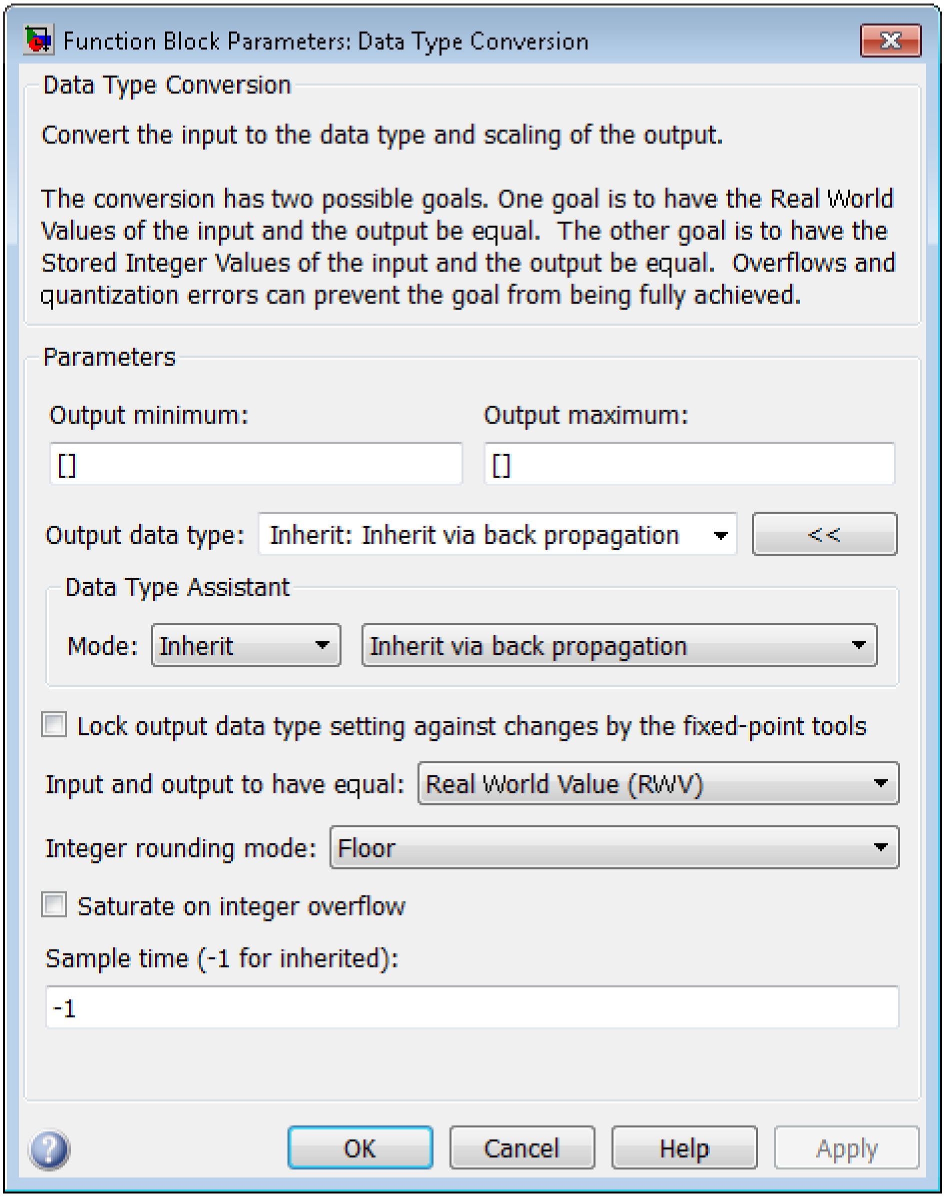
See “About Simulink Enumerations” for information on working with enumerated types.

Data Type Support

The Data Type Conversion block handles any data type that Simulink supports, including fixed-point and enumerated data types.

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

Parameters and Dialog Box



Function Block Parameters: Data Type Conversion
Data Type Conversion
Parameters
Data Type Assistant
<<
OK
Cancel
Help
Apply

Convert the input to the data type and scaling of the output.

The conversion has two possible goals. One goal is to have the Real World Values of the input and the output be equal. The other goal is to have the Stored Integer Values of the input and the output be equal. Overflows and quantization errors can prevent the goal from being fully achieved.

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via back propagation

Mode: Inherit — Inherit via back propagation

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

Input and output to have equal: Real World Value (RWV)

Integer rounding mode: Floor

☐ Saturate on integer overflow

Sample time (-1 for inherited): -1

Show data type assistant

Display the **Data Type Assistant**.

Settings

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

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

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

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

Settings

Default: Off

☑ On
Locks the output data type setting for this block.

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Input and output to have equal

Specify which type of input and output should be equal, in the context of fixed point data representation. See <INSERT LINK HERE>.

Settings

Default: `Real World Value (RWV)`

`Real World Value (RWV)`
: Specifies the goal of making the `Real World Value (RWV)` of the input equal to the `Real World Value (RWV)` of the output.

`Stored Integer (SI)`
: Specifies the goal of making the `Stored Integer (SI)` value of the input equal to the `Stored Integer (SI)` value of the output.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
: Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`
: Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`
: Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`
: Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`
: Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`
: Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`
: Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see “Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the minimum to perform:

- Parameter range checking (see "Check Parameter Values") for some blocks
- Simulation range checking (see "Signal Ranges" and "Enabling Simulation Range Checking")
- Automatic scaling of fixed-point data types

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Inherit via back propagation`

`Inherit:  Inherit via back propagation`
Use data type of the driving block.

`double`
Output data type is `double`.

`single`
Output data type is `single`.

`int8`
Output data type is `int8`.

`uint8`
Output data type is `uint8`.

`int16`
Output data type is `int16`.

`uint16`
Output data type is `uint16`.

`int32`
Output data type is `int32`.

`uint32`
Output data type is `uint32`.

`boolean`
Output data type is `boolean`. The Data Type Conversion block converts real, nonzero numeric values (including `NaN` and `Inf`) to `boolean` `true` (1).

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

`fixdt(1,16,2^0,0)`
: Output data type is fixed point `fixdt(1,16,2^0,0)`.

`Enum: <class name>`
: Use an enumerated data type, for example, `Enum: BasicColors`.

`<data type expression>`
: Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specify Block Output Data Types”.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables `Inherit via back propagation`.

`Built in`

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

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

`Fixed point`

Fixed-point data types.

`Enumerated`

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

`Expression`

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

Dependency

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant".

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`

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

`Off`

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

Tip

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

Dependency

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

Signedness

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

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Word length

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

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Scaling

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

Settings

Default: `Best precision`

`Binary point`
Specify binary point location.

`Slope and bias`
Enter slope and bias.

`Best precision`
Specify best-precision values.

Dependencies

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

Selecting `Binary point` enables:

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

Selecting `Slope and bias` enables:

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

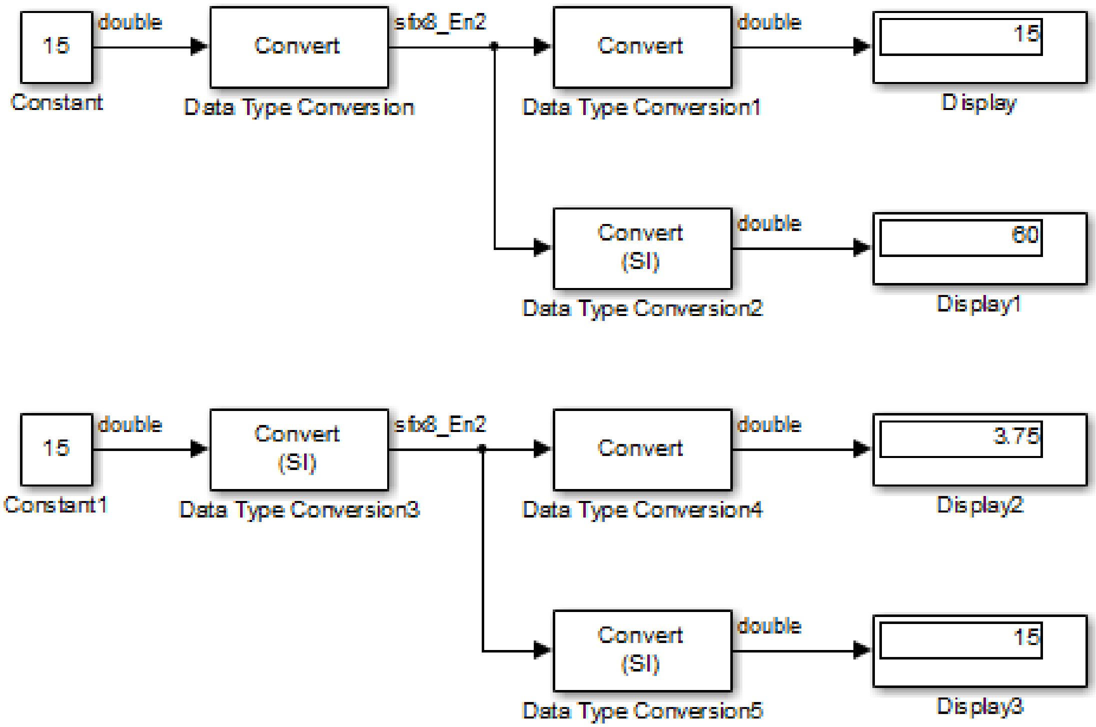
See Also

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

Examples

Real World Values Versus Stored Integers

The following model uses the Data Type Conversion block to explain the difference between a real-world value and a stored integer. Consider these two fixed-point block diagrams:

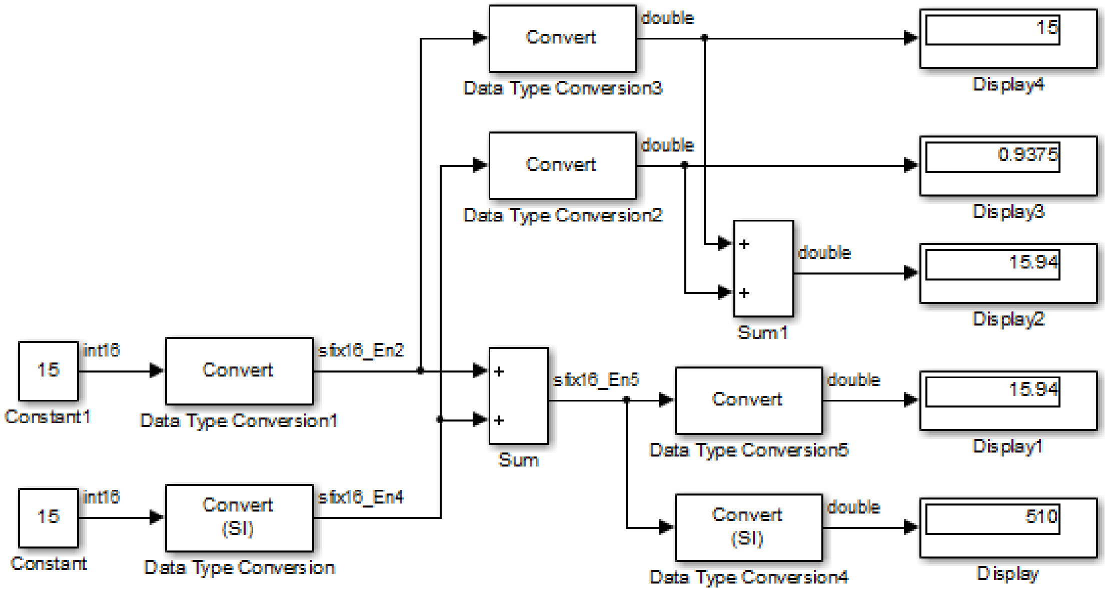

15
double
Constant
Convert
Data Type Conversion
sfix8_En2
Convert
Data Type Conversion1
double
15
Display
Convert
(SI)
Data Type Conversion2
double
60
Display1
15
double
Constant1
Convert
(SI)
Data Type Conversion3
sfix8_En2
Convert
Data Type Conversion4
double
3.75
Display2
Convert
(SI)
Data Type Conversion5
double
15
Display3

In the top block diagram, the Data Type Conversion block treats the input as a real-world value, and maps that value to an 8-bit signed generalized fixed-point data type with a scaling of 2^{-2} (i.e., binary point scaling with slope = 2^{-2} and bias = 0). When the value is then output from the Data Type Conversion1 block as a real-world value, the scaling and data type information is retained and the output value is 001111.00, or 15. When the value is output from the Data Type Conversion2 block as a stored integer, the scaling and data type information is not retained and the stored integer is interpreted as 00111100, or 60.

In the bottom block diagram, the Data Type Conversion3 block treats the input as a stored integer, and the data type and scaling information is not applied. When the value is then output from the Data Type Conversion4 block as a real-world value, the scaling and data type information is applied to the stored integer (Slope = 2^{-2} and Bias = 0), and the output value is 000011.11, or 3.75. When the value is output from the Data Type Conversion5 block as a stored integer, you get back the original input value of 15.

Real World Values and Stored Integers in Summations

The following model shows how a summation operation applies to real-world values and stored integers, and how the generated code handles scaling information.


Convert
double
Data Type Conversion3
15
Display4
Convert
double
Data Type Conversion2
0.9375
Display3
+
+
double
Sum1
15.94
Display2
15
int16
Constant1
Convert
Data Type Conversion1
sfix16_En2
+
+
Sum
sfix16_En5
Convert
double
Data Type Conversion5
15.94
Display1
15
int16
Constant
Convert
(SI)
Data Type Conversion
sfix16_En4
Convert
(SI)
double
Data Type Conversion4
510
Display

Note that the summation operation produces the correct result when the Data Type Conversion (2 or 5) block outputs a real-world value. This is because the specified scaling information is applied to the stored integer value. However, when the Data Type Conversion4 block outputs a stored integer value, then the summation operation produces an unexpected result due to the absence of scaling information.

If you generate code for the above model, then the code captures the appropriate scaling information. The code for the Sum block is shown below. The inputs to this block are tagged with the specified scaling information so that the necessary shifts are performed for the summation operation.

```
/* Sum Block: <Root>/Sum
   *
   *  y =  u0 + u1
   *
   * Input0  Data Type:  Fixed Point    S16  2^-2
   * Input1  Data Type:  Fixed Point    S16  2^-4
   * Output0 Data Type:  Fixed Point    S16  2^-5
   *
   * Round Mode: Floor
   * Saturation Mode: Wrap
   *
   */
  sum = ((in1) << 3);
  sum += ((in2) << 1);
```

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Data Type Conversion Inherited

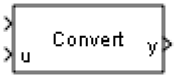
Purpose

Convert from one data type to another using inherited data type and scaling

Library

Signal Attributes

Description



The Data Type Conversion Inherited block forces dissimilar data types to be the same. The first input is used as the reference signal and the second input is converted to the reference type by inheriting the data type and scaling information. (See "How to Rotate a Block" in the Simulink documentation for a description of the port order for various block orientations.) Either input undergoes scalar expansion such that the output has the same width as the widest input.

Inheriting the data type and scaling provides these advantages:

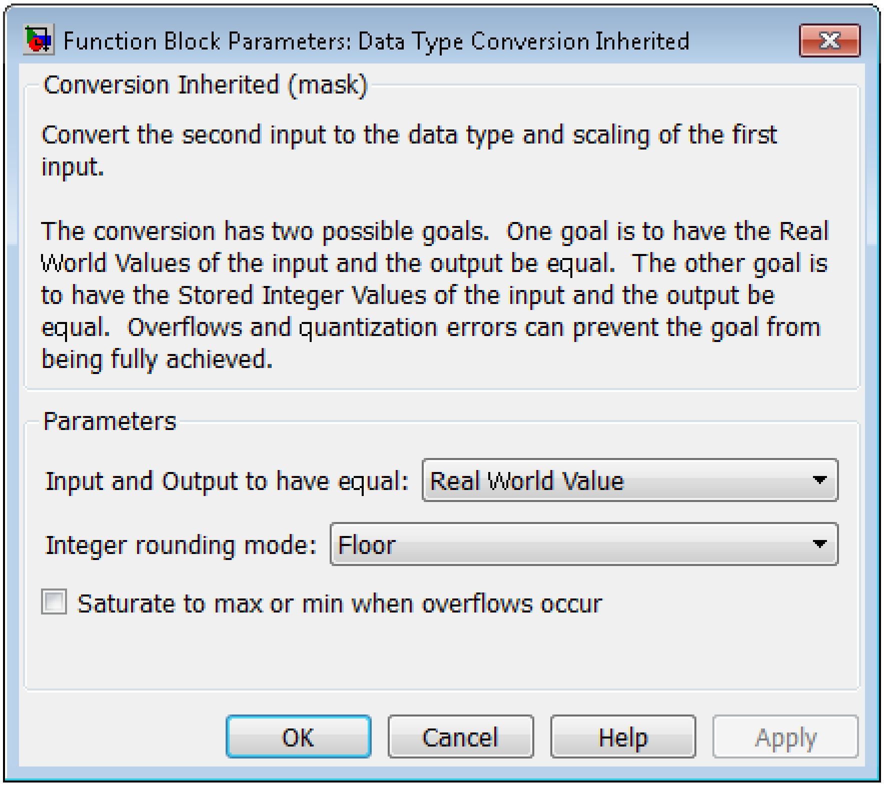
- It makes reusing existing models easier.
- It allows you to create new fixed-point models with less effort since you can avoid the detail of specifying the associated parameters.

Data Type Support

The Data Type Conversion Inherited block accepts signals of the following data types:

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

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



Function Block Parameters: Data Type Conversion Inherited

Conversion Inherited (mask)

Convert the second input to the data type and scaling of the first input.

The conversion has two possible goals. One goal is to have the Real World Values of the input and the output be equal. The other goal is to have the Stored Integer Values of the input and the output be equal. Overflows and quantization errors can prevent the goal from being fully achieved.

Parameters

Input and Output to have equal: Real World Value

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Parameters and Dialog Box

Input and Output to have equal

Specify whether the `Real World Value (RWV)` or the `Stored Integer (SI)` of the input and output should be the same. Refer to Description in the Data Type Conversion block reference page for more information about these choices.

Integer rounding mode

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

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

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

Characteristics

Direct Feedthrough	Yes

See Also

Data Type Conversion

Data Type Duplicate

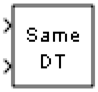
Purpose

Force all inputs to same data type

Library

Signal Attributes

Description



The Data Type Duplicate block forces all inputs to have exactly the same data type. Other attributes of input signals, such as dimension, complexity, and sample time, are completely independent.

You can use the Data Type Duplicate block to check for consistency of data types among blocks. If all signals do not have the same data type, the block returns an error message.

The Data Type Duplicate block is typically used such that one signal to the block controls the data type for all other blocks. The other blocks are set to inherit their data types via back propagation.

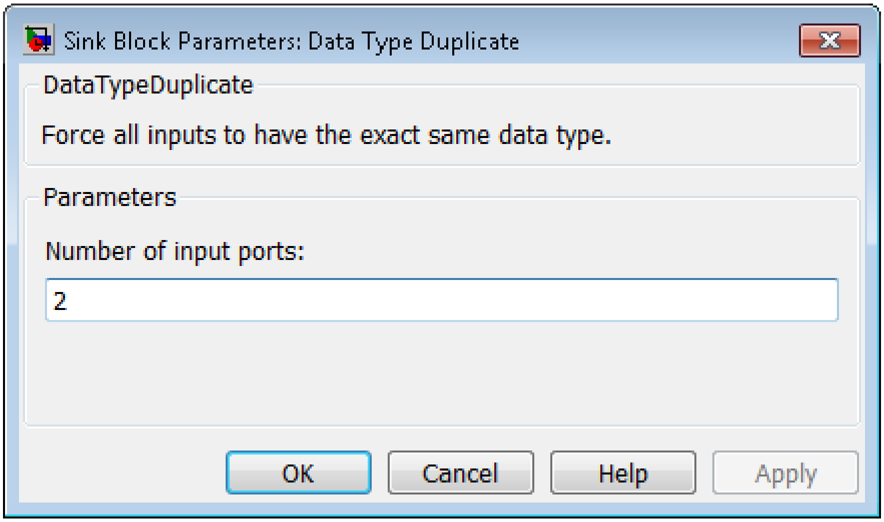
The block is also used in a user created library. These library blocks can be placed in any model, and the data type for all library blocks are configured according to the usage in the model. To create a library block with more complex data type rules than duplication, use the Data Type Propagation block.

Data Type Support

The Data Type Duplicate block accepts signals of the following data types:

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

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



Sink Block Parameters: Data Type Duplicate

DataTypeDuplicate

Force all inputs to have the exact same data type.

Parameters

Number of input ports: 2

Parameters and Dialog Box

Number of input ports

Specify the number of inputs to this block.

Characteristics

Sample Time	Inherited from the driving block
Scalar Expansion	Yes
States	0
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Data Type Propagation

Data Type Propagation

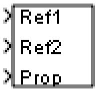
Purpose

Set data type and scaling of propagated signal based on information from reference signals

Library

Signal Attributes

Description



The Data Type Propagation block allows you to control the data type and scaling of signals in your model. You can use this block in conjunction with fixed-point blocks that have their **Output data type** parameter configured to `Inherit:  Inherit via back propagation`.

The block has three inputs: Ref1 and Ref2 are the reference inputs, while the Prop input back propagates the data type and scaling information gathered from the reference inputs. This information is then passed on to other fixed-point blocks.

The block provides you with many choices for propagating data type and scaling information. For example, you can:

- Use the number of bits from the Ref1 reference signal, or use the number of bits from widest reference signal.
- Use the range from the Ref2 reference signal, or use the range of the reference signal with the greatest range.
- Use a bias of zero, regardless of the biases used by the reference signals.
- Use the precision of the reference signal with the least precision.

You specify how data type information is propagated with the **Propagated data type** parameter list. If the parameter list is configured as `Specify via dialog`, then you manually specify the data type via the **Propagated data type** edit field. If the parameter list is configured as `Inherit via propagation rule`, then you must use the parameters described in “Parameters and Dialog Box” on page 1-265.

You specify how scaling information is propagated with the **Propagated scaling** parameter list. If the parameter list is configured as `Specify via dialog`, then you manually specify the scaling via the **Propagated scaling** edit field. If the parameter list is configured as `Inherit via` `propagation rule`, then you must use the parameters described in "Parameters and Dialog Box" on page 1-265.

After you use the information from the reference signals, you can apply a second level of adjustments to the data type and scaling by using individual multiplicative and additive adjustments. This flexibility has a variety of uses. For example, if you are targeting a DSP, then you can configure the block so that the number of bits associated with a MAC (multiply and accumulate) operation is twice as wide as the input signal, and has a certain number of guard bits added to it.

The Data Type Propagation block also provides a mechanism to force the computed number of bits to a useful value. For example, if you are targeting a 16-bit micro, then the target C compiler is likely to support sizes of only 8 bits, 16 bits, and 32 bits. The block will force these three choices to be used. For example, suppose the block computes a data type size of 24 bits. Since 24 bits is not directly usable by the target chip, the signal is forced up to 32 bits, which is natively supported.

There is also a method for dealing with floating-point reference signals. This makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or vice versa.

The Data Type Propagation block allows you to set up libraries of useful subsystems that will be properly configured based on the connected signals. Without this data type propagation process, a subsystem that you use from a library will almost certainly not work as desired with most integer or fixed-point signals, and manual intervention to configure the data type and scaling would be required. This block can eliminate the manual intervention in many situations.

Precedence Rules

The precedence of the dialog box parameters decreases from top to bottom. Additionally:

- Double-precision reference inputs have precedence over all other data types.

- Single-precision reference inputs have precedence over integer and fixed-point data types.
- Multiplicative adjustments are carried out before additive adjustments.
- The number of bits is determined before the precision or positive range is inherited from the reference inputs.

Data Type Support

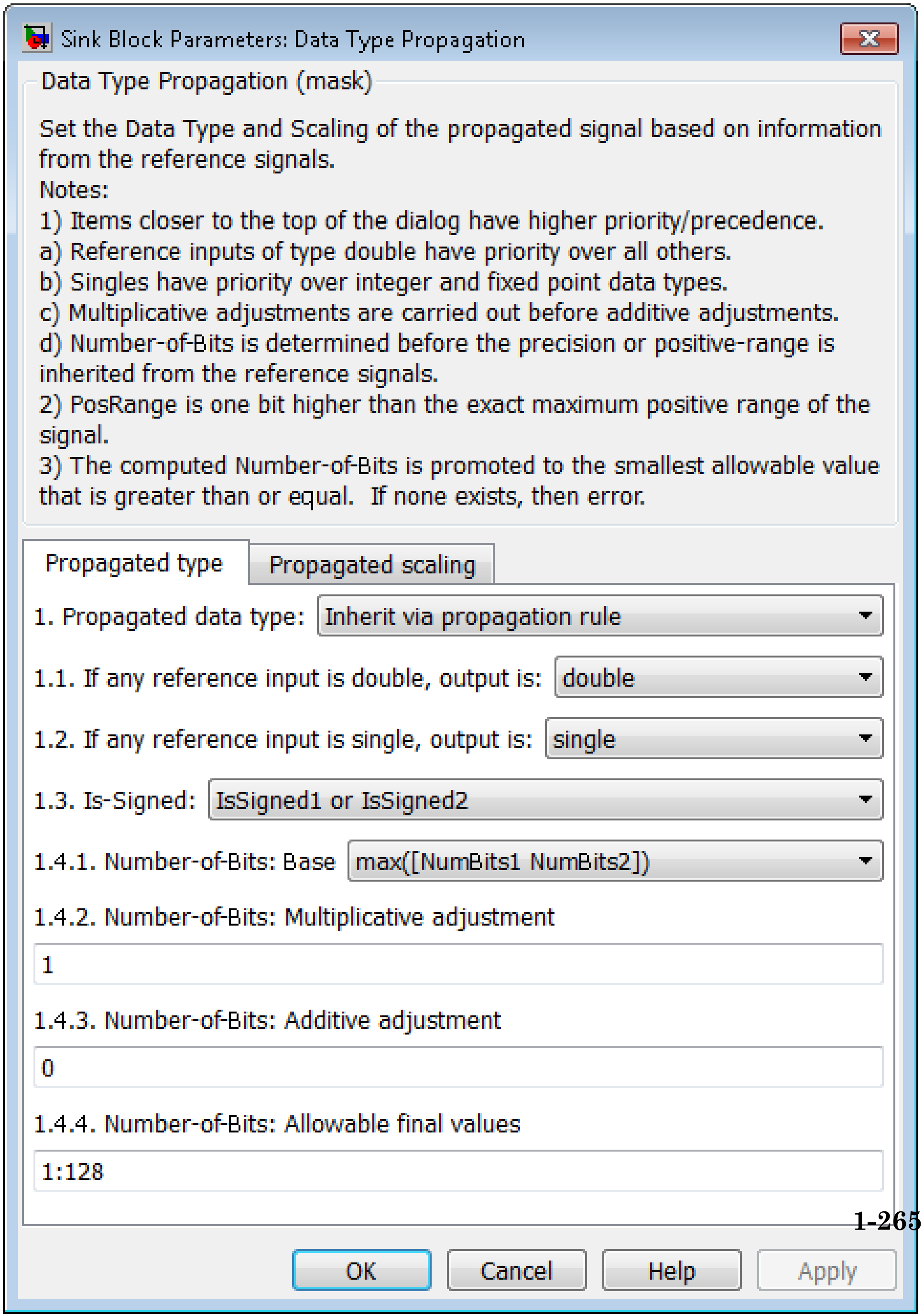
The Data Type Propagation block accepts signals of the following data types:

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

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

Parameters and Dialog Box

The **Propagated type** pane of the Data Type Propagation block dialog box appears as follows:



Data Type Propagation (mask)

Set the Data Type and Scaling of the propagated signal based on information from the reference signals.
Notes:
1) Items closer to the top of the dialog have higher priority/precedence.
a) Reference inputs of type double have priority over all others.
b) Singles have priority over integer and fixed point data types.
c) Multiplicative adjustments are carried out before additive adjustments.
d) Number-of-Bits is determined before the precision or positive-range is inherited from the reference signals.
2) PosRange is one bit higher than the exact maximum positive range of the signal.
3) The computed Number-of-Bits is promoted to the smallest allowable value that is greater than or equal. If none exists, then error.

1. Propagated data type: Inherit via propagation rule

1.1. If any reference input is double, output is: double

1.2. If any reference input is single, output is: single

1.3. Is-Signed: IsSigned1 or IsSigned2

1.4.1. Number-of-Bits: Base max([NumBits1 NumBits2])

1.4.2. Number-of-Bits: Multiplicative adjustment
1

1.4.3. Number-of-Bits: Additive adjustment
0

1.4.4. Number-of-Bits: Allowable final values
1:128

If any reference input is double, output is

Specify `single` or `double`. This parameter makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or vice versa.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

If any reference input is single, output is

Specify `single` or `double`. This parameter makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or visa versa.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Is-Signed

Specify the sign of Prop as one of the following values:

Parameter Value	Description
`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 **Is-Signed** parameter is `IsSigned1 or IsSigned2`, then Prop is forced to be a signed data type.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Number-of-bits: Base

Specify the number of bits used by Prop for the base data type as one of the following values:

Parameter Value	Description
`NumBits1`	The number of bits for Prop is given by the number of bits for Ref1.
`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 in the Simulink Fixed Point™ documentation.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Number-of-bits: Multiplicative adjustment

Specify the number of bits used by Prop by including a multiplicative adjustment. For example, suppose you want to guarantee that the number of bits associated with a multiply and accumulate (MAC) operation is twice as wide as the input signal. To do this, you configure this parameter to the value `2`.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Number-of-bits: Additive adjustment

Specify the number of bits used by Prop by including an additive adjustment. For example, if you are performing multiple additions during a MAC operation, the result might overflow. To prevent overflow, you can associate guard bits with the propagated data type. To associate four guard bits, you specify the value `4`.

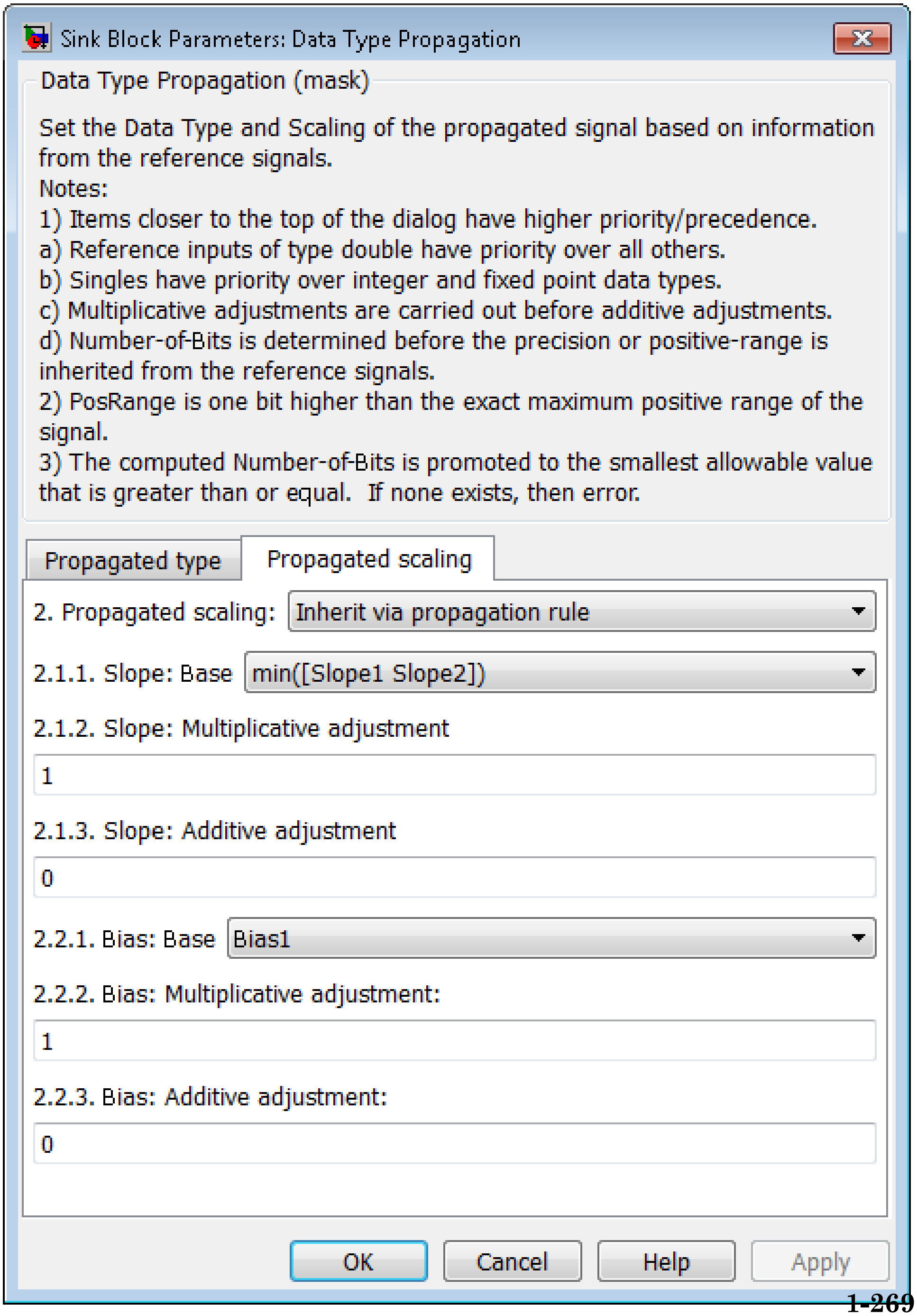
This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Number-of-bits: Allowable final values

Force the computed number of bits used by Prop to a useful value. For example, if you are targeting a processor that supports only 8, 16, and 32 bits, then you configure this parameter to `[8,16,32]`. The block always propagates the smallest specified value that fits. If you want to allow all fixed-point data types, you would specify the value `1:128`.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

The **Propagated scaling** pane of the Data Type Propagation block dialog box appears as follows:



Data Type Propagation (mask)

Set the Data Type and Scaling of the propagated signal based on information from the reference signals.

Notes:

1) Items closer to the top of the dialog have higher priority/precedence.

a) Reference inputs of type double have priority over all others.

b) Singles have priority over integer and fixed point data types.

c) Multiplicative adjustments are carried out before additive adjustments.

d) Number-of-Bits is determined before the precision or positive-range is inherited from the reference signals.

2) PosRange is one bit higher than the exact maximum positive range of the signal.

3) The computed Number-of-Bits is promoted to the smallest allowable value that is greater than or equal. If none exists, then error.

2. Propagated scaling: Inherit via propagation rule

2.1.1. Slope: Base min([Slope1 Slope2])

2.1.2. Slope: Multiplicative adjustment

1

2.1.3. Slope: Additive adjustment

0

2.2.1. Bias: Base Bias1

2.2.2. Bias: Multiplicative adjustment:

1

2.2.3. Bias: Additive adjustment:

0

Propagated scaling

Use the parameter list to propagate the scaling via the dialog box, inherit the scaling from the reference signals, or calculate the scaling to obtain best precision.

Propagated scaling (Slope or [Slope Bias])

Specify the scaling as either a slope or a slope and bias.

This parameter is visible only when you set **Propagated scaling** to `Specify via dialog`.

Values used to determine best precision scaling

Specify any values to be used to constrain the precision, such as the upper and lower limits on the propagated input. Based on the data type, the scaling will automatically be selected such that these values can be represented with no overflow error and minimum quantization error.

This parameter is visible only when you set **Propagated scaling** to `Obtain via best precision`.

Slope: Base

Specify the slope used by Prop for the base data type as one of the following values:

Parameter Value	Description
`Slope1`	The slope of Prop is given by the slope of Ref1.
`Slope2`	The slope of Prop is given by the slope of Ref2.
`max([Slope1 Slope2])`	The slope of Prop is given by the maximum slope of the reference signals.
`min([Slope1 Slope2])`	The slope of Prop is given by the minimum slope of the reference signals.

Parameter Value	Description
`Slope1*Slope2`	The slope of Prop is given by the product of the reference signal slopes.
`Slope1/Slope2`	The slope of Prop is given by the ratio of the Ref1 slope to the Ref2 slope.
`PosRange1`	The range of Prop is given by the range of Ref1.
`PosRange2`	The range of Prop is given by the range of Ref2.
`max([PosRange1 PosRange2])`	The range of Prop is given by the maximum range of the reference signals.
`min([PosRange1 PosRange2])`	The range of Prop is given by the minimum range of the reference signals.
`PosRange1*PosRange2`	The range of Prop is given by the product of the reference signal ranges.
`PosRange1/PosRange2`	The range of Prop is given by the ratio of the Ref1 range to the Ref2 range.

You control the precision of Prop with `Slope1` and `Slope2`, and you control the range of Prop with `PosRange1` and `PosRange2`. Additionally, `PosRange1` and `PosRange2` are one bit higher than the maximum positive range of the associated reference signal.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Slope: Multiplicative adjustment

Specify the slope used by Prop by including a multiplicative adjustment. For example, if you want 3 bits of additional precision (with a corresponding decrease in range), the multiplicative adjustment is `2^-3`.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Slope: Additive adjustment

Specify the slope used by Prop by including an additive adjustment. An additive slope adjustment is often not needed. The most likely use is to set the multiplicative adjustment to `0`, and set the additive adjustment to force the final slope to a specified value.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Bias: Base

Specify the bias used by Prop for the base data type. The parameter values are described as follows:

Parameter Value	Description
`Bias1`	The bias of Prop is given by the bias of Ref1.
`Bias2`	The bias of Prop is given by the bias of Ref2.
`max([Bias1 Bias2])`	The bias of Prop is given by the maximum bias of the reference signals.
`min([Bias1 Bias2])`	The bias of Prop is given by the minimum bias of the reference signals.
`Bias1*Bias2`	The bias of Prop is given by the product of the reference signal biases.
`Bias1/Bias2`	The bias of Prop is given by the ratio of the Ref1 bias to the Ref2 bias.
`Bias1+Bias2`	The bias of Prop is given by the sum of the reference biases.
`Bias1-Bias2`	The bias of Prop is given by the difference of the reference biases.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Bias: Multiplicative adjustment

Specify the bias used by Prop by including a multiplicative adjustment.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Bias: Additive adjustment

Specify the bias used by Prop by including an additive adjustment.

If you want to guarantee that the bias associated with Prop is zero, you should configure both the multiplicative adjustment and the additive adjustment to `0`.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Data Type Duplicate

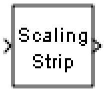
Purpose

Remove scaling and map to built in integer

Library

Signal Attributes

Description



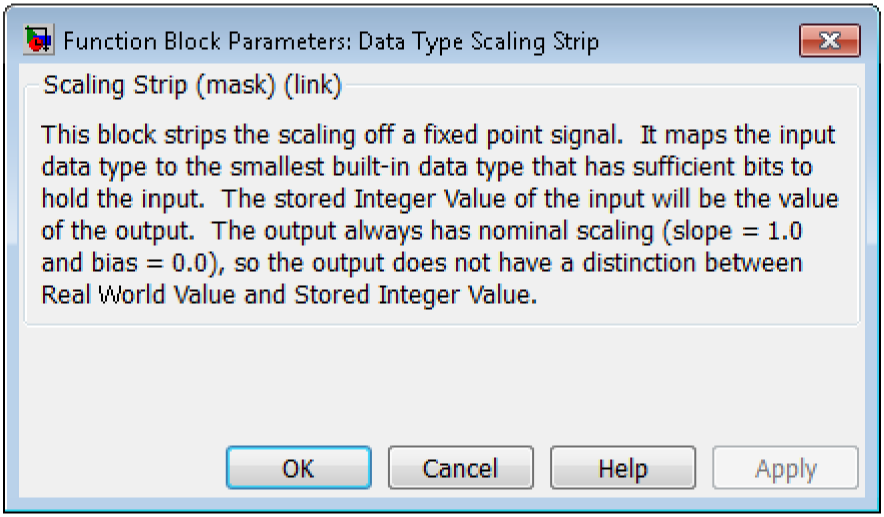
The Scaling Strip block strips the scaling off a fixed point signal. It maps the input data type to the smallest built in data type that has enough data bits to hold the input. The stored integer value of the input is the value of the output. The output always has nominal scaling (slope = 1.0 and bias = 0.0), so the output does not make a distinction between real world value and stored integer value.

Data Type Support

The Data Type Scaling Strip block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

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

Parameters and Dialog Box


Function Block Parameters: Data Type Scaling Strip
Scaling Strip (mask) (link)
OK
Cancel
Help
Apply

This block strips the scaling off a fixed point signal. It maps the input data type to the smallest built-in data type that has sufficient bits to hold the input. The stored Integer Value of the input will be the value of the output. The output always has nominal scaling (slope = 1.0 and bias = 0.0), so the output does not have a distinction between Real World Value and Stored Integer Value.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Dead Zone

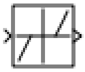
Purpose

Provide region of zero output

Library

Discontinuities

Description



The Dead Zone block generates zero output within a specified region, called its dead zone. You specify the lower limit (`LL`) and upper limit (`UL`) of the dead zone as the **Start of dead zone** and **End of dead zone** parameters, respectively. The block output depends on the input (`U`) and the values for the lower and upper limits:

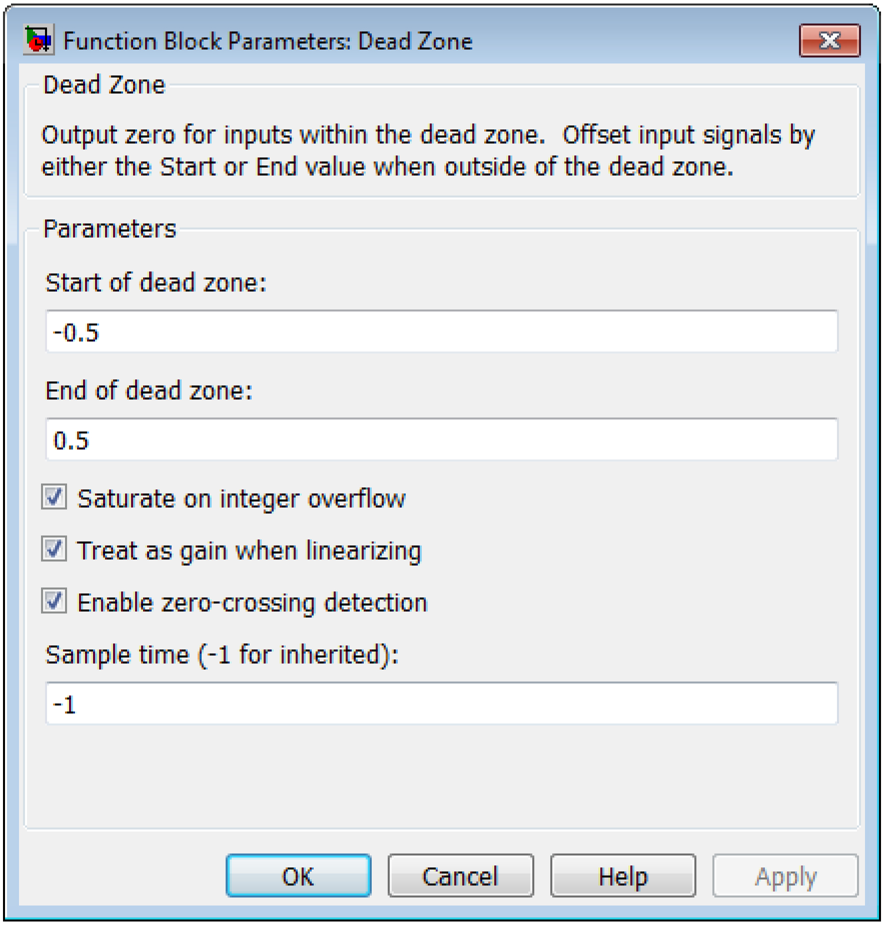
Input	Output
`U >= LL` and `U <= UL`	Zero
`U > UL`	`U – UL`
`U < LL`	`U – LL`

Data Type Support

The Dead Zone block accepts and outputs real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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



Function Block Parameters: Dead Zone

Dead Zone

Output zero for inputs within the dead zone. Offset input signals by either the Start or End value when outside of the dead zone.

Parameters

Start of dead zone:

-0.5

End of dead zone:

0.5

☑ Saturate on integer overflow

☑ Treat as gain when linearizing

☑ Enable zero-crossing detection

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Start of dead zone
Specify the lower limit of the dead zone. The default is `-0.5`.

End of dead zone
Specify the upper limit of the dead zone. The default is `0.5`.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 “Checking 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.

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

Treat as gain when linearizing

The linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the commands to treat the gain as 1; otherwise, the commands treat the gain as 0.

Enable zero-crossing detection

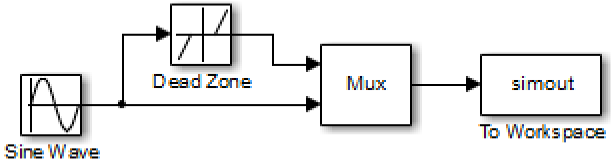
Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time (-1 for inherited)

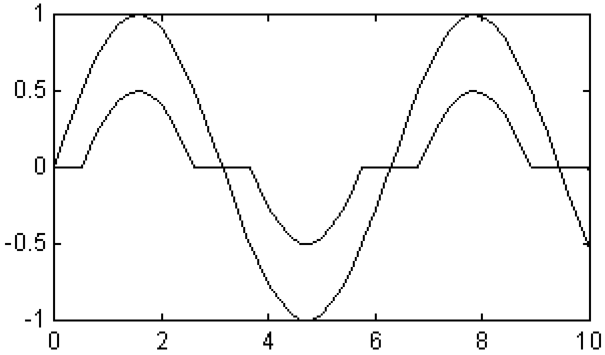
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See Specifying Sample Time in the Simulink documentation.

Examples

The following model uses lower and upper limits of -0.5 and 0.5, with a sine wave as input.



This plot shows the effect of the Dead Zone block on the sine wave. When the input sine wave is between –0.5 and 0.5, the output is zero.



Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of parameters

Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled

See Also

Dead Zone Dynamic

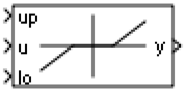
Purpose

Set inputs within bounds to zero

Library

Discontinuities

Description



The Dead Zone Dynamic block dynamically bounds the range of the input signal, providing a region of zero output. The bounds change according to the upper and lower limit input signals where

- The input within the bounds is set to zero.
- The input below the lower limit is shifted down by the lower limit.
- The input above the upper limit is shifted down by the upper limit.

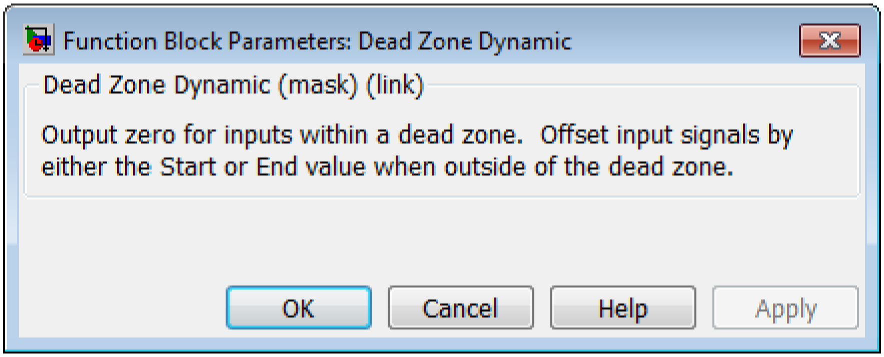
The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

Data Type Support

The Dead Zone Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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


Function Block Parameters: Dead Zone Dynamic

Dead Zone Dynamic (mask) (link)

Output zero for inputs within a dead zone. Offset input signals by either the Start or End value when outside of the dead zone.

Parameters and Dialog Box

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Dead Zone

Decrement Real World

Purpose

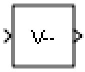
Decrease real world value of signal by one

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Real World block decreases the real world value of the signal by one. Overflows always wrap.



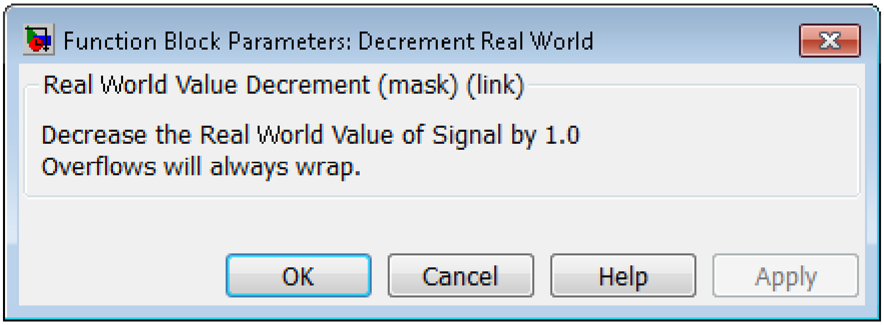
Data Type Support

The Decrement Real World block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Function Block Parameters: Decrement Real World

Real World Value Decrement (mask) (link)

Decrease the Real World Value of Signal by 1.0
Overflows will always wrap.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Decrement Stored Integer, Decrement Time To Zero, Decrement To Zero, Increment Real World

Purpose

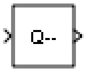
Decrease stored integer value of signal by one

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Stored Integer block decreases the stored integer value of a signal by one.



Floating-point signals also decrease by one, and overflows always wrap.

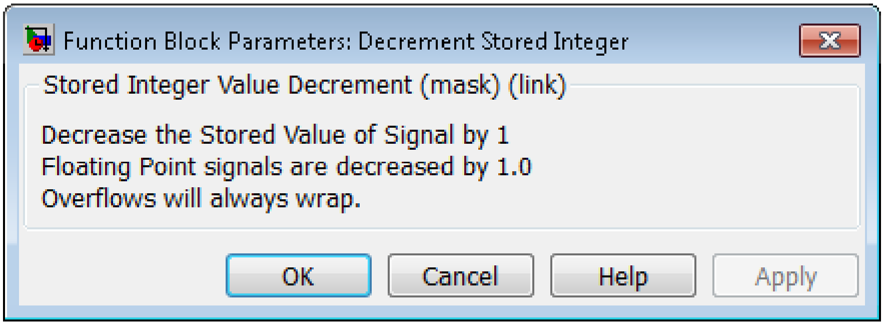
Data Type Support

The Decrement Stored Integer block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box


Function Block Parameters: Decrement Stored Integer

Stored Integer Value Decrement (mask) (link)

Decrease the Stored Value of Signal by 1
Floating Point signals are decreased by 1.0
Overflows will always wrap.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Decrement Real World, Decrement Time To Zero, Decrement To Zero, Increment Stored Integer

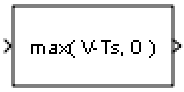
Purpose

Decrease real-world value of signal by sample time, but only to zero

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description



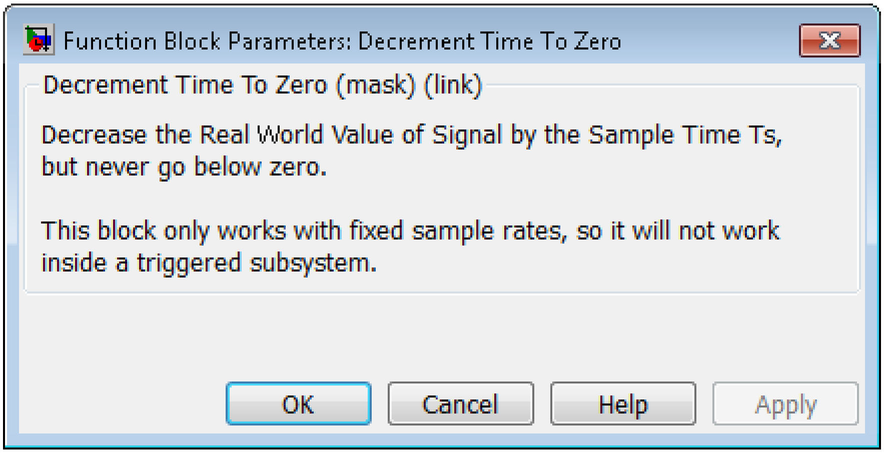
The Decrement Time To Zero block decreases the real-world value of the signal by the sample time, `Ts`. The output never goes below zero. This block works only with fixed sample rates and does not work inside a triggered subsystem.

Data Type Support

The Decrement Time To Zero block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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



Function Block Parameters: Decrement Time To Zero

Decrement Time To Zero (mask) (link)

Decrease the Real World Value of Signal by the Sample Time Ts, but never go below zero.

This block only works with fixed sample rates, so it will not work inside a triggered subsystem.

Parameters and Dialog Box

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Decrement Real World, Decrement Stored Integer, Decrement To Zero

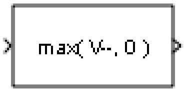
Purpose

Decrease real-world value of signal by one, but only to zero

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description



The Decrement To Zero block decreases the real-world value of the signal by one. The output never goes below zero.

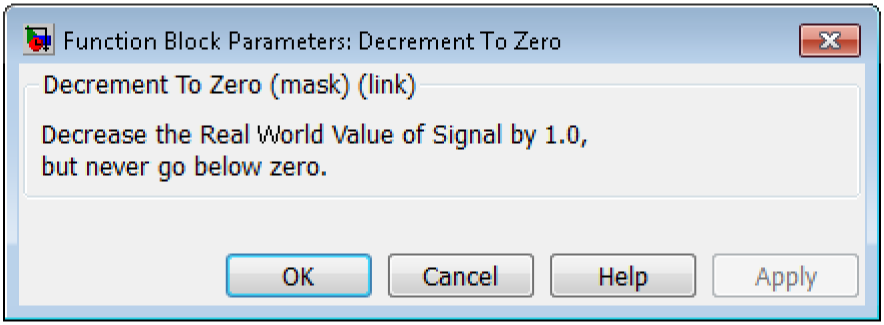
Data Type Support

The Decrement To Zero block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Function Block Parameters: Decrement To Zero

Decrement To Zero (mask) (link)

Decrease the Real World Value of Signal by 1.0,
but never go below zero.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Decrement Real World, Decrement Stored Integer, Decrement Time To Zero

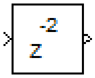
Purpose

Delay input signal by fixed or variable sample periods

Library

Discrete

Description



Supported Block Operations

The Delay block supports:

- Variable delay length
- Specification of the initial condition from an input port
- Resetting the state to the initial condition with an external reset signal
- State storage
- Using a circular buffer instead of an array buffer for state storage

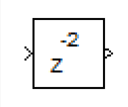
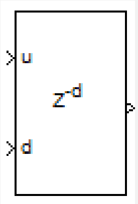
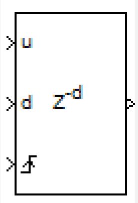
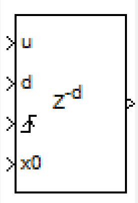
The Delay block delays an input `u` according to the **Delay length** parameter, which you specify on the dialog box, or a delay length that a signal supplies to the `d` input port. The initial block output depends on the **Initial condition** parameter, which you specify on the dialog box, or an initial condition that a signal supplies to the `x0` input port. Whether or not the block output resets to the initial condition depends on the **External reset** parameter, which you specify on the dialog box.

The following dimensional requirements apply:

- **Delay length** and **External reset** must be scalar.
- **Initial condition** can be scalar or nonscalar.
- For frame-based processing, signal dimensions of the data input port `u` cannot be larger than two.

Block Icon Appearance

The icon changes depending on the number of block inputs.

Icon	Number of Inputs
	One: - Input signal to delay, u
	Two: - Input signal to delay, u - Delay length, d
	Three: - Input signal to delay, u - Delay length, d - External reset
	Four: - Input signal to delay, u - Delay length, d - External reset - Initial condition, x0

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 Boolean Enumerated
`d`	Floating point Fixed-point integer
`x0`	Floating point Built-in integer Fixed point Boolean Enumerated

When `u` is Boolean, `x0` must be Boolean. When `u` uses an enumerated type, `x0` must use the same enumerated type. Otherwise, `x0` can use a floating-point, built-in integer, or fixed-point data type that fits in the data type of `u`. For example, when `u` uses `int32`, `x0` can use `int8` but not `double`.

The data type of the output signal is the same as the input signal `u`.

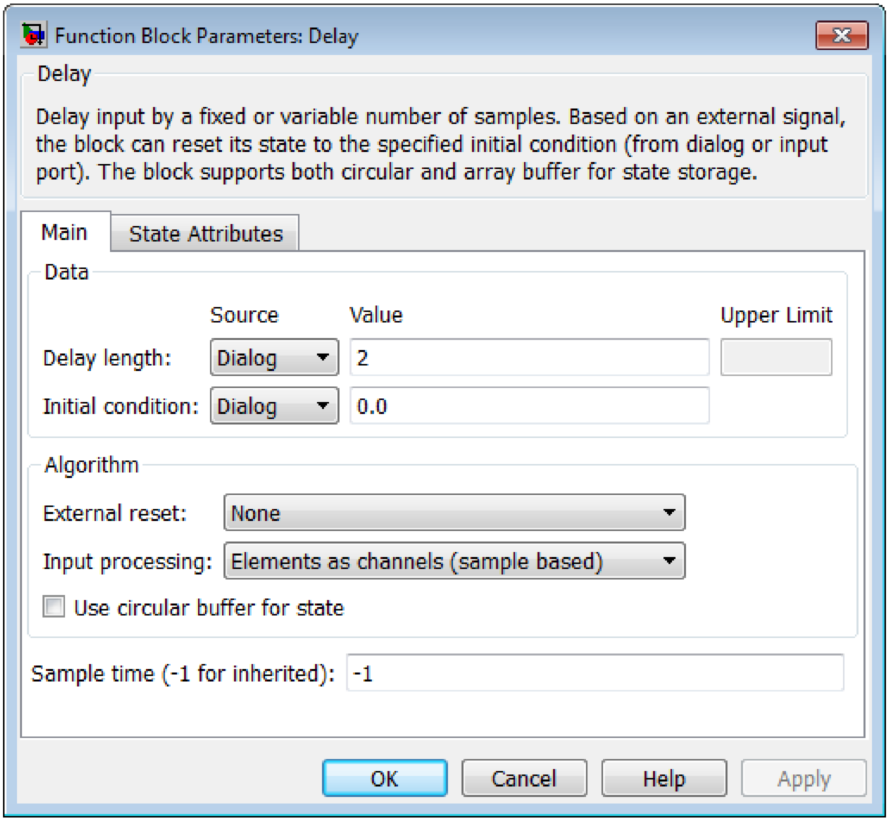
When you specify **Delay length** and **Initial condition** on the dialog box, you can use the following data types for those parameters:

Parameter	Supported Data Types
Delay length	• Floating point • Fixed-point integer
Initial condition	• Floating point • Built-in integer • Fixed point • Boolean • Enumerated

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

Parameters and Dialog Box

The **Main** pane of the Delay block dialog box appears as follows:



Function Block Parameters: Delay
Main
State Attributes
Data
Source
Value
Upper Limit
Algorithm
OK
Cancel
Help
Apply

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.

Delay length: Dialog 2

Initial condition: Dialog 0.0

External reset: None

Input processing: Elements as channels (sample based)

☐ Use circular buffer for state

Sample time (-1 for inherited): -1

Delay length

Specify whether to enter the delay length directly on the dialog box (fixed delay) or to inherit the delay from an input port (variable delay).

- If you set **Source** to `Dialog`, enter the delay length in the edit field under **Value**.

- If you set **Source** to `Input port`, verify that an upstream signal supplies a delay length for the `d` input port. You can also specify an **Upper limit** for the delay length.

The scalar delay length (from the dialog parameter or an input port) must be a real, positive integer.

Initial condition

Specify whether to enter the initial condition directly on the dialog box or to inherit the initial condition from an input port.

- If you set **Source** to `Dialog`, enter the initial condition in the edit field under **Value**.
- If you set **Source** to `Input port`, verify that an upstream signal supplies an initial condition for the `x0` input port.

Simulink converts offline the data type of **Initial condition** to the data type of the input signal `u` using a round-to-nearest operation and saturation.

Note When **State name must resolve to Simulink signal object** is selected on the **State Attributes** pane, the block copies the initial value of the signal object to the **Initial condition** parameter. However, when the source for **Initial condition** is `Input port`, the block ignores the initial value of the signal object.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
`None`	No reset.
`Rising`	Reset on a rising edge.
`Falling`	Reset on a falling edge.

Reset Mode	Behavior
`Either`	Reset on either a rising or falling edge.
`Level`	Reset in either of these cases: • when there is a nonzero at the current time step • when the time step value changes from nonzero at the previous time step to zero at the current time step
`Level hold`	Reset when nonzero at the current time step.

The reset signal must be scalar.

Input processing

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

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

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

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

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

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

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

Use circular buffer for state

Select to use a circular buffer for storing the state in simulation and code generation. Otherwise, an array buffer stores the state.

Using a circular buffer can improve execution speed when the delay length is large. For an array buffer, the number of copy operations increases as the delay length goes up. For a circular buffer, the number of copy operations is constant for increasing delay length.

If one of the following conditions is true, an array buffer always stores the state because a circular buffer does not improve execution speed:

- For sample-based signals, the delay length is 1.
- For frame-based signals, the delay length is no larger than the frame size.

Prevent direct feedthrough by increasing delay length to lower limit

Select to increase the delay length from zero to the lower limit for the **Input processing** mode:

- For sample-based signals, increase the minimum delay length to 1.
- For frame-based signals, increase the minimum delay length to the frame length.

Selecting this check box prevents direct feedthrough from the input port, `u`, to the output port. However, this check box cannot prevent direct feedthrough from the initial condition port, `x0`, to the output port.

This check box is available when you set **Delay length: Source** to `Input port`.

Remove protection against out-of-range delay length in generated code

Select to remove code that checks for out-of-range delay length.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range delay length.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range delay length.	For safety-critical applications

This check box is available when you set **Delay length: Source** to `Input port`.

Diagnostic for out-of-range delay length

Specify whether to produce a warning or error when the input `d` is less than the lower limit or greater than the **Delay length: Upper limit**. The lower limit depends on the setting for **Prevent direct feedthrough by increasing delay length to lower limit**.

- 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` — No warning or error appears.
- `Warning` — Display a warning in the MATLAB Command Window and continue the simulation.
- `Error` — Stop the simulation and display an error in the Simulation Diagnostics Viewer.

This parameter is available when you set **Delay length: Source** to `Input port`.

Sample time (-1 for inherited)

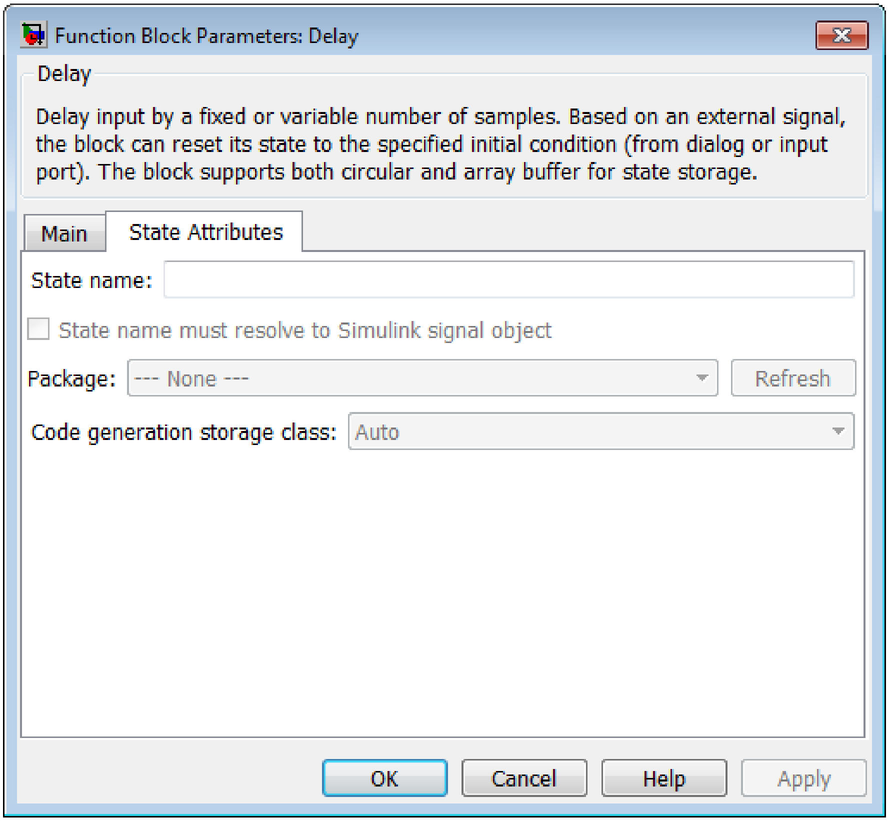
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. This block supports discrete sample time, but not continuous sample time.

As described in "Specify Sample Time", you can specify a discrete sample time as `[Tsampling, Toffset]`, where *`Tsampling`* is the sampling period and *`Toffset`* is the initial time offset. If *`Tstart`* is the simulation start time and *`n`* is the delay length, the Delay block output is as follows.

Simulation Time Range	Block Output
(*`Tstart`*) to (*`Tstart`* + *`Toffset`*)	Zero
(*`Tstart`* + *`Toffset`*) to (*`Tstart`* + *`Toffset`* + *`n`* * *`Tsampling`*)	Initial condition
After (*`Tstart`* + *`Toffset`* + *`n`* * *`Tsampling`*)	Input signal

If the Delay block inherits or specifies a nonzero value for *`Toffset`*, the Delay block outputs zero at *`Tstart`*. Otherwise, the Delay block outputs the initial condition at *`Tstart`*.

The **State Attributes** pane of the Delay block dialog box appears as follows:



Function Block Parameters: Delay
Main
State Attributes
Refresh
OK
Cancel
Help
Apply

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.

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Code generation storage class: Auto

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “States” in the Simulink Coder documentation.

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Package

Select a package that defines the custom storage class you want to apply. If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Code generation storage class

From the list, select a state storage class.

`Auto`

Auto is the storage class to use for states that do not need to interface to external code.

`ExportedGlobal`

The state is stored in a global variable.

`ImportedExtern`

`*model*_private.h` declares the state as an extern variable.

`ImportedExternPointer`

`*model*_private.h` declares the state as an extern pointer.

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Code generation storage type qualifier

Specify a Simulink Coder storage type qualifier. The default is `' '`. When this field is blank, no qualifier is assigned.

The Simulink Coder product does not check this string for errors. Thus, whatever value you enter appears automatically in the variable declaration.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

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

Input Processing Mode	Rules for Variable-Size Signal 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

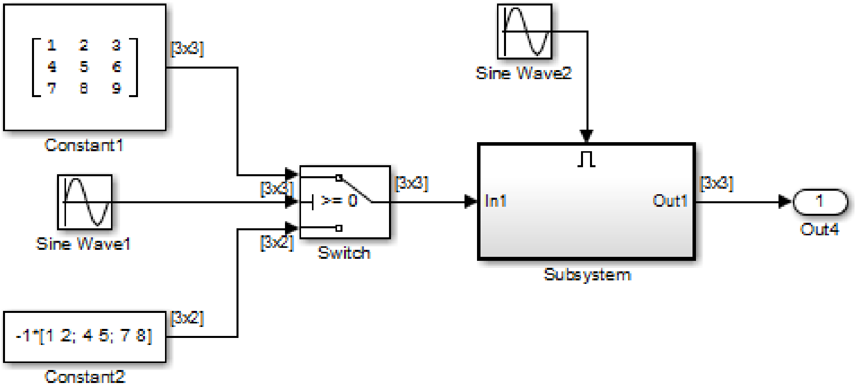
To use a bus signal as the input to a Delay block, you must specify the initial condition on the dialog box. In other words, the initial condition cannot come from the input port `x0`. Support for virtual and nonvirtual buses depends on the initial condition that you specify and whether the **State name** 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

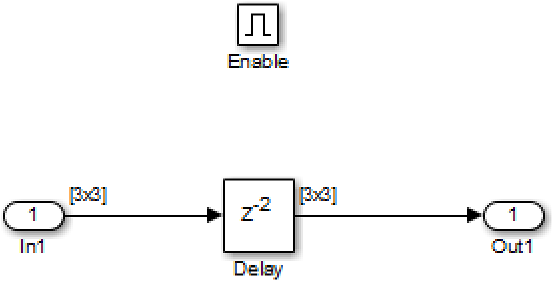
Examples

Variable-Size Signals for Sample-Based Processing

This model shows how the Delay block supports variable-size signals for sample-based processing.


1 2 3
4 5 6
7 8 9
[3x3]
Constant1
Sine Wave1
[3x3]
>= 0
[3x2]
Switch
[3x3]
-1*[1 2; 4 5; 7 8]
[3x2]
Constant2
Sine Wave2
In1
Out1
Subsystem
[3x3]
1
Out4

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.

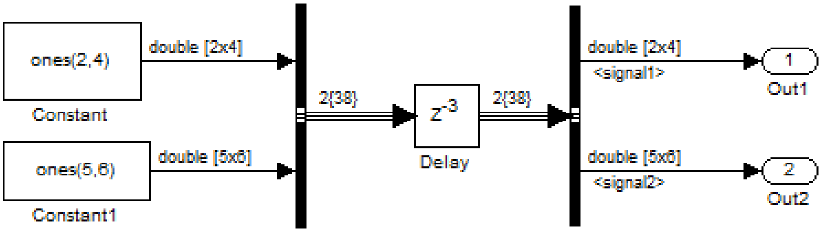

Enable
1
In1
[3x3]
z-2
Delay
[3x3]
1
Out1

The model follows the rules for variable-size signals when the Delay block uses sample-based processing.

Rule	How the Model Follows the Rule
The signal dimensions change only during state reset when the block is enabled.	The Enable block sets **Propagate sizes of variable-size signals** to `Only when enabling`.
The initial condition must be scalar.	The Delay block sets **Initial condition** to `0.0`, a scalar value.

Bus Signals for Frame-Based Processing

This model shows how the Delay block supports bus signals for frame-based processing.


ones(2,4)
Constant
double [2x4]
ones(5,6)
Constant1
double [5x6]
2{38}
z^{-3}
Delay
2{38}
double [2x4]
<signal1>
1
Out1
double [5x6]
<signal2>
2
Out2

Each Constant block supplies an input signal to the Bus Creator block, which outputs a two-dimensional bus signal. After the Delay block delays the bus signal by three sample periods, the Bus Selector block separates the bus back into the two original signals.

The model follows the rules for bus signals when the Delay block uses frame-based processing.

Rule	How the Model Follows the Rule
For the initial condition, set the value on the dialog box.	The Delay block sets **Initial condition** to `0`, a scalar value.
For frame-based processing, signal dimensions of the data input port `u` cannot be larger than two. (This rule applies to all inputs for the port `u`, not just bus signals.)	The bus input to the Delay block has two dimensions.

Characteristics

Direct Feedthrough	Yes, when you clear **Prevent direct feedthrough by increasing delay length to lower limit**
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of the input `u` when **Initial condition** is nonscalar and sample-based processing applies
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

- Resettable Delay
- Variable Integer Delay

Demux

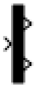
Purpose

Extract and output elements of vector signal

Library

Signal Routing

Description



The Demux block extracts the components of an input signal and outputs the components as separate signals. The output signals are ordered from top to bottom output port. See "How to Rotate a Block" for a description of the port order for various block orientations. To avoid adding clutter to a model, Simulink hides the name of a Demux block when you copy it from the Simulink library to a model. See "Mux Signals" for information about creating and decomposing vectors.

The **Number of outputs** parameter allows you to 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 for you.

The Demux block operates in either vector mode or bus selection mode, depending on whether you selected the **Bus selection mode** parameter. The two modes differ in the types of signals they accept. Vector mode accepts only a vector-like signal, that is, either a scalar (one-element array), vector (1-D array), or a column or row vector (one row or one column 2-D array). Bus selection mode accepts only a bus signal.

Note MathWorks discourages enabling **Bus selection mode** and using a Demux block to extract elements of a bus signal. Muxes and buses should not be intermixed in new models. See "Avoid Mux/Bus Mixtures" for more information.

The **Number of outputs** parameter determines the number and dimensionality of the block outputs, depending on the mode in which the block operates.

Specifying the Number of Outputs in Vector Mode

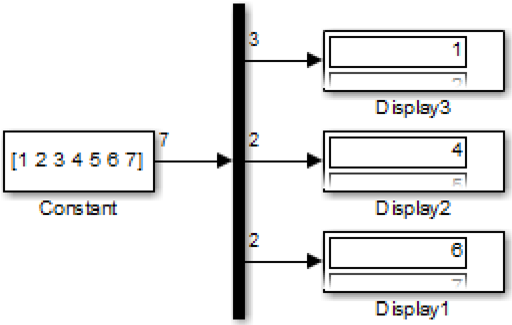
In vector mode, the value of the parameter can be a scalar specifying the number of outputs or a vector whose elements specify the widths of the block's output ports. The block determines the size of its outputs from the size of the input signal and the value of the **Number of outputs** parameter.

The following table summarizes how the block determines the outputs for an input vector of width `n`.

Parameter Value	Block outputs...	Comments
`p = n`	`p` scalar signals	For example, if the input is a three-element vector and you specify three outputs, the block outputs three scalar signals.
`p > n`	Error	
`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.
$[p_1\ p_2\ \ldots\ p_m]$ $p_1+p_2+\ldots+p_m=n$ $p_i > 0$	`m` vector signals having widths $p_1,\ p_2,\ \ldots\ p_m$	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.

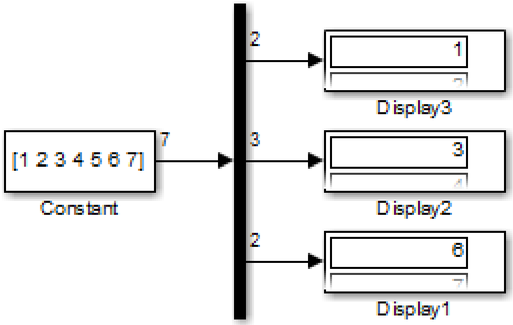
Parameter Value	**Block outputs...**	**Comments**
$[p_1\ p_2\ \ldots\ p_m]$ $p_1+p_2+\ldots+p_m=n$ some or all $p_i = -1$	`m` vector signals	If `pi` is greater than zero, the corresponding output has width p_i. If p_i is -1, the width of the corresponding output is dynamically sized.
$[p_1\ p_2\ \ldots\ p_m]$ $p_1+p_2+\ldots+p_m != n$ $p_i => 0$	Error	

Note that you can specify the number of outputs as fewer than the number of input elements, in which case the block distributes the elements as evenly as possible over the outputs as illustrated in the following example:



You can use –1 in a vector expression to indicate that the block should dynamically size the corresponding port. For example, the expression `[-1, 3 -1]` causes the block to output three signals where the second signal always has three elements. The sizes of the first and third signals depend on the size of the input signal.

If a vector expression comprises positive values and –1 values, the block assigns as many elements as needed to the ports with positive values and distributes the remain elements as evenly as possible over the ports with –1 values. For example, suppose that the block input is seven elements wide and you specify the output as `[-1, 3 -1]`. In this case, the block outputs two elements on the first port, three elements on the second, and two elements on the third.

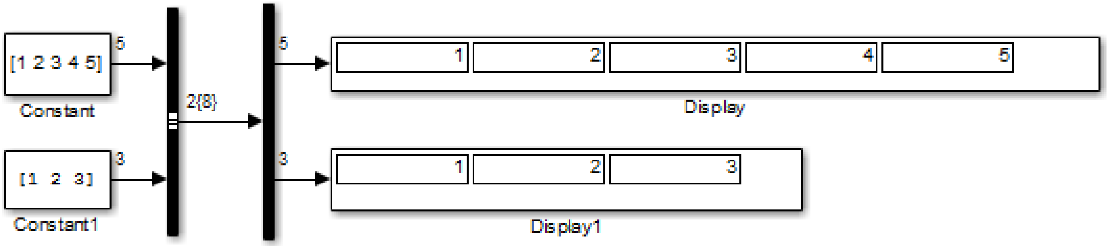

[1 2 3 4 5 6 7]
Constant
7
2
3
2
1
Display3
3
Display2
6
Display1

Specifying the Number of Outputs in Bus Selection Mode

In bus selection mode, the value of the **Number of outputs** parameter can be a:

- Scalar specifying the number of output ports

  The specified value must equal the number of input signals. For example, if the input bus comprises two signals and the value of this parameter is a scalar, the value must equal `2`.

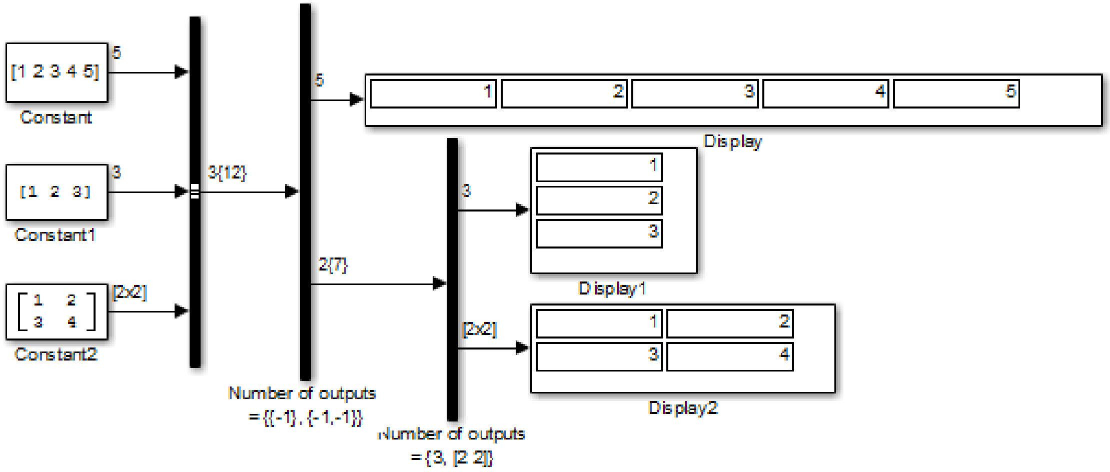

[1 2 3 4 5]
5
Constant
[1 2 3]
3
Constant1
2{8}
5
1
2
3
4
5
Display
3
1
2
3
Display1

- Vector each of whose elements specifies the number of signals to output on the corresponding port

  For example, if the input bus contains five signals, you can specify the output as `[3, 2]`, in which case the block outputs three of the input signals on one port and the other two signals on a second port.

- Cell array each of whose elements is a cell array of vectors specifying the dimensions of the signals output by the corresponding port

The cell array format constrains the Demux block to accept only signals of specified dimensions. For example, the cell array `{{[2 2], 3} {1}}` tells the block to accept only a bus signal comprising a 2-by-2 matrix, a three-element vector, and a scalar signal. You can use the value –1 in a cell array expression to let the block determine the dimensionality of a particular output based on the input. For example, the following diagram uses the cell array expression `{{ 1}, { 1, 1}}` to specify the output of the leftmost Demux block.


[1 2 3 4 5]
5
Constant
[1 2 3]
3
Constant1
1 2
3 4
[2x2]
Constant2
3{12}
5
2{7}
Number of outputs
={{-1}, {-1,-1}}
3
[2x2]
Number of outputs
={3, [2 2]}
1
2
3
4
5
Display
1
2
3
Display1
1
2
3
4
Display2

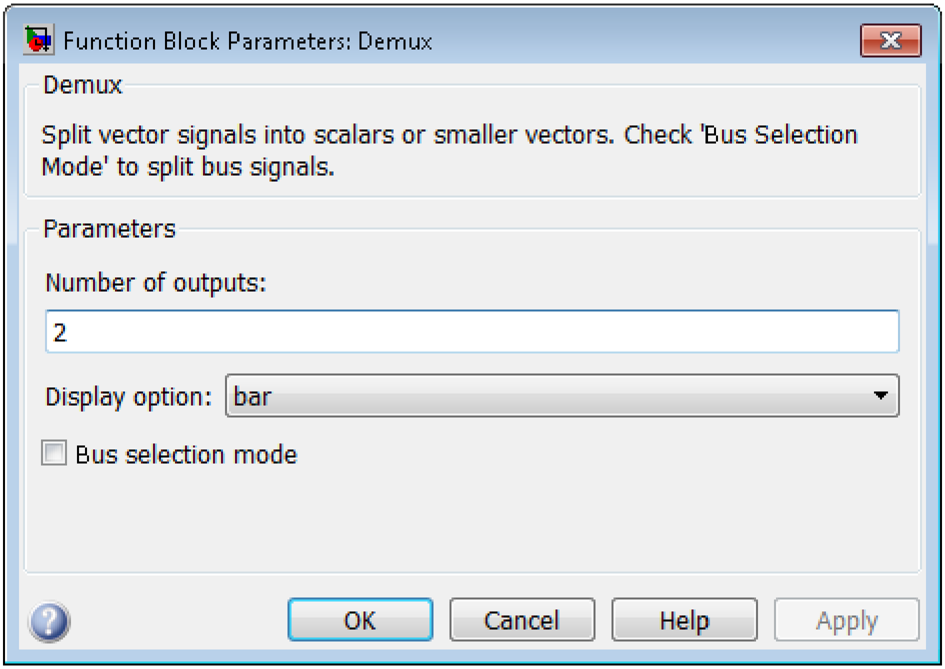
In bus selection mode, if you specify the dimensionality of an output port (that is, specify any value other than –1), the corresponding input element must match the specified dimensionality.

Note MathWorks discourages enabling **Bus selection mode** and using a Demux block to extract elements of a bus signal. Muxes and buses should not be intermixed in new models. See “Avoid Mux/Bus Mixtures” for more information.

Data Type Support

The Demux block accepts and outputs complex or real signals of any data type that Simulink supports, including fixed-point and enumerated data types.

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



Function Block Parameters: Demux
Demux

Split vector signals into scalars or smaller vectors. Check 'Bus Selection Mode' to split bus signals.

Parameters

Number of outputs: 2

Display option: bar

☐ Bus selection mode

OK
Cancel
Help
Apply

Parameters and Dialog Box

Number of outputs

Specify the number and dimensions of outputs.

Settings

Default: `2`

This block interprets this parameter depending on the **Bus selection mode** parameter. See the block description for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See the Demux block reference page for more information.

Display option

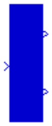
Select options to display the Demux block. The options are

Settings

Default: `bar`

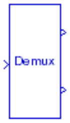
`bar`

Display the icon as a solid bar of the block's foreground color.



`none`

Display the icon as a box containing the block's type name.



Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See the Demux block reference page for more information.

Bus selection mode

Enable bus selection mode.

Settings

Default: Off

☑ On
   Enable bus selection mode.

☐ Off
   Disable bus selection mode.

Tips

MathWorks discourages enabling **Bus selection mode** and using a Demux block to extract elements of a bus signal. Muxes and buses should not be intermixed in new models. See “Avoid Mux/Bus Mixtures” for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Characteristics

Virtual	Yes For more information, see “Virtual Blocks” in the Simulink documentation.

See Also

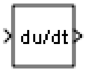
Mux

Derivative

Purpose Output time derivative of input

Library Continuous

Description The Derivative block approximates the derivative of the input signal u with respect to the simulation time t. You obtain the approximation of



$$\frac{du}{dt},$$

by computing a numerical difference $\Delta u / \Delta t$, where Δu is the change in input value and Δt is the change in time since the previous simulation (major) time step.

This block accepts one input and generates one output. The initial output for the block is zero.

The precise relationship between the input and output of this block is:

$$y(t) = \frac{\Delta u}{\Delta t} = \left. \frac{u(t) - u(T_{previous})}{t - T_{previous}} \right| t > T_{previous},$$

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 very sensitive to the dynamics of the entire model. The accuracy of the output signal depends on the size of the time steps taken in the simulation. Smaller steps allow a smoother and more accurate output curve from this block. However, unlike with blocks that have continuous states, the solver does not take smaller steps when the input to this block changes rapidly. Depending on the dynamics of the driving signal and model, the output signal of this block might contain unexpected fluctuations. These fluctuations are primarily due to the driving signal output and solver step size.

Because of these sensitivities, structure your models to use integrators (such as Integrator blocks) instead of Derivative blocks. Integrator blocks have states that allow solvers to adjust step size and improve accuracy of the simulation. See "Circuit Model" for an example of choosing the best-form mathematical model to avoid using Derivative blocks in your models,

If you must use the Derivative block with a variable step solver, set the solver maximum step size settings to a value such that the Derivative block can generate answers with adequate accuracy. To determine this value, you might need to repeatedly run the simulation using different solver settings.

When the input to this block is a discrete signal, the continuous derivative of the input exhibits an impulse when the value of the input changes. Otherwise, it is 0. Alternatively, you can define the discrete derivative of a discrete signal using the difference of the last two values of the signal, as follows:

$$y(k) = \frac{1}{\Delta t}(u(k) - u(k-1))$$

Taking the z-transform of this equation results in:

$$\frac{Y(z)}{u(z)} = \frac{1 - z^{-1}}{\Delta t} = \frac{z-1}{\Delta t \cdot z}.$$

The Discrete Derivative block models this behavior. Use this block instead of the Derivative bock to approximate the discrete-time derivative of a discrete signal.

Improved Linearization with Transfer Fcn Blocks

The Laplace domain transfer function for the operation of differentiation is:

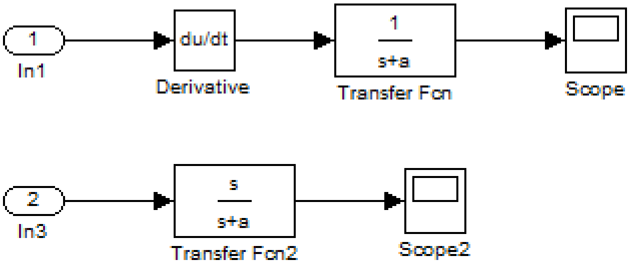
$$Y(s) \big/ X(s) = s$$

This equation is not a proper transfer function, nor does it have a state-space representation. As such, the Simulink software linearizes this block as an effective gain of 0 unless you explicitly specify that a proper first-order transfer function should be used to approximate the linear behavior of this block (see "Coefficient c in the transfer function approximation s/(c*s + 1) used for linearization" on page 1-324).

To improve linearization, you can also try to incorporate the derivative term in other blocks. For example, if you have a Derivative block in series with a Transfer Fcn block, try using a single Transfer Fcn block of the form

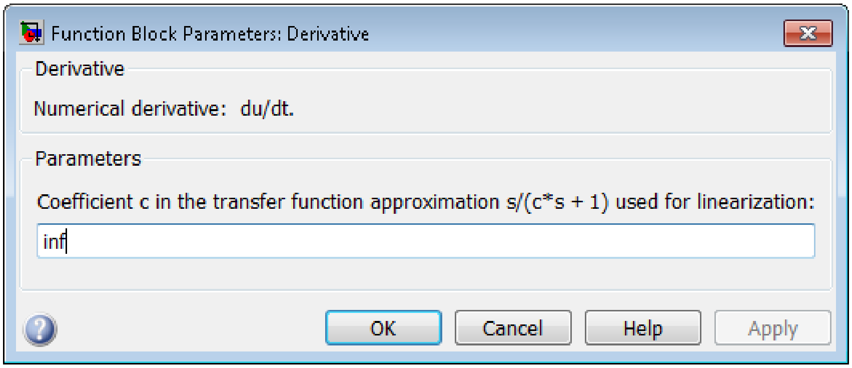
$$\frac{s}{s+a}.$$

For example, you can replace the first set of blocks in this figure with the blocks below them.



Data Type Support

The Derivative block accepts and outputs a real signal of type `double`. For more information, see "Data Types Supported by Simulink".


Function Block Parameters: Derivative
Derivative

Numerical derivative: du/dt.

Parameters

Coefficient c in the transfer function approximation s/(c*s + 1) used for linearization:

inf

OK
Cancel
Help
Apply

Parameters and Dialog Box

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.

Settings

Default: `inf`

- The exact linearization of the Derivative block is difficult, because the dynamic equation for the block is $y = \dot{u}$, which you cannot represent as a state-space system. However, you can approximate the linearization by adding a pole to the Derivative to create a transfer function $s/(c*s+1)$. The addition of a pole filters the signal before differentiating it, which removes the effect of noise.
- The default value `inf` corresponds to a linearization of `0`.

Tips

- A best practice is to change the value of *c* to $\frac{1}{f_b}$, where f_b is the break frequency for the filter.
- **Coefficient c in the transfer function approximation s/(c*s+1) used for linearization** must be a finite positive value. This value must be nonzero.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous
Scalar Expansion	N/A
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also Discrete Derivative

Detect Change

Purpose

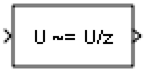
Detect change in signal value

Library

Logic and Bit Operations

Description

The Detect Change block determines if an input does not equal its previous value.



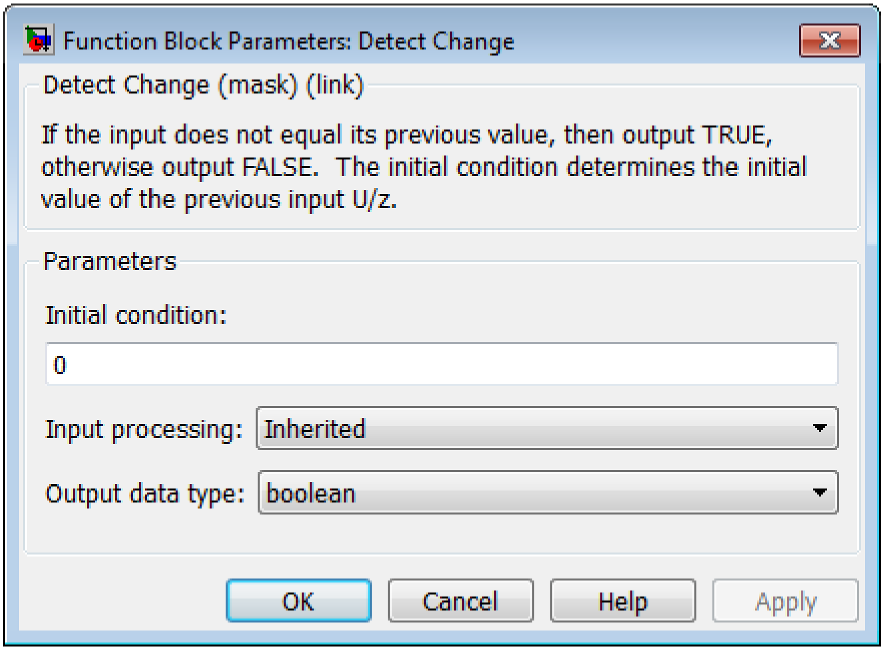
- The output is true (equal to 1) when the input signal does not equal its previous value.
- The output is false (equal to 0) when the input signal equals its previous value.

Data Type Support

The Detect Change block accepts signals of the following data types:

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

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



Function Block Parameters: Detect Change
Detect Change (mask) (link)

If the input does not equal its previous value, then output TRUE, otherwise output FALSE. The initial condition determines the initial value of the previous input U/z.

Parameters

Initial condition: 0

Input processing: Inherited

Output data type: boolean

OK
Cancel
Help
Apply

Parameters and Dialog Box

Initial condition

Set the initial condition for the previous input U/z.

Input processing

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

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

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

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

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

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

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

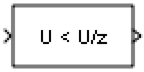
See Also

Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Purpose Detect decrease in signal value

Library Logic and Bit Operations

Description The Detect Decrease block determines if an input is strictly less than its previous value.

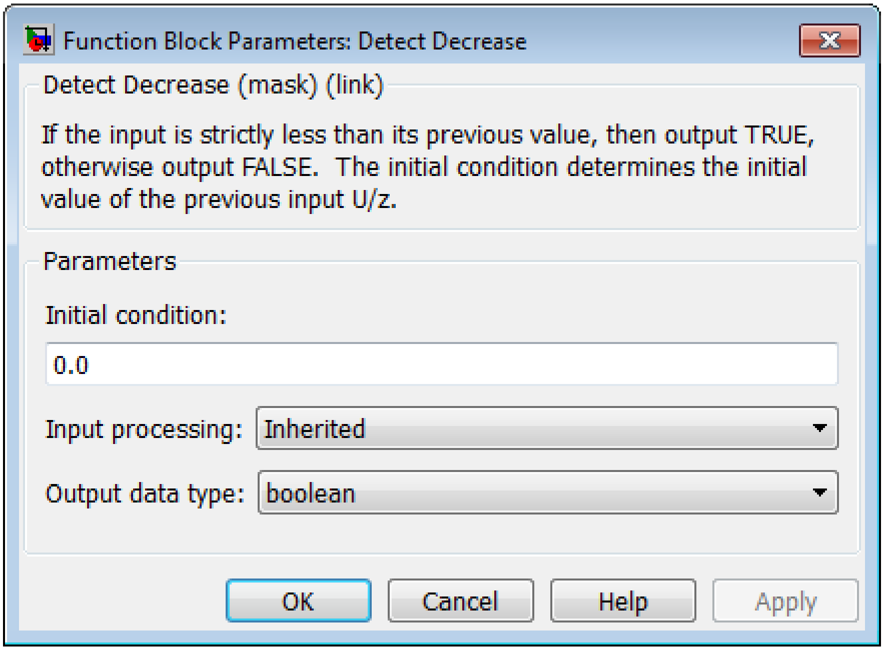


- The output is true (equal to 1) when the input signal is less than its previous value.
- The output is false (equal to 0) when the input signal is greater than or equal to its previous value.

Data Type Support The Detect Decrease block accepts signals of the following data types:

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

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



Function Block Parameters: Detect Decrease
Detect Decrease (mask) (link)

If the input is strictly less than its previous value, then output TRUE, otherwise output FALSE. The initial condition determines the initial value of the previous input U/z.

Parameters

Initial condition: 0.0

Input processing: Inherited

Output data type: boolean

OK
Cancel
Help
Apply

Parameters and Dialog Box

Initial condition

Set the initial condition for the previous input U/z.

Input processing

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

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

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

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

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

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

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Detect Change, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Detect Fall Negative

Purpose

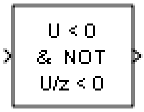
Detect falling edge when signal value decreases to strictly negative value, and its previous value was nonnegative

Library

Logic and Bit Operations

Description

The Detect Fall Negative block determines if the input is less than zero, and its previous value was greater than or equal to zero.



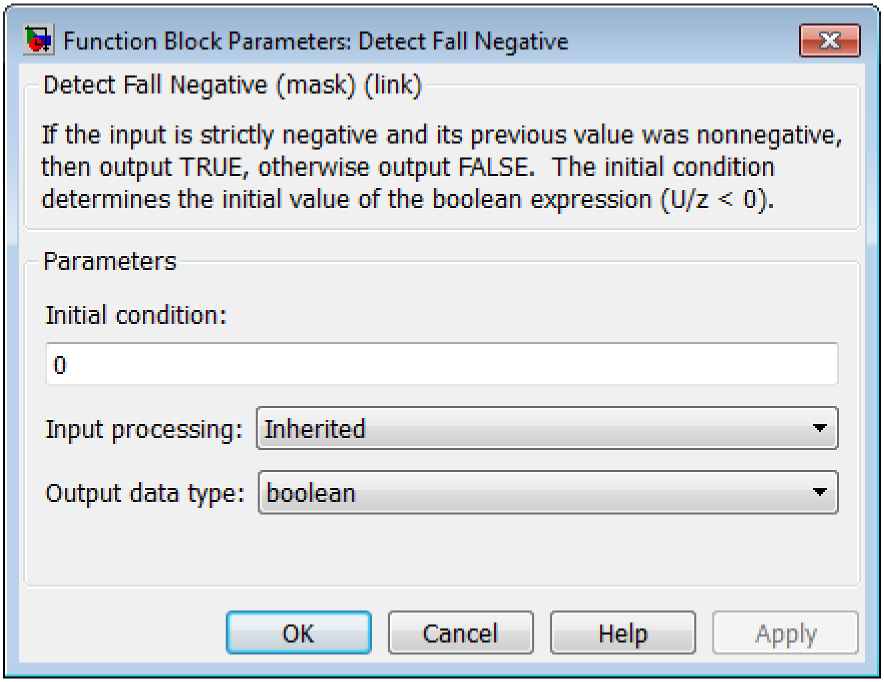
- The output is true (equal to 1) when the input signal is less than zero, and its previous value was greater than or equal to zero.
- The output is false (equal to 0) when the input signal is greater than or equal to zero, or if the input signal is negative, its previous value was also negative.

Data Type Support

The Detect Fall Negative block accepts signals of the following data types:

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

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



Detect Fall Negative (mask) (link)

If the input is strictly negative and its previous value was nonnegative, then output TRUE, otherwise output FALSE. The initial condition determines the initial value of the boolean expression (U/z < 0).

Parameters

Initial condition: 0

Input processing: Inherited

Output data type: boolean

Parameters and Dialog Box

Initial condition

Set the initial condition of the Boolean expression U/z < 0.

Input processing

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

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

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

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

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

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

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

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

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

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Detect Change, Detect Decrease, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Detect Fall Nonpositive

Purpose

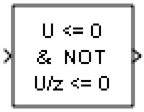
Detect falling edge when signal value decreases to nonpositive value, and its previous value was strictly positive

Library

Logic and Bit Operations

Description

The Detect Fall Nonpositive block determines if the input is less than or equal to zero, and its previous value was greater than zero.



- The output is true (equal to 1) when the input signal is less than or equal to zero, and its previous value was greater than zero.
- The output is false (equal to 0) when the input signal is greater than zero, or if it is nonpositive, its previous value was also nonpositive.

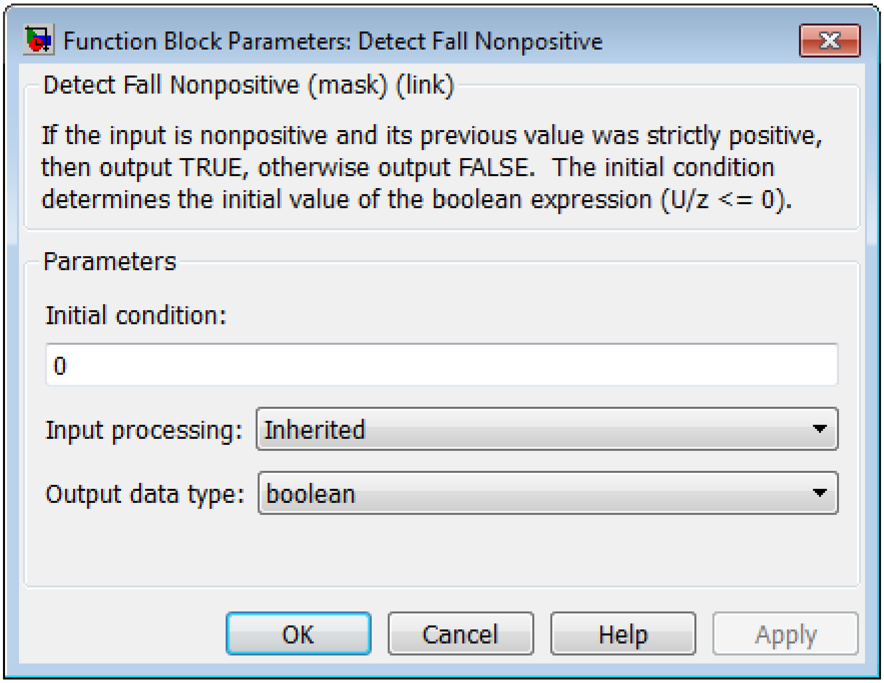
Data Type Support

The Detect Fall Nonpositive block accepts signals of the following data types:

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

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

Parameters and Dialog Box



Detect Fall Nonpositive (mask) (link)

If the input is nonpositive and its previous value was strictly positive, then output TRUE, otherwise output FALSE. The initial condition determines the initial value of the boolean expression (U/z <= 0).

Parameters

Initial condition: 0

Input processing: Inherited

Output data type: boolean

Initial condition

Set the initial condition of the Boolean expression U/z <= 0.

Input processing

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

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

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

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

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

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

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Detect Increase

Purpose

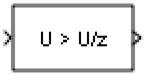
Detect increase in signal value

Library

Logic and Bit Operations

Description

The Detect Increase block determines if an input is strictly greater than its previous value.



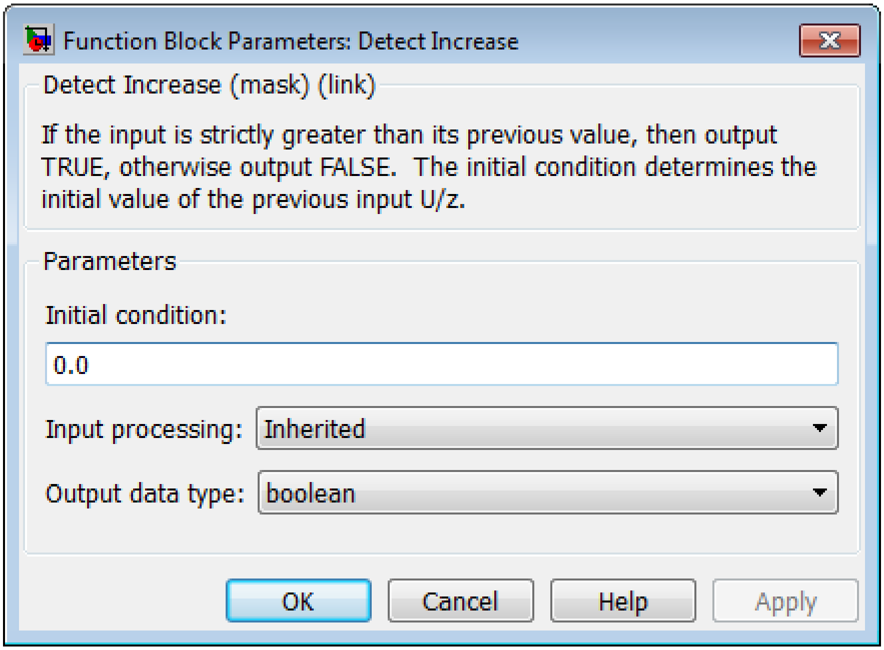
- The output is true (equal to 1) when the input signal is greater than its previous value.
- The output is false (equal to 0) when the input signal is less than or equal to its previous value.

Data Type Support

The Detect Increase block accepts signals of the following data types:

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

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



Detect Increase (mask) (link)

If the input is strictly greater than its previous value, then output TRUE, otherwise output FALSE. The initial condition determines the initial value of the previous input U/z.

Parameters

Initial condition: 0.0

Input processing: Inherited

Output data type: boolean

Parameters and Dialog Box

Initial condition

Set the initial condition for the previous input U/z.

Input processing

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

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

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

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

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

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

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Rise Nonnegative, Detect Rise Positive

Detect Rise Nonnegative

Purpose

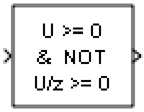
Detect rising edge when signal value increases to nonnegative value, and its previous value was strictly negative

Library

Logic and Bit Operations

Description

The Detect Rise Nonnegative block determines if the input is greater than or equal to zero, and its previous value was less than zero.



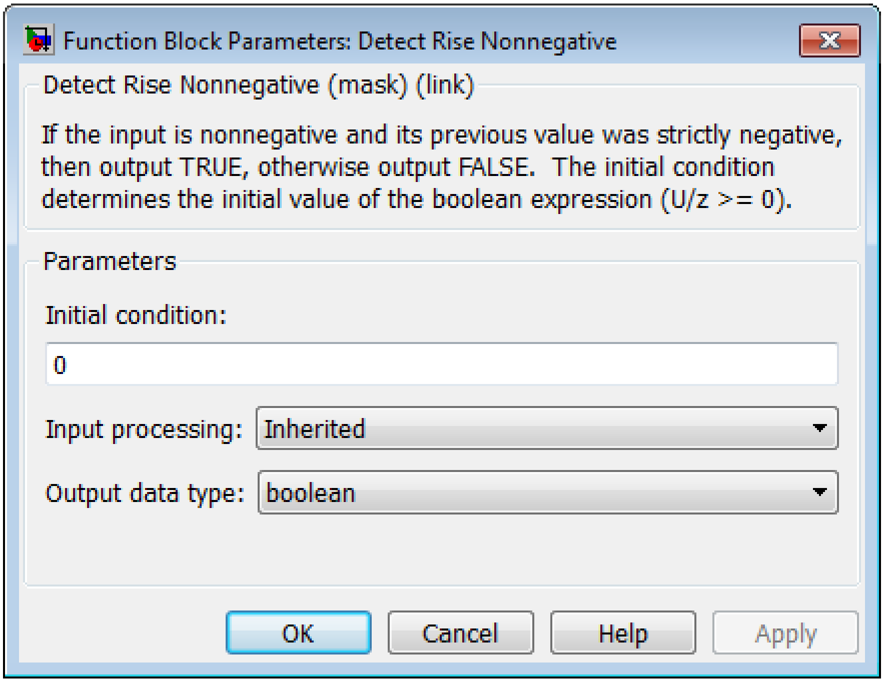
- The output is true (equal to 1) when the input signal is greater than or equal to zero, and its previous value was less than zero.
- The output is false (equal to 0) when the input signal is less than zero, or if the input signal is nonnegative, its previous value was also nonnegative.

Data Type Support

The Detect Rise Nonnegative block accepts signals of the following data types:

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

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



Function Block Parameters: Detect Rise Nonnegative

Detect Rise Nonnegative (mask) (link)

If the input is nonnegative and its previous value was strictly negative, then output TRUE, otherwise output FALSE. The initial condition determines the initial value of the boolean expression (U/z >= 0).

Parameters

Initial condition: 0

Input processing: Inherited

Output data type: boolean

Parameters and Dialog Box

Initial condition

Set the initial condition of the Boolean expression U/z >= 0.

Input processing

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

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

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

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

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

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

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Positive

Purpose

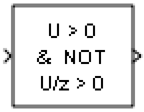
Detect rising edge when signal value increases to strictly positive value, and its previous value was nonpositive

Library

Logic and Bit Operations

Description

The Detect Rise Positive block determines if the input is strictly positive, and its previous value was nonpositive.



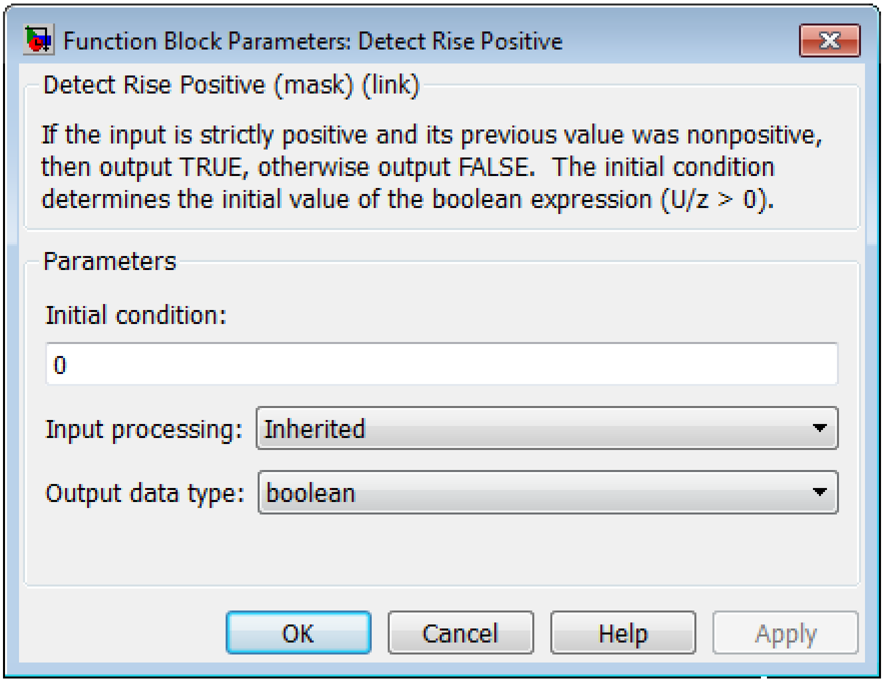
- The output is true (equal to 1) when the input signal is greater than zero, and the previous value was less than or equal to zero.
- The output is false (equal to 0) when the input is negative or zero, or if the input is positive, the previous value was also positive.

Data Type Support

The Detect Rise Positive block accepts signals of the following data types:

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

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


Function Block Parameters: Detect Rise Positive
Detect Rise Positive (mask) (link)

If the input is strictly positive and its previous value was nonpositive, then output TRUE, otherwise output FALSE. The initial condition determines the initial value of the boolean expression (U/z > 0).

Parameters

Initial condition: 0

Input processing: Inherited

Output data type: boolean

OK
Cancel
Help
Apply

Parameters and Dialog Box

Initial condition

Set the initial condition of the Boolean expression U/z > 0.

Input processing

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

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

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

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

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

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

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative

Difference

Purpose

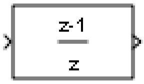
Calculate change in signal over one time step

Library

Discrete

Description

The Difference block outputs the current input value minus the previous input value.

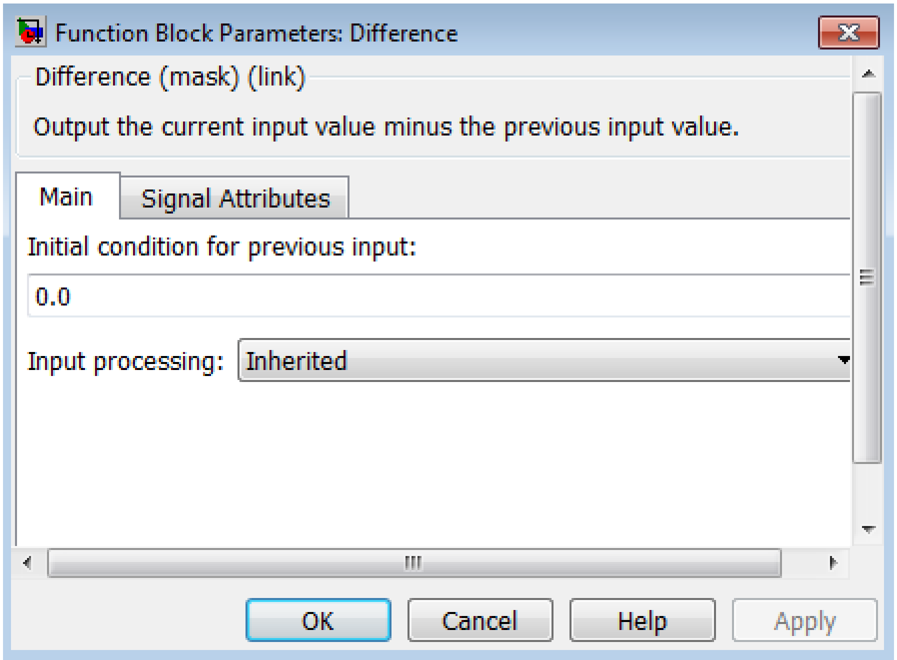


Data Type Support

The Difference block accepts signals of any numeric data type that Simulink supports, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Difference block dialog box appears as follows:



Function Block Parameters: Difference

Difference (mask) (link)

Output the current input value minus the previous input value.

Initial condition for previous input: 0.0

Input processing: Inherited

Initial condition for previous input

Set the initial condition for the previous input.

Input processing

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

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

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

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

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

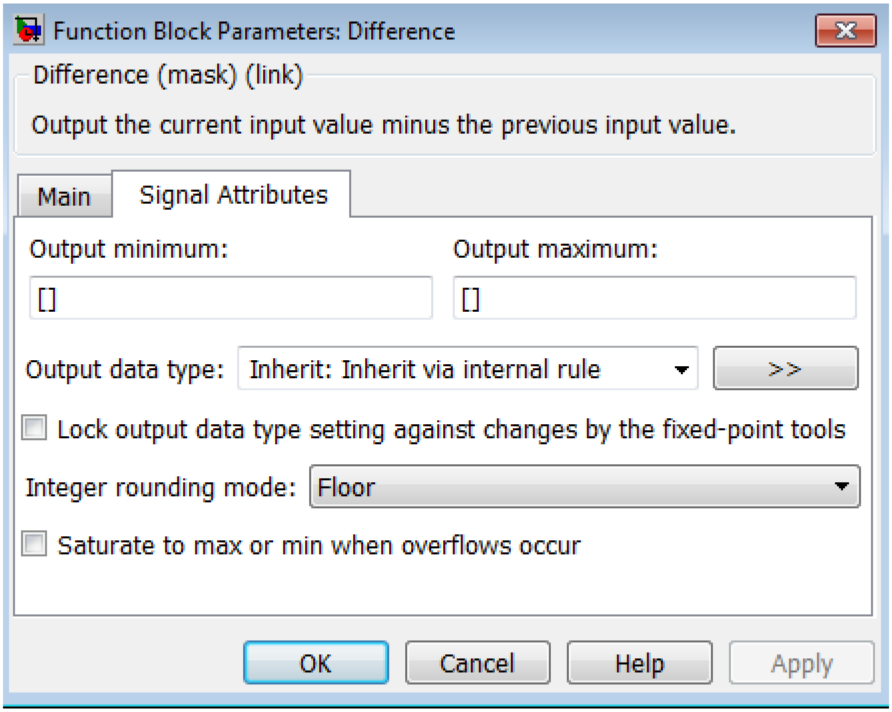
Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

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

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

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

The **Signal Attributes** pane of the Difference block dialog box appears as follows:



Function Block Parameters: Difference

Difference (mask) (link)

Output the current input value minus the previous input value.

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

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

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Output minimum

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

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

Output maximum

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

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

Output data type

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

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

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

See “Specify Block Output Data Types” in the *Simulink User’s Guide* for more information.

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

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

Integer rounding mode

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

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

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

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of inputs and gain

Digital Clock

Purpose

Output simulation time at specified sampling interval

Library

Sources

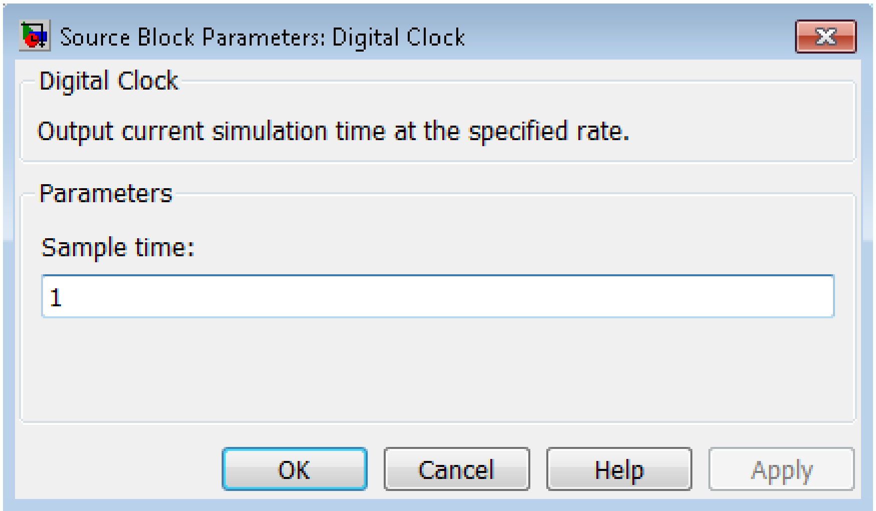
Description

The Digital Clock block outputs the simulation time only at the specified sampling interval. At other times, the block holds the output at the previous value. To control the precision of this block, set the **Sample time** parameter in the block dialog box.

Use this block rather than the Clock block (which outputs continuous time) when you need the current simulation time within a discrete system.

Data Type Support

The Digital Clock block outputs a real signal of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Source Block Parameters: Digital Clock

Digital Clock

Output current simulation time at the specified rate.

Parameters

Sample time:

1

Parameters and Dialog Box

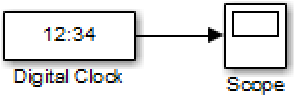
Sample time

Specify the sampling interval. The default value is `1` second. For more information, see Specifying Sample Time in the Simulink documentation.

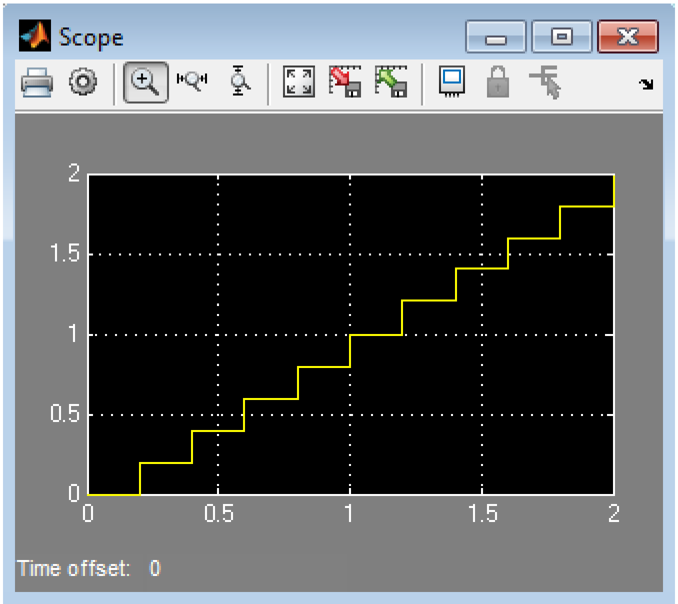
Do not specify a continuous sample time, either `0` or `[0,0]`. Also, avoid specifying `-1` (inheriting the sample time) because this block is a source.

Examples

In the following model, the Scope block shows the output of a Digital Clock block with a **Sample time** of `0.2`.



The Digital Clock block outputs the simulation time every 0.2 seconds. Otherwise, the block holds the output at the previous value.


Scope
Time offset: 0
0
0.5
1
1.5
2
0
0.5
1
1.5
2

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Clock

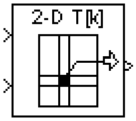
Purpose

Index into N-dimensional table to retrieve element, column, or 2-D matrix

Library

Lookup Tables

Description



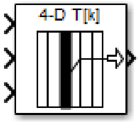
Block Inputs and Outputs

The Direct Lookup Table (n-D) block uses inputs as zero-based indices into an *n*-dimensional table. The number of inputs varies with the shape of the output: an element, column, or 2-D matrix.

You define a set of output values as the **Table data** parameter. The first input specifies the zero-based index to the table dimension that is *one higher* than the output dimensionality. The next input specifies the zero-based index to the next table dimension, and so on.

Output Shape	Output Dimensionality	Table Dimension That Maps to the First Input
Element	0	1
Column	1	2
Matrix	2	3

Suppose that you want to select a column of values from a 4-D table:



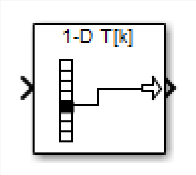
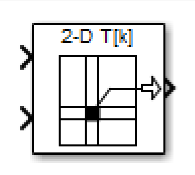
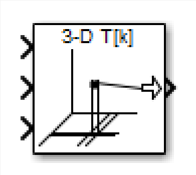
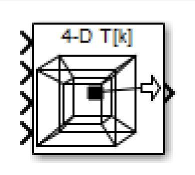
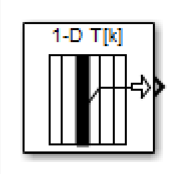
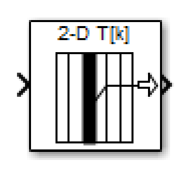
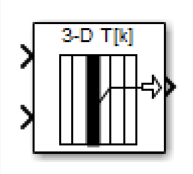
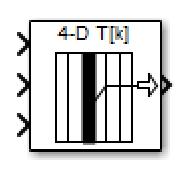
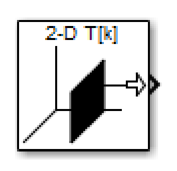
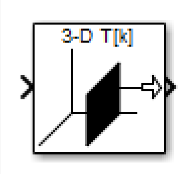
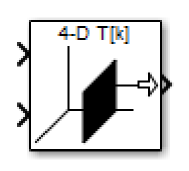
The following mapping of block input port to table dimension applies:

This input port...	Is the index for this table dimension...
1	2
2	3
3	4

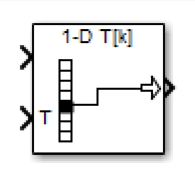
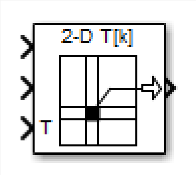
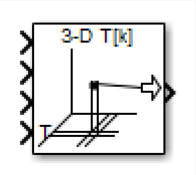
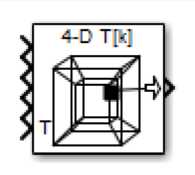
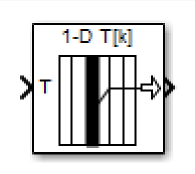
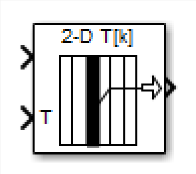
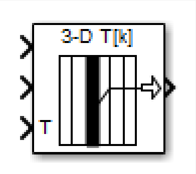
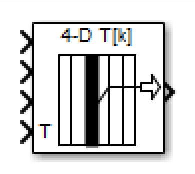
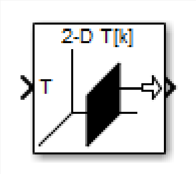
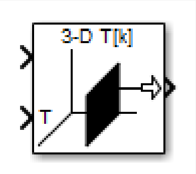
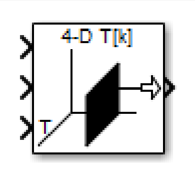
Changes in Block Icon Appearance

Depending on parameters you set, the block icon changes appearance. For table dimensions higher than 4, the icon matches the 4-D version but shows the exact number of dimensions in the top text.

When you use the **Table data** parameter, you see the following icons:

Object That Inputs Select from the Table	**Number of Table Dimensions**			
	1	**2**	**3**	**4**
Element				
Column				
2-D Matrix	Not applicable			

When you use the table input port, you see the following icons:

Object That Inputs Select from the Table	**Number of Table Dimensions**			
	1	**2**	**3**	**4**
Element				
Column				
2-D Matrix	Not applicable			

Data Type Support

The Direct Lookup Table (n-D) block accepts input signals of different data types.

Type of Input Port	Data Types Supported
Index port	• Floating point • Built-in integer • Boolean
Table port (with the label `T`)	• Floating point • Built-in integer • Fixed point • Boolean

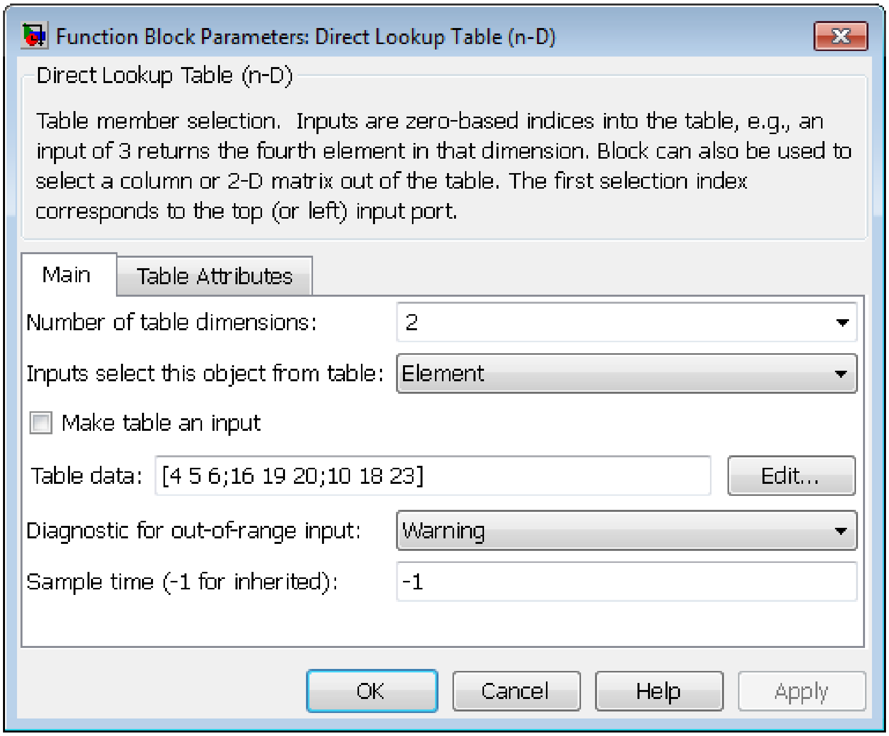
The output data type is the same as the table data type. Inputs for indexing must be real, but table data can be complex.

When the table data is...	The block inherits the output type from...
Not an input	The **Table data type** parameter
An input	The table input port

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

Parameters and Dialog Box

The **Main** pane of the Direct Lookup Table (n-D) block appears as follows:



Function Block Parameters: Direct Lookup Table (n-D)
Direct Lookup Table (n-D)
Main
Table Attributes
Edit...
OK
Cancel
Help
Apply

Table member selection. Inputs are zero-based indices into the table, e.g., an input of 3 returns the fourth element in that dimension. Block can also be used to select a column or 2-D matrix out of the table. The first selection index corresponds to the top (or left) input port.

Number of table dimensions: 2

Inputs select this object from table: Element

☐ Make table an input

Table data: [4 5 6;16 19 20;10 18 23]

Diagnostic for out-of-range input: Warning

Sample time (-1 for inherited): -1

Number of table dimensions

Specify the number of dimensions that the **Table data** parameter must have. This value determines the number of independent variables for the table and the number of inputs to the block.

To specify...	Do this...
1, 2, 3, or 4	Select the value from the drop-down list.
A higher number of table dimensions	Enter a positive integer directly in the field. The maximum number of table dimensions that this block supports is 30.

Inputs select this object from table

Specify whether the output data is a single element, a column, or a 2-D matrix. The number of input ports for indexing depends on your selection.

Selection	Number of Input Ports for Indexing
`Element`	Number of table dimensions
`Column`	Number of table dimensions – 1
`2-D Matrix`	Number of table dimensions – 2

This numbering matches MATLAB indexing. For example, if you have a 4-D table of data, follow these guidelines:

To access...	Specify...	As in...
An element	Four indices	`array(1,2,3,4)`
A column	Three indices	`array(:,2,3,4)`
A 2-D matrix	Two indices	`array(:,:,3,4)`

Make table an input

Select this check box to force the Direct Lookup Table (n-D) block to ignore the **Table data** parameter. Instead, a new input port appears with `T` next to it. Use this port to input table data.

Table data

Specify the table of output values. The matrix size must match the dimensions of the **Number of table dimensions** parameter. The **Table data** field is available only if you clear the **Make table an input** check box.

Tip During block diagram editing, you can leave the **Table data** field empty. But for simulation, you must match the number of dimensions in **Table data** to the **Number of table dimensions**. For details on how to construct multidimensional MATLAB arrays, see "Multidimensional Arrays" in the MATLAB documentation.

Click **Edit** to open the Lookup Table Editor. For more information, see "Edit Existing LookupTables" in the Simulink documentation.

Diagnostic for out-of-range input

Specify whether to show a warning or error when an index is out of range with respect to the table dimension. Options include:

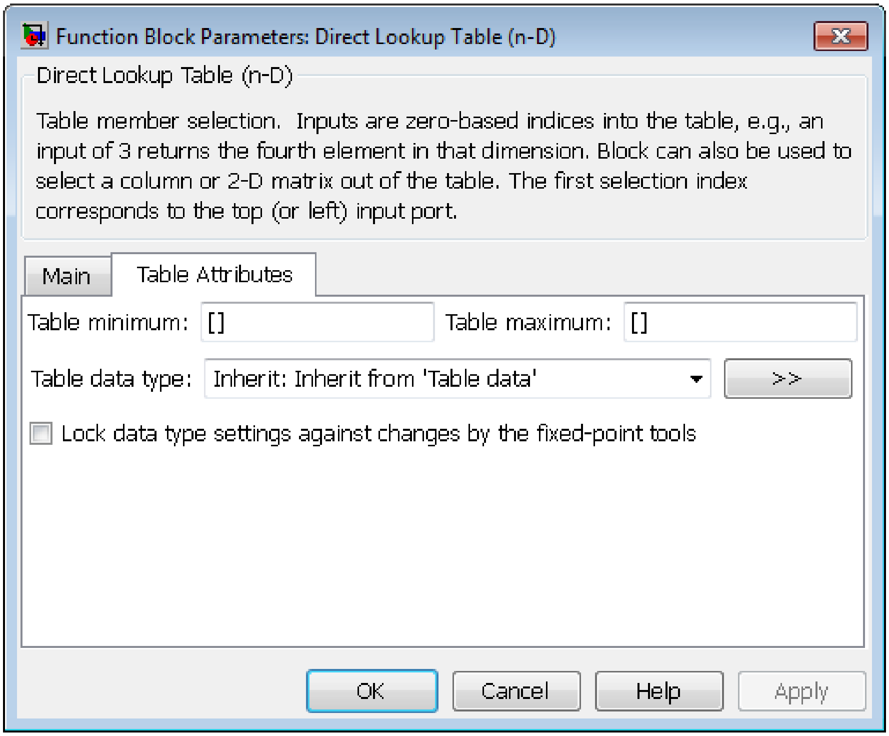
- `None` — do not display any warning or error message
- `Warning` — display a warning message in the MATLAB Command Window and continue the simulation
- `Error` — halt the simulation and display an error message in the Simulation Diagnostics Viewer

When you select `None` or `Warning`, the block clamps out-of-range indices to fit table dimensions. For example, if the specified index is 5.3 and the maximum index for that table dimension is 4, the block clamps the index to 4.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation for more information.

The **Table Attributes** pane of the Direct Lookup Table (n-D) block appears as follows:



Function Block Parameters: Direct Lookup Table (n-D)

Direct Lookup Table (n-D)

Table member selection. Inputs are zero-based indices into the table, e.g., an input of 3 returns the fourth element in that dimension. Block can also be used to select a column or 2-D matrix out of the table. The first selection index corresponds to the top (or left) input port.

Table minimum: []

Table maximum: []

Table data type: Inherit: Inherit from 'Table data'

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

---

Note The parameters in the **Table Attributes** pane are not available if you select **Make table an input**. In this case, the block inherits all table attributes from the input port with the label `T`.

Table minimum

Specify the minimum value for table data. The default value is `[]` (unspecified).

Table maximum

Specify the maximum value for table data. The default value is `[]` (unspecified).

Table data type

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

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

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

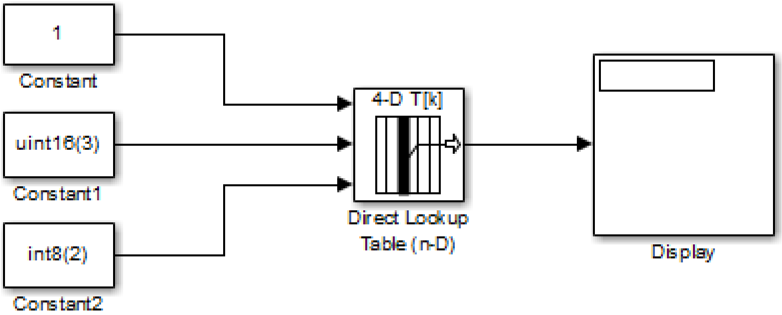
Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Locking the Output Data Type Setting”in the Fixed-Point Designer documentation.

Examples

When Table Data Is Not an Input

Suppose that you have the following model:


1
Constant
uint16(3)
Constant1
int8(2)
Constant2
4-D T[k]
Direct Lookup Table (n-D)
Display

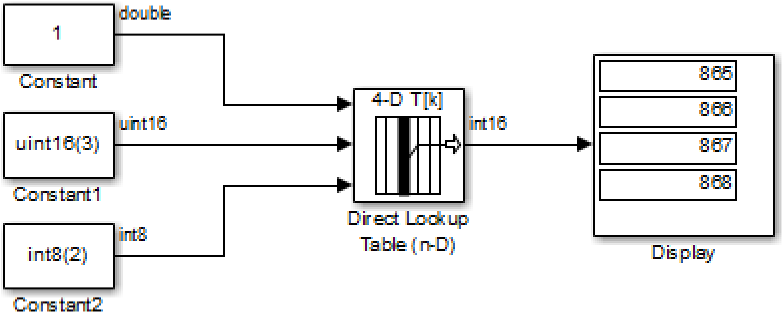
The Direct Lookup Table (n-D) block parameters are:

Block Parameter	Value
Number of table dimensions	`4`
Inputs select this object from table	`Column`
Make table an input	`off`
Table data	`a`
Diagnostic for out-of-range input	`Warning`
Sample time	`1`
Table minimum	`[]`
Table maximum	`[]`
Table data type	`int16`
Lock data type settings against changes by the fixed-point tools	`off`

In this example, `a` is a 4-D array of linearly increasing values that you define with the following model preload function:

```
a = reshape(1:2800, [4 5 20 7]);
```

When you run the model, you get the following results:


1
double
Constant
uint16(3)
uint16
Constant1
int8(2)
int8
Constant2
4-D T[k]
int16
Direct Lookup Table (n-D)
865
866
867
868
Display

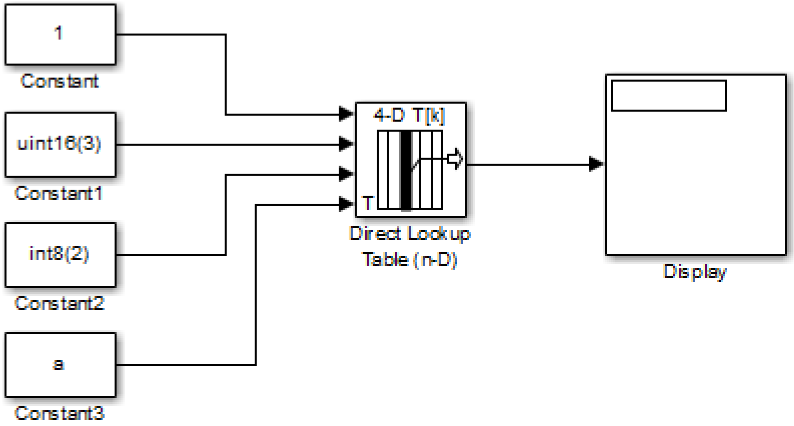
Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```
a(:,2,4,3)
```

The output has the same data type as the table: `int16`.

When Table Data Is an Input

Suppose that you have the following model:


1
Constant
uint16(3)
Constant1
int8(2)
Constant2
a
Constant3
4-D T[k]
T
Direct Lookup Table (n-D)
Display

The Direct Lookup Table (n-D) block parameters are:

Block Parameter	Value
Number of table dimensions	`4`
Inputs select this object from table	`Column`
Make table an input	`on`
Diagnostic for out-of-range input	`Warning`
Sample time	`1`

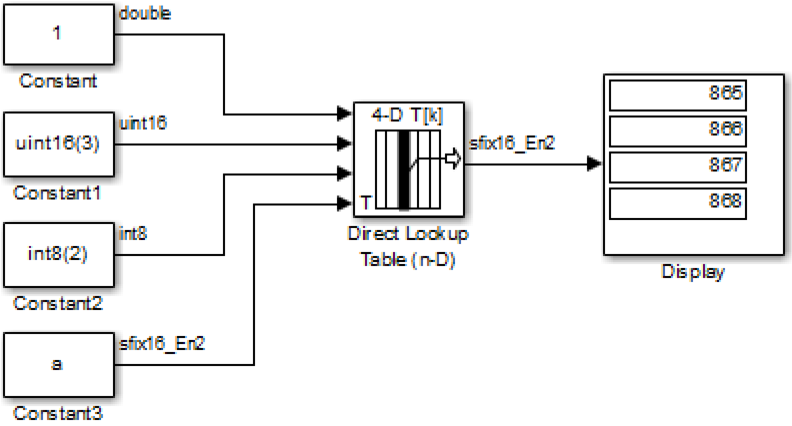
The key parameters of the Constant3 block are:

Block Parameter	Value
Constant value	`a`
Output data type	`fixdt(1,16,2)`

In this example, `a` is a 4-D array of linearly increasing values that you define with the following model preload function:

```
a = reshape(1:2800, [4 5 20 7]);
```

When you run the model, you get the following results:


1
double
Constant
uint16(3)
uint16
Constant1
int8(2)
int8
Constant2
a
sfix16_En2
Constant3
4-D T[k]
T
Direct Lookup Table (n-D)
sfix16_En2
865
866
867
868
Display

The Constant3 block feeds the 4-D array to the Direct Lookup Table (n-D) block, using the fixed-point data type `fixdt(1,16,2)`. Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```
a(:,2,4,3)
```

The output has the same data type as the table: `fixdt(1,16,2)`.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	For scalar lookups only (not when returning a column or a 2-D matrix from the table)
Dimensionalized	For scalar lookups only (not when returning a column or a 2-D matrix from the table)
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

n-D Lookup Table

Purpose

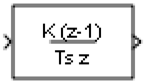
Compute discrete-time derivative

Library

Discrete

Description

The Discrete Derivative block computes an optionally scaled discrete time derivative as follows



$$y(t_n) = \frac{Ku(t_n)}{T_s} - \frac{Ku(t_{n-1})}{T_s}$$

where $u(t_n)$ and $y(t_n)$ are the block's input and output at the current time step, respectively, $u(t_{n-1})$ is the block's input at the previous time step, K is a scaling factor, and T_s is the simulation's discrete step size, which must be fixed.

Guidelines for Usage in Triggered Subsystems

When you use the Discrete Derivative block in triggered subsystems, follow these guidelines:

- When the **Sample time type** parameter of the trigger port is `triggered`, verify that your model does not trigger the subsystem at the first time step.

  If your model triggers the subsystem at the first time step, the Discrete Derivative block generates a divide-by-zero warning message. This behavior occurs because the time step at t = 0 is zero.

- When the **Sample time type** parameter of the trigger port is `periodic`, your model can trigger the subsystem at any time step.

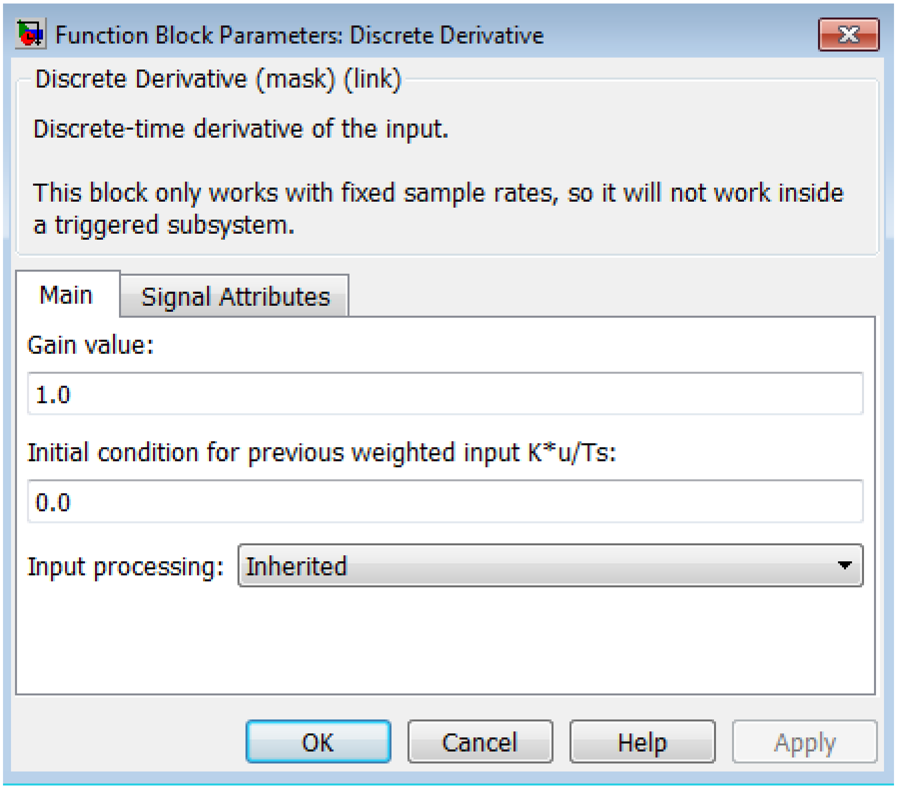
Data Type Support

The Discrete Derivative block supports all numeric Simulink data types, including fixed-point data types.

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

Parameters and Dialog Box

The **Main** pane of the Discrete Derivative block dialog box appears as follows:



Function Block Parameters: Discrete Derivative
Main
Signal Attributes
OK
Cancel
Help
Apply

Discrete Derivative (mask) (link)

Discrete-time derivative of the input.

This block only works with fixed sample rates, so it will not work inside a triggered subsystem.

Gain value: 1.0

Initial condition for previous weighted input K*u/Ts: 0.0

Input processing: Inherited

Gain value

Scaling factor used to weight the block's input at the current time step.

Initial condition for previous weighted input K*u/Ts

Set the initial condition for the previous scaled input.

Input processing

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

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

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

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

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

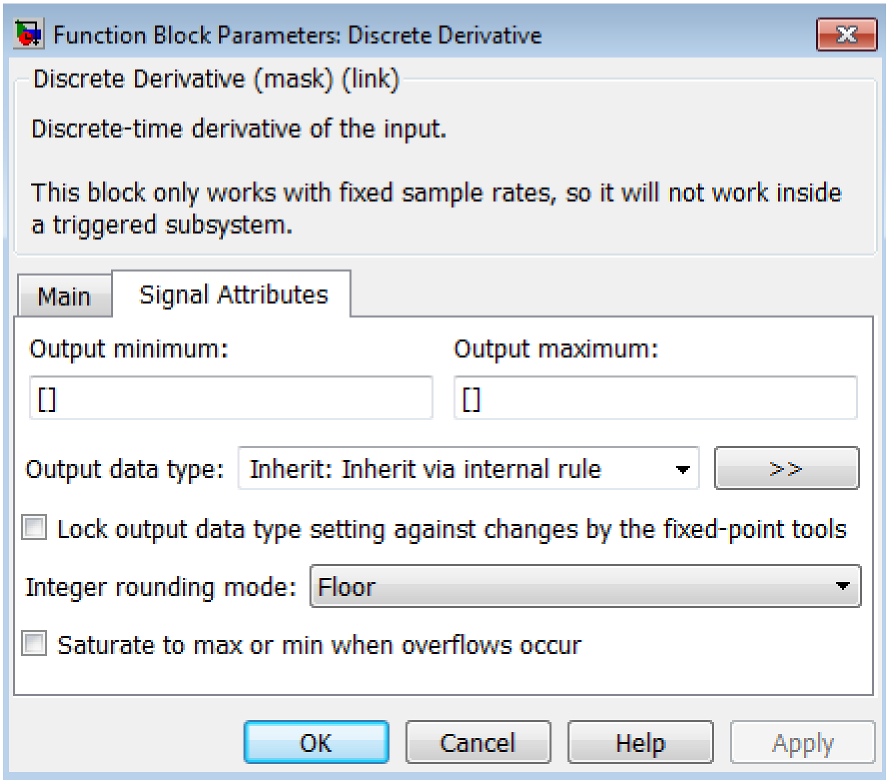
Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

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

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

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

The **Signal Attributes** pane of the Discrete Derivative block dialog box appears as follows:


Function Block Parameters: Discrete Derivative

Discrete Derivative (mask) (link)

Discrete-time derivative of the input.

This block only works with fixed sample rates, so it will not work inside a triggered subsystem.

Main
Signal Attributes

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

>>

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

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

OK
Cancel
Help
Apply

Output minimum

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

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

Output maximum

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

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

Output data type

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

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

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

See “Specify Block Output Data Types” in *Simulink User’s Guide* for more information.

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

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

Integer rounding mode

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

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

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

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of inputs and gain

See Also

Derivative

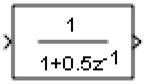
Purpose

Model Infinite Impulse Response (IIR) filters

Library

Discrete

Description



The Discrete Filter block independently filters each channel of the input signal with the specified digital IIR filter. You can specify the filter structure as one of | `Direct form I` | `Direct form I transposed` | `Direct form II` | `Direct form II transposed`. The block implements static filters with fixed coefficients. You can tune the coefficients of these static filters.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify how the block treats each element of the input. You can specify treating input elements as an independent channel (sample-based processing), or treating each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal those of the input, except when you specify a matrix of filter taps for the **Numerator coefficients** parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

Use the **Numerator coefficients** parameter to specify the coefficients of the discrete filter numerator polynomial. Use the **Denominator coefficients** parameter to specify the coefficients of the denominator polynomial of the function. The **Denominator coefficients** parameter must be a vector of coefficients.

Specify the coefficients of the numerator and denominator polynomials in ascending powers of z^{-1}. The Discrete Filter block lets you use polynomials in z^{-1} (the delay operator) to represent a discrete system. This method is the one that signal processing engineers typically use. Conversely, the Discrete Transfer Fcn block lets you use polynomials in z to represent a discrete system. This method is the one that control engineers typically use. When the numerator and denominator polynomials have the same length, the two methods are identical.

Coefficient Source

The Digital Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** Enter information about the filter such as structure and coefficients in the block mask.
- **Input port(s)** Enter the filter structure in the block mask, and the filter coefficients come in through one or more block ports. This mode is useful for specifying time-varying filters.
- **Discrete-time filter object (DFILT)** Specify the filter using a `dfilt` object.

Supported Filter Structures

When you select **Discrete-time filter object (DFILT)**, the following `dfilt` structures are supported:

- `dfilt.df1`
- `dfilt.df1t`
- `dfilt.df2`
- `dfilt.df2t`

The following table shows the vector of filter coefficients you must provide for each filter structure. For more information on how to specify filter coefficients for various filter structures, see “Specify Static Filters” and “Specify Time-Varying Filters”.

Filter Structures and Filter Coefficients

Supported Filter Structures	Filter Coefficient Specification

`Direct form I`
`Direct form I transposed`
`Direct form II`
`Direct form II transposed`
	• Numerator coefficients vector [b0, b1, b2, ..., bn]
	• Denominator coefficients vector [a0, a1, a2, ..., am]
	See Special Consideration for the Leading Denominator Coefficient.

Note Each structure listed in the preceding table supports both fixed-point and floating-point signals.

Special Considerations for the Leading Denominator Coefficient

In some cases, the Digital Filter block requires the leading denominator coefficient (a_0) to be 1. This requirement applies under the following conditions:

- The Digital Filter block is operating in a fixed-point mode. The block operates in a fixed-point mode when at least one of the following statements is true:
    - The input to the Digital Filter block has a fixed-point or integer data type.
    - The **Fixed-point instrumentation mode** parameter in the Fixed-Point Tool has a setting of `Minimums, maximums and overflows`.
- The **Coefficient source** has a setting of `Dialog` or `Input port(s)`.

Note The fixed-point modeling situations described in the previous bullet, require additional settings, if **Coefficient source** is set to `Input port(s)`. You must select the **First denominator coefficient = 1** and also select the **remove a0 term in the structure** check box.

If you use such a configuration and your leading denominator coefficient (a_0) does not equal 1, the Digital Filter block produces an error. To resolve the error, set your leading denominator coefficient to 1 by scaling all numerator and denominator coefficients by a factor of a_0.

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) on page 1-389. 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	For a filter with two delay elements: $[d_1\ d_2]$	Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each

Valid Initial States (Continued)

Initial state	Examples	Description
elements to each channel)	The delay elements for all channels are d1 and d2.	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) on page 1-389).
Vector or matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: $[d_1\ d_2\ D_1\ D_2\ d_1\ d_2]$ or $\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$ • 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) on page 1-389). • 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) on page 1-389), 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)	• Scalar • Matrix (M-by-N)

Frame-Based Processing (Continued)

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
		• 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)	$K \times N$	• Scalar • Column vector (M-by-1)	• Scalar

Sample-Based Processing (Continued)

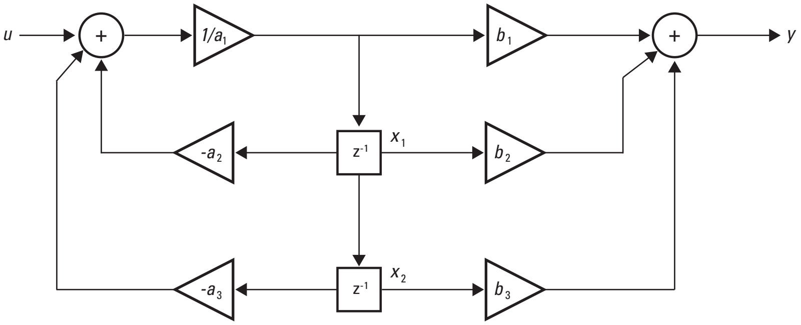
Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
		• Row vector (1-by-M) • Matrix (M-by-(K×N))	

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.

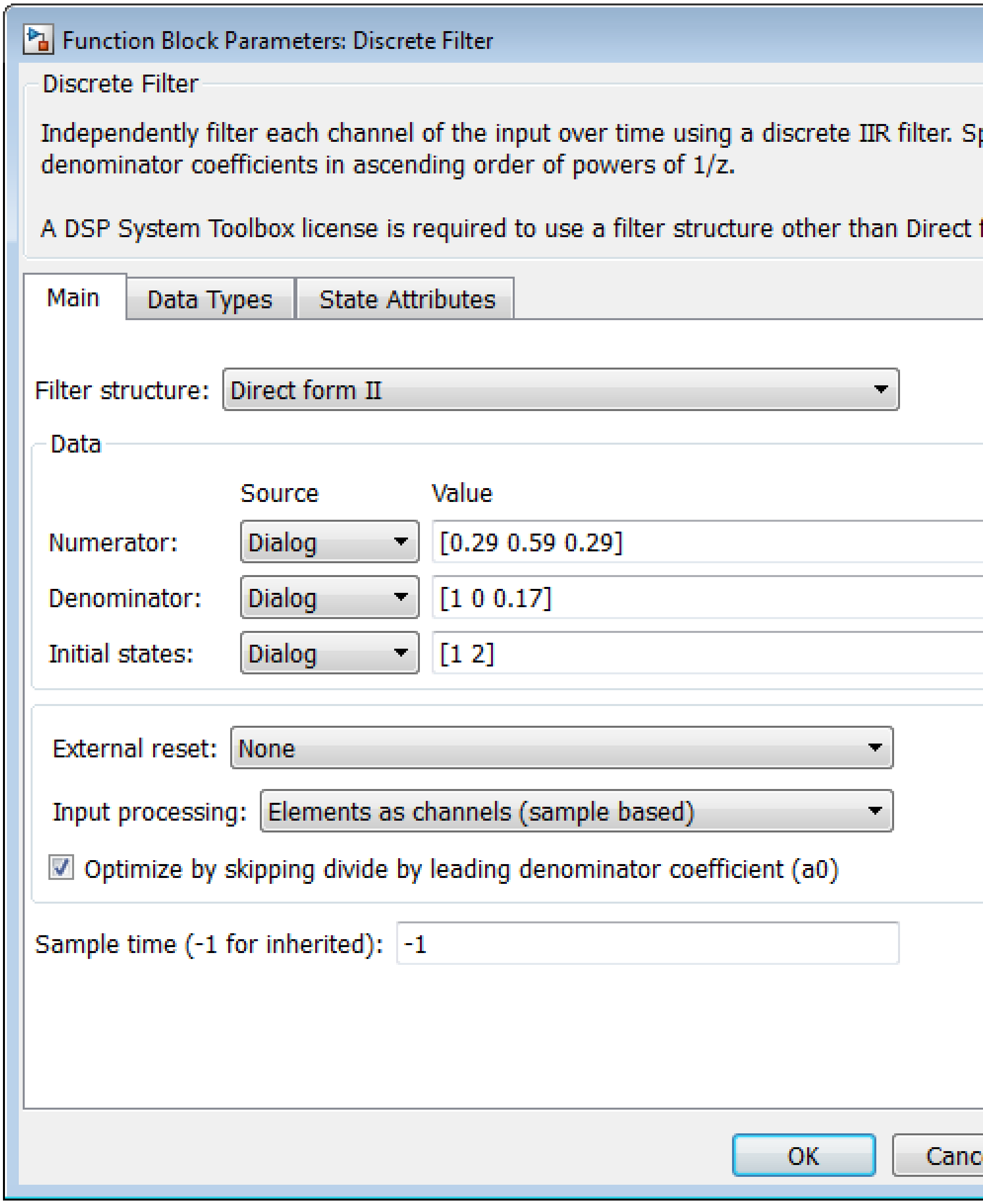
The following example shows the relationship between the initial filter output and the initial input and state. Given an initial input u_1, the first output y_1 is related to the initial state [x_1, x_2] and initial input by:

$$y_1 = b_1\left[\frac{(u_1 - a_2x_1 - a_3x_2)}{a_1}\right] + b_2x_1 + b_3x_2$$



To see an example of how to set initial conditions as a vector:

- Click on the model `ex_discretefilter_nonzero_ic`, or type it at the MATLAB command prompt.
- Double-click on the Discrete Filter block, and set the parameters. The following shows how to set the initial conditions of the Discrete Filter block to [1 2].



Function Block Parameters: Discrete Filter

Discrete Filter

Independently filter each channel of the input over time using a discrete IIR filter. Sp
denominator coefficients in ascending order of powers of 1/z.

A DSP System Toolbox license is required to use a filter structure other than Direct f

Main
Data Types
State Attributes

Filter structure: Direct form II

Data

	Source	Value
Numerator:	Dialog	[0.29 0.59 0.29]
Denominator:	Dialog	[1 0 0.17]
Initial states:	Dialog	[1 2]

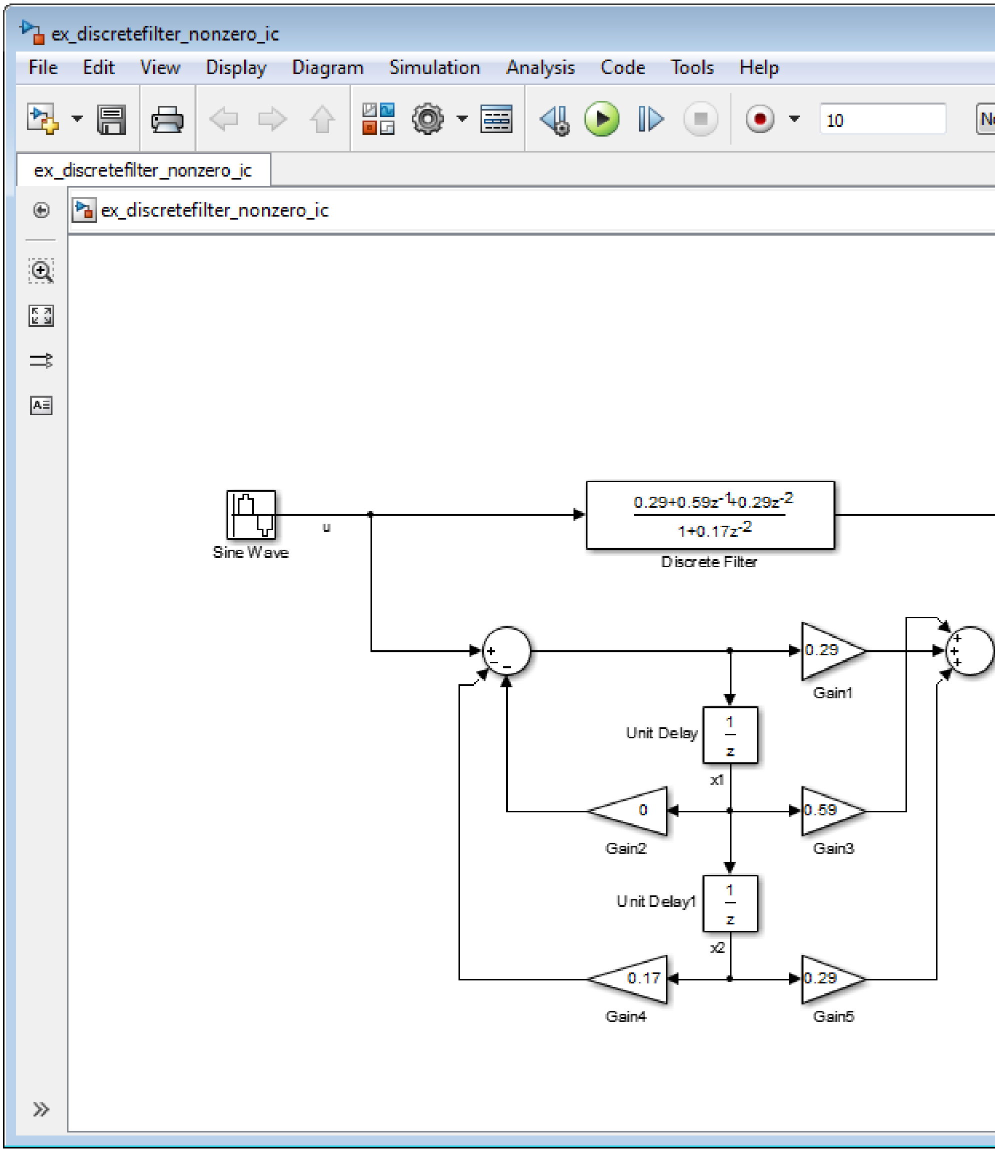
External reset: None

Input processing: Elements as channels (sample based)

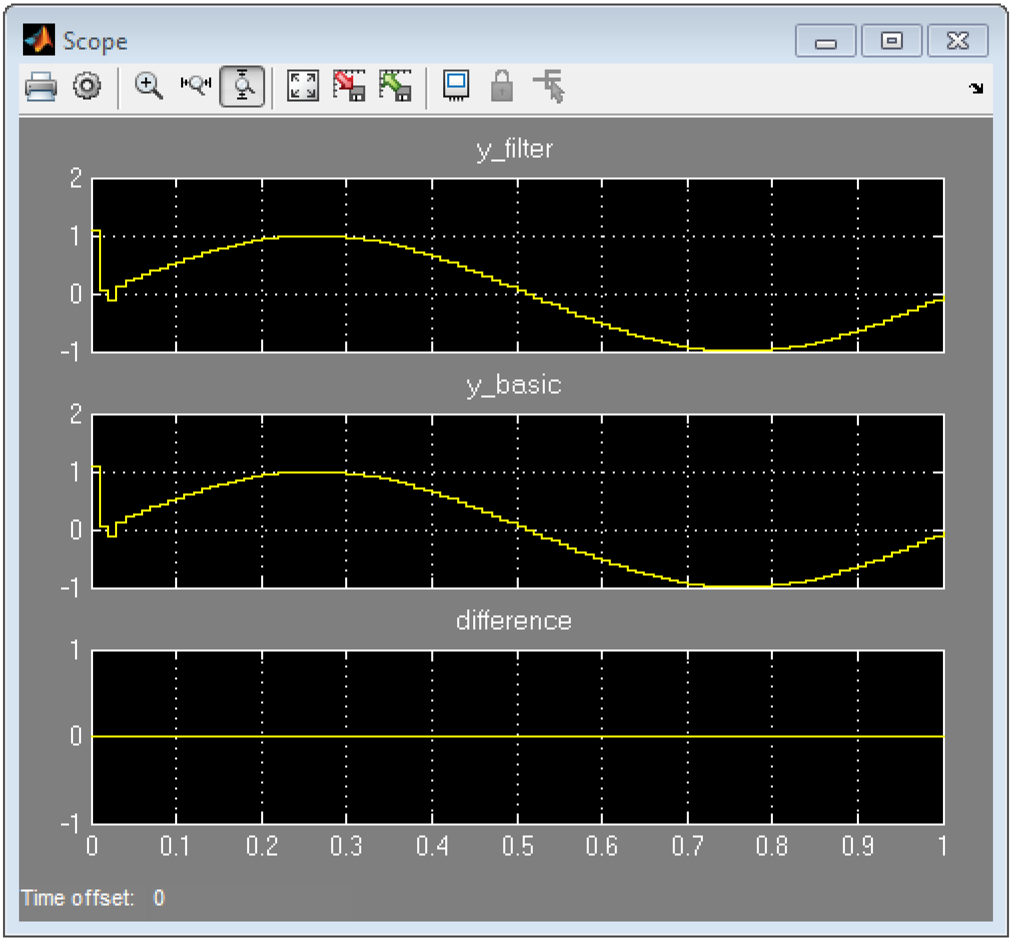
☑ Optimize by skipping divide by leading denominator coefficient (a0)

Sample time (-1 for inherited): -1

- Simulate the model, by left-clicking the green simulation icon.


ex_discretefilter_nonzero_ic
File
Edit
View
Display
Diagram
Simulation
Analysis
Code
Tools
Help
10
ex_discretefilter_nonzero_ic
ex_discretefilter_nonzero_ic
Sine Wave
u
$\frac{0.29+0.59z^{-1}+0.29z^{-2}}{1+0.17z^{-2}}$
Discrete Filter
+
–
+
+
+
0.29
Gain1
Unit Delay
$\frac{1}{z}$
x1
0
Gain2
0.59
Gain3
Unit Delay1
$\frac{1}{z}$
x2
0.17
Gain4
0.29
Gain5

- Double-click the scope. You can see that the difference between the signal filtered by the Discrete Filter block, and the signal from the filter's building blocks, is zero.


Scope
y_filter
2
1
0
-1
y_basic
2
1
0
-1
difference
1
0
-1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Time offset: 0

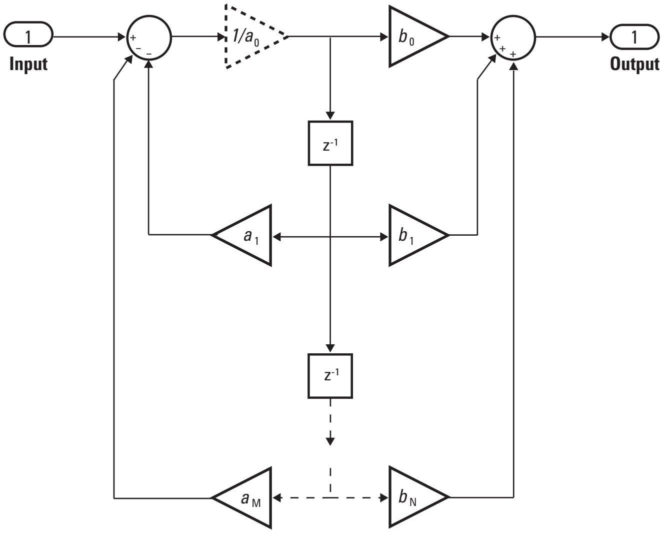
This demonstrates that you can enter the initial conditions of the Discrete Filter block as a vector of [1 2]. You can also set the initial condition of the first Unit Delay to 1 and the second Unit Delay to 2. The resulting outputs are the same.

Data Type Support

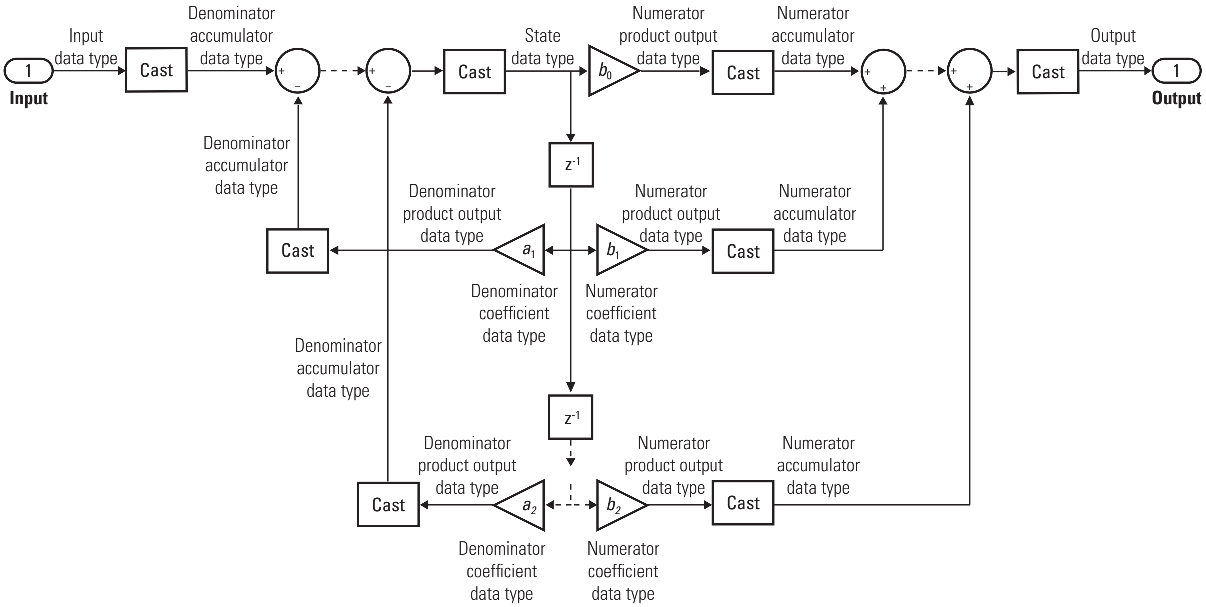
The Discrete Filter block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

The following diagrams show the filter structure and the data types used within the Discrete Filter block for fixed-point signals.

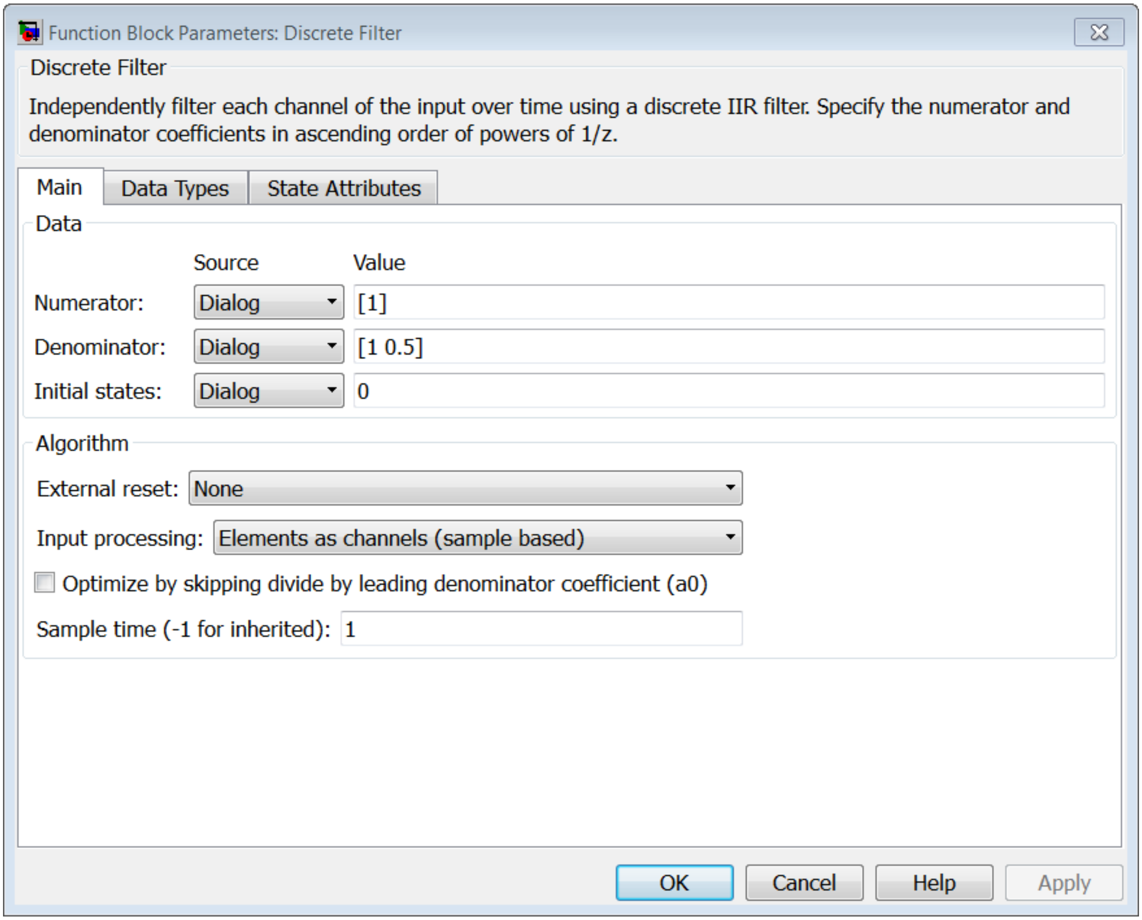


The block omits the dashed divide when you select the **Optimize by skipping divide by leading denominator coefficient (a0)** parameter.


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
+
–
+
–
+
+
+
+

Parameters and Dialog Box

The **Main** pane of the Discrete Filter block dialog box appears as follows.


Function Block Parameters: Discrete Filter
Main
Data Types
State Attributes
OK
Cancel
Help
Apply

Discrete Filter

Independently filter each channel of the input over time using a discrete IIR filter. Specify the numerator and denominator coefficients in ascending order of powers of 1/z.

Data

	Source	Value
Numerator:	Dialog	[1]
Denominator:	Dialog	[1 0.5]
Initial states:	Dialog	0

Algorithm

External reset: None

Input processing: Elements as channels (sample based)

☐ Optimize by skipping divide by leading denominator coefficient (a0)

Sample time (-1 for inherited): 1

Numerator

Numerator coefficients of the discrete filter. To specify the coefficients, set the **Source** to `Dialog`. Then, enter the coefficients in **Value** as descending powers of z. Use a row vector to specify the coefficients for a single numerator polynomial.

Denominator

Denominator coefficients of the discrete filter. To specify the coefficients, set the **Source** to `Dialog`. Then, enter the coefficients in **Value** as descending powers of z. Use a row vector to specify the coefficients for a single denominator polynomial.

Initial states

If the **Source** is `Dialog`, then, in **Value**, specify the initial states of the filter states. To learn how to specify initial states, see "Specifying Initial States" on page 1-518.

If the **Source** is `Input port`, then you do not need to specify **Value**.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
`None`	No reset.
`Rising`	Reset on a rising edge.
`Falling`	Reset on a falling edge.
`Either`	Reset on either a rising or falling edge.
`Level`	Reset in either of these cases: • when there is a nonzero at the current time step • when the time step value changes from nonzero at the previous time step to zero at the current time step
`Level hold`	Reset when nonzero at the current time step.

The reset signal must be scalar.

Input processing

Specify whether the block performs sample- or frame-based processing.

- `Elements as channels (sample based)` — Process each element of the input as an independent channel.
- `Columns as channels (frame based)` — Process each column of the input as an independent channel.

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

For more information, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

Optimize by skipping divide by leading denominator coefficient (a0)

Select when the leading denominator coefficient, a_0, equals 1. This parameter optimizes your code.

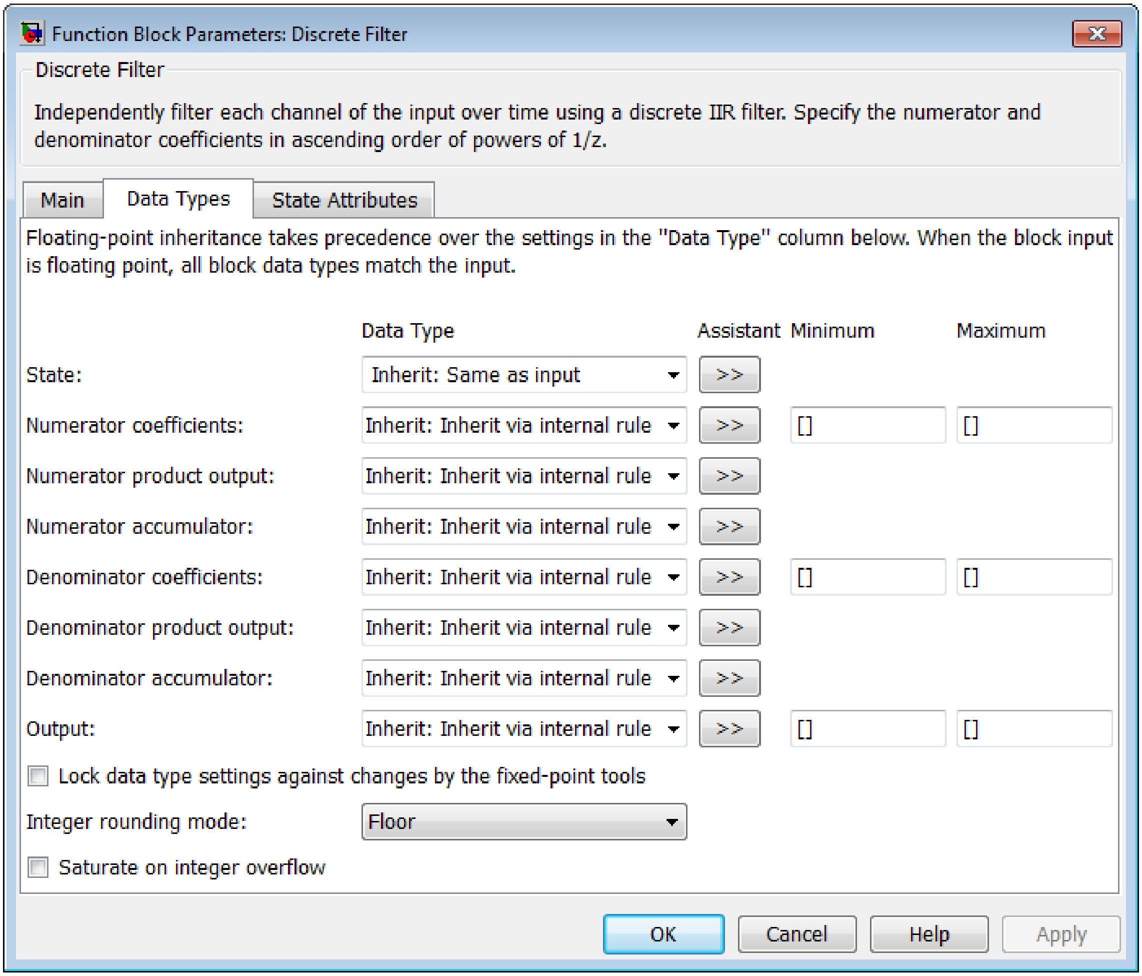
When you select this check box, the block does not perform a divide-by-a_0 either in simulation or in the generated code. An error occurs if a_0 is not equal to one.

When you clear this check box, the block is fully tunable during simulation. It performs a divide-by-a_0 in both simulation and code generation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in "How Simulink Works" in the *Simulink User's Guide*.

The **Data Types** pane of the Discrete Filter block dialog box appears as follows.



Independently filter each channel of the input over time using a discrete IIR filter. Specify the numerator and denominator coefficients in ascending order of powers of 1/z.

Floating-point inheritance takes precedence over the settings in the "Data Type" column below. When the block input is floating point, all block data types match the input.

	Data Type	Assistant	Minimum	Maximum
State:	Inherit: Same as input	>>		
Numerator coefficients:	Inherit: Inherit via internal rule	>>	[]	[]
Numerator product output:	Inherit: Inherit via internal rule	>>		
Numerator accumulator:	Inherit: Inherit via internal rule	>>		
Denominator coefficients:	Inherit: Inherit via internal rule	>>	[]	[]
Denominator product output:	Inherit: Inherit via internal rule	>>		
Denominator accumulator:	Inherit: Inherit via internal rule	>>		
Output:	Inherit: Inherit via internal rule	>>	[]	[]

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

Integer rounding mode: Floor

☐ Saturate on integer overflow

State

Specify the state data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Numerator coefficients

Specify the numerator coefficient data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Numerator coefficient minimum

Specify the minimum value that a numerator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")
- Automatic scaling of fixed-point data types

Numerator coefficient maximum

Specify the maximum value that a numerator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")
- Automatic scaling of fixed-point data types

Numerator product output

Specify the product output data type for the numerator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See "Specify Data Types Using Data Type Assistant" for more information.

Numerator accumulator

Specify the accumulator data type for the numerator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Denominator coefficients

Specify the denominator coefficient data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Denominator coefficient minimum

Specify the minimum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Denominator coefficient maximum

Specify the maximum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Denominator product output

Specify the product output data type for the denominator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Denominator accumulator

Specify the accumulator data type for the denominator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Output

Specify the output data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Block Output Data Types” for more information.

Output minimum

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

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

Output maximum

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

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

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

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Locking the Output Data Type Setting”in the Fixed-Point Designer documentation.

Integer rounding mode

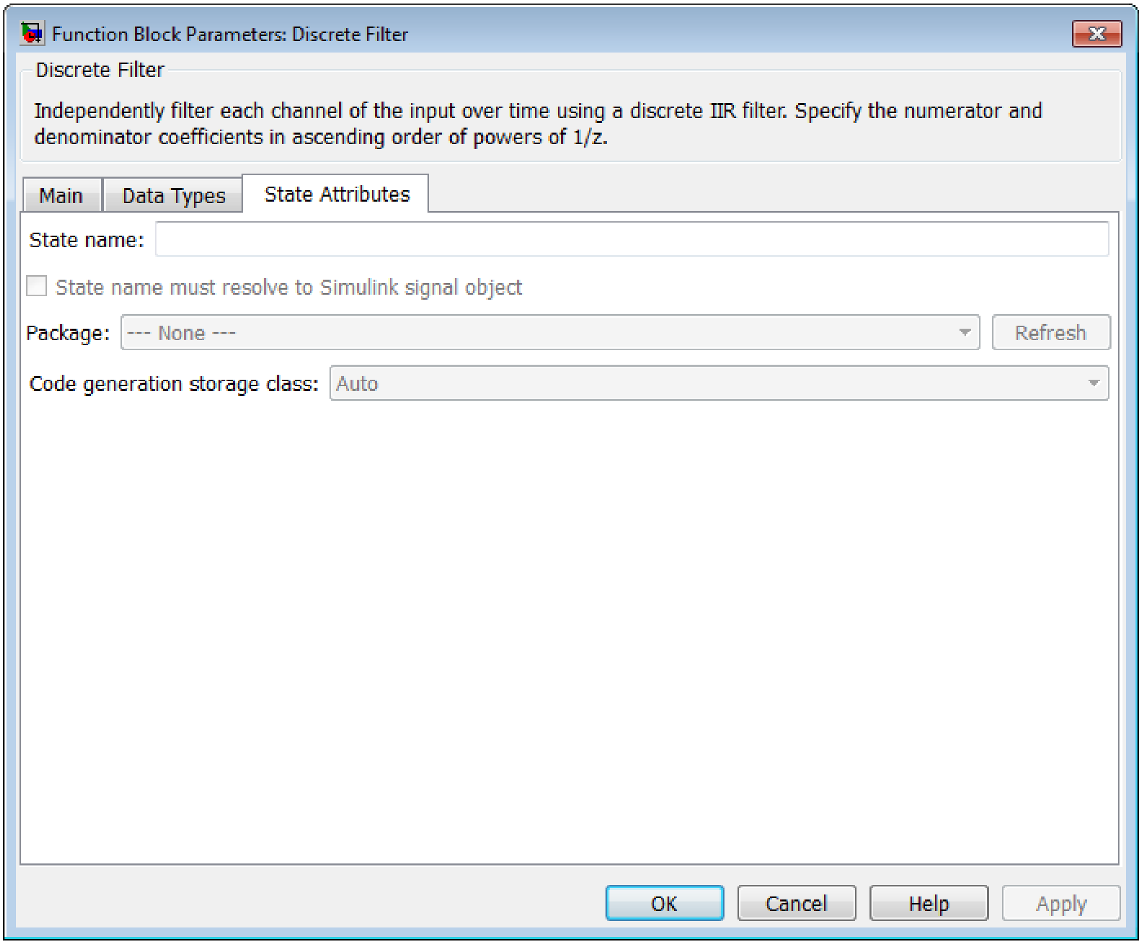
Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 “Checking 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.

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.

The **State Attributes** pane of the Discrete Filter block dialog box appears as follows.


Function Block Parameters: Discrete Filter
Main
Data Types
State Attributes
OK
Cancel
Help
Apply
Refresh

Discrete Filter

Independently filter each channel of the input over time using a discrete IIR filter. Specify the numerator and denominator coefficients in ascending order of powers of 1/z.

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Code generation storage class: Auto

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “States” in the Simulink Coder documentation.

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Package

Select a package that defines the custom storage class you want to apply. If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Code generation storage class

From the list, select a state storage class.

`Auto`

Auto is the storage class to use for states that do not need to interface to external code.

`ExportedGlobal`

The state is stored in a global variable.

`ImportedExtern`

`*model*_private.h` declares the state as an extern variable.

`ImportedExternPointer`

`*model*_private.h` declares the state as an extern pointer.

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Code generation storage type qualifier

Specify a Simulink Coder storage type qualifier. The default is `' '`. When this field is blank, no qualifier is assigned.

The Simulink Coder product does not check this string for errors. Thus, whatever value you enter appears automatically in the variable declaration.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name resolves
- Minimum and maximum values of the signal object

For more information, see “States” in the Simulink Coder documentation.

Characteristics

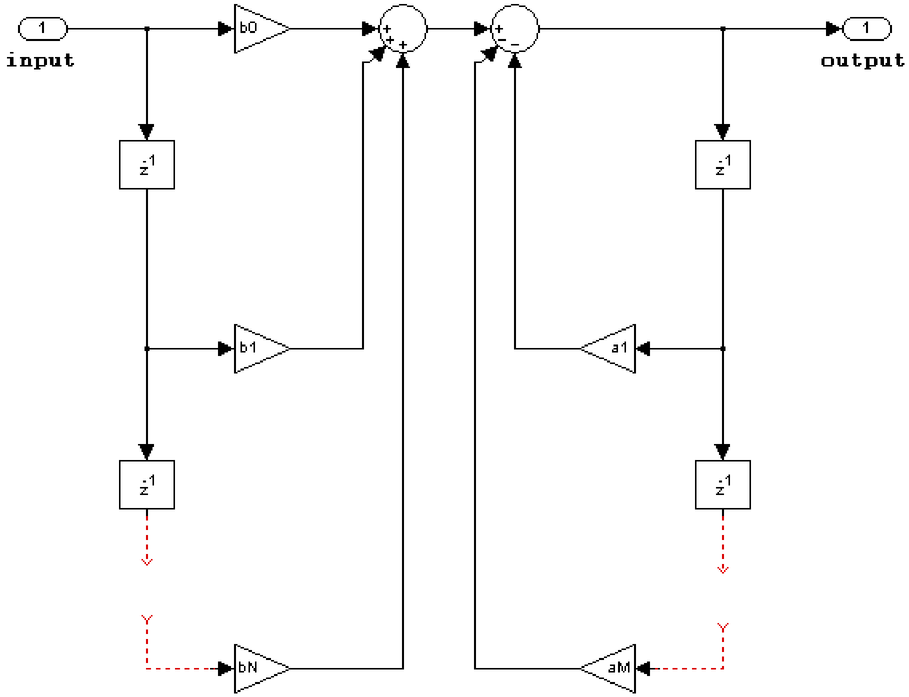
Direct Feedthrough	Only when the leading numerator coefficient does not equal zero
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of initial states
States	See “Specifying Initial States” on page 1-387
Dimensionalized	Yes
Zero-Crossing Detection	No

Filter Structure Diagrams

The diagrams in the following sections show the filter structures supported by the Digital Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block dialog.

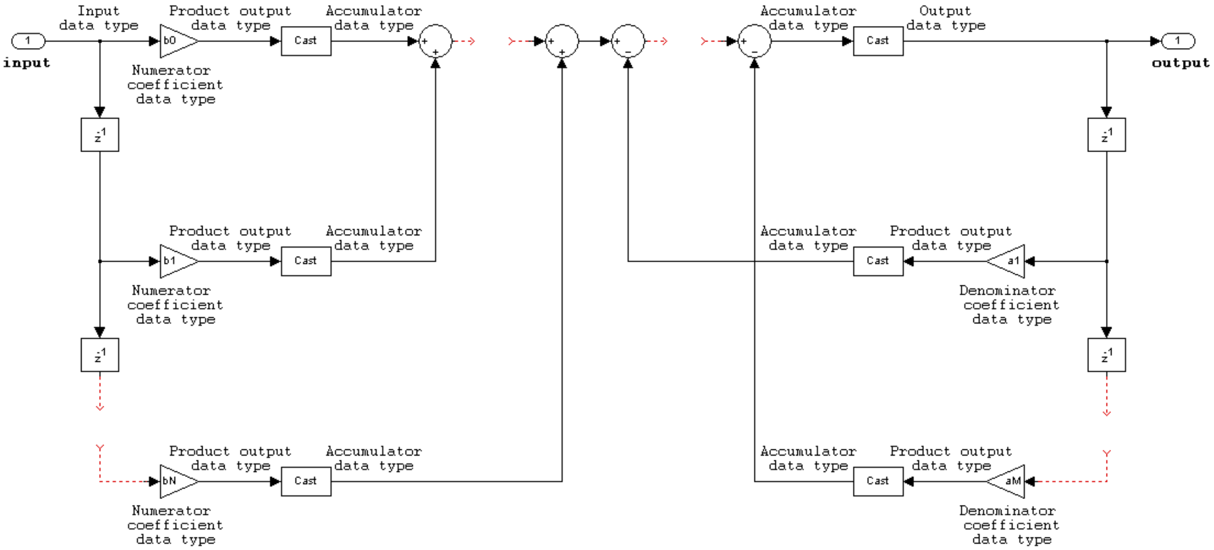
- "IIR direct form I" on page 1-414
- "IIR direct form I transposed" on page 1-417
- "IIR direct form II" on page 1-420
- "IIR direct form II transposed" on page 1-422

IIR direct form I

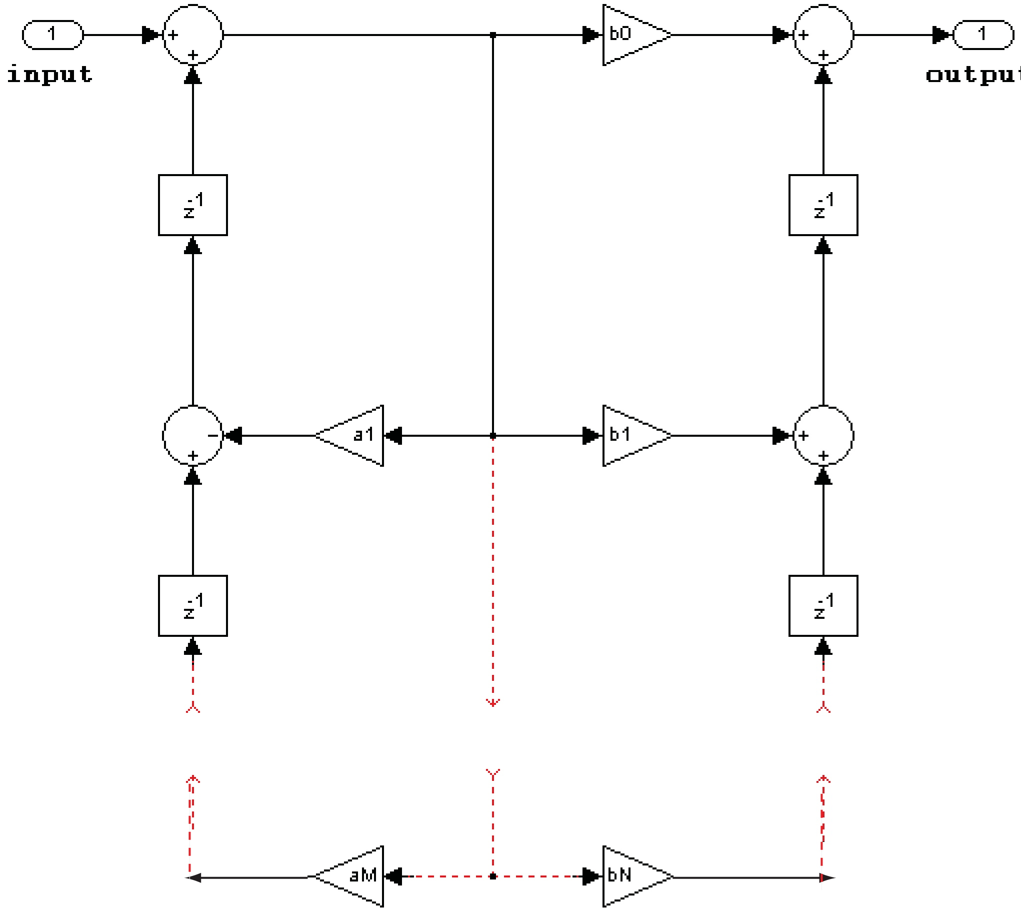


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics.
    - When the numerator and denominator coefficients are specified using input ports and have different complexities from each other, you get an error.
    - When the numerator and denominator coefficients are specified in the dialog box and have different complexities from each other, the block does not error. Instead, it processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.
- The State data type cannot be specified on the block mask for this structure. Doing so is not possible because the input and output states have the same data types as the input and output buffers.

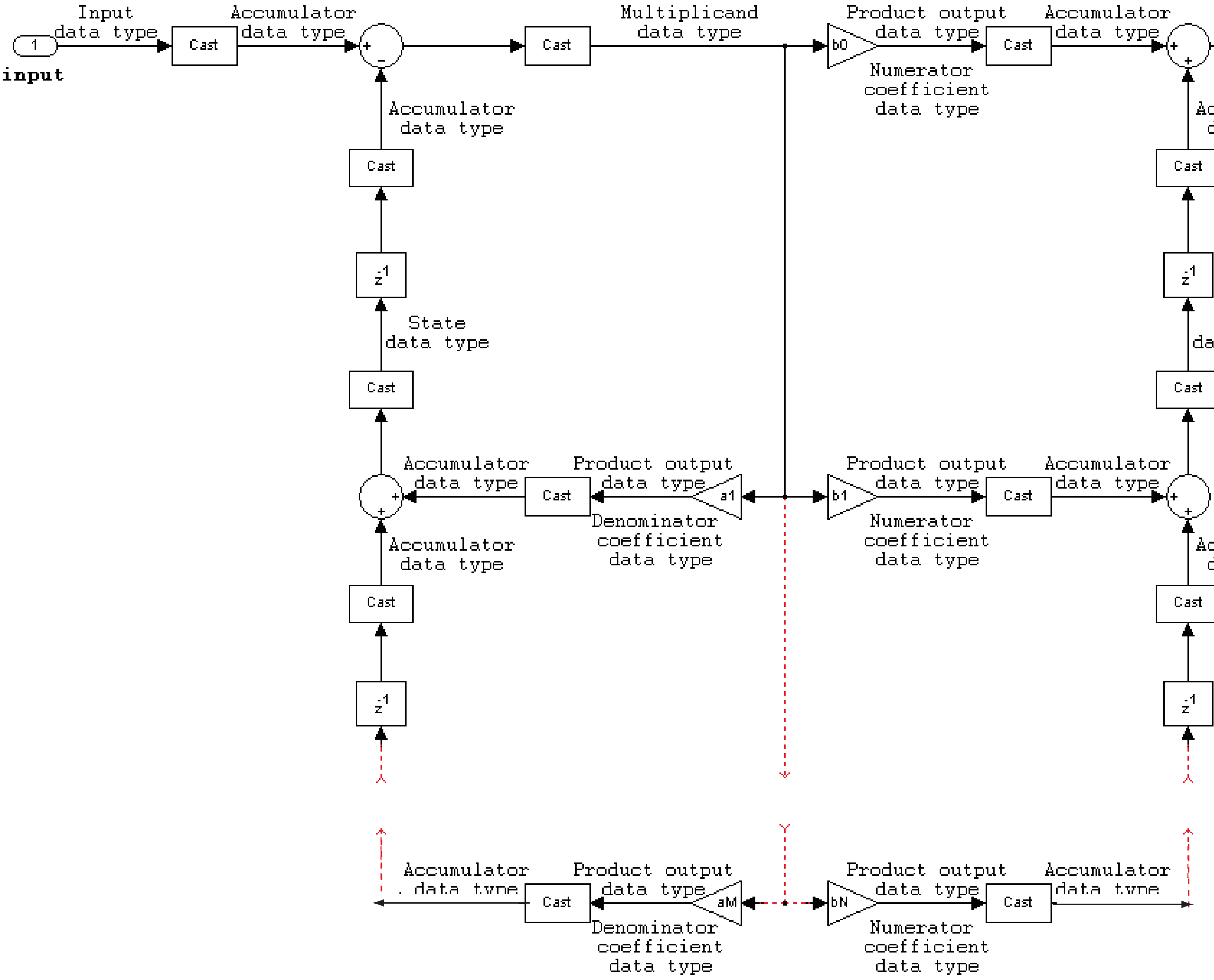

Input
data type
Product output
data type
Accumulator
data type
1
input
b0
Cast
Numerator
coefficient
data type
z-1
b1
Product output
data type
Accumulator
data type
Cast
Numerator
coefficient
data type
z-1
bN
Product output
data type
Accumulator
data type
Cast
Numerator
coefficient
data type
Accumulator
data type
Output
data type
Cast
1
output
z-1
a1
Accumulator
data type
Product output
data type
Cast
Denominator
coefficient
data type
z-1
aM
Accumulator
data type
Product output
data type
Cast
Denominator
coefficient
data type

IIR direct form I transposed

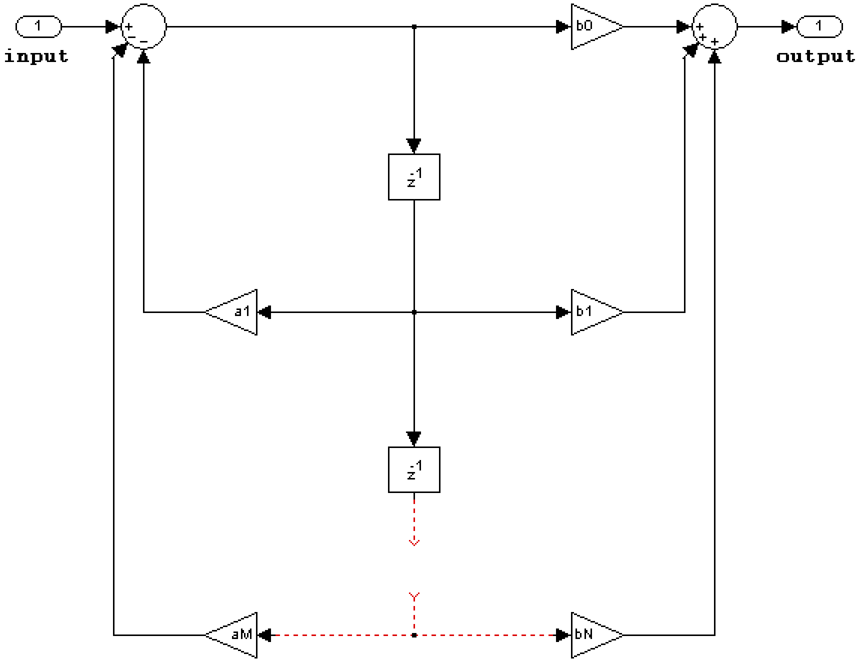

1
input
b0
1
outpu
z-1
z-1
a1
b1
z-1
z-1
aM
bN

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics.
  - When the numerator and denominator coefficients are specified using input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog box and have different complexities from each other, the block does not error. Instead, it processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the input or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.


Input
data type
1
input
Cast
Accumulator
data type
+
–
Cast
Multiplicand
data type
b0
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
Accumulator
data type
Cast
z-1
State
data type
Cast
Accumulator
data type
Product output
data type
Cast
a1
Denominator
coefficient
data type
b1
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
Accumulator
data type
Cast
z-1
Accumulator
data type
Product output
data type
Cast
aM
Denominator
coefficient
data type
bN
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type

IIR direct form II

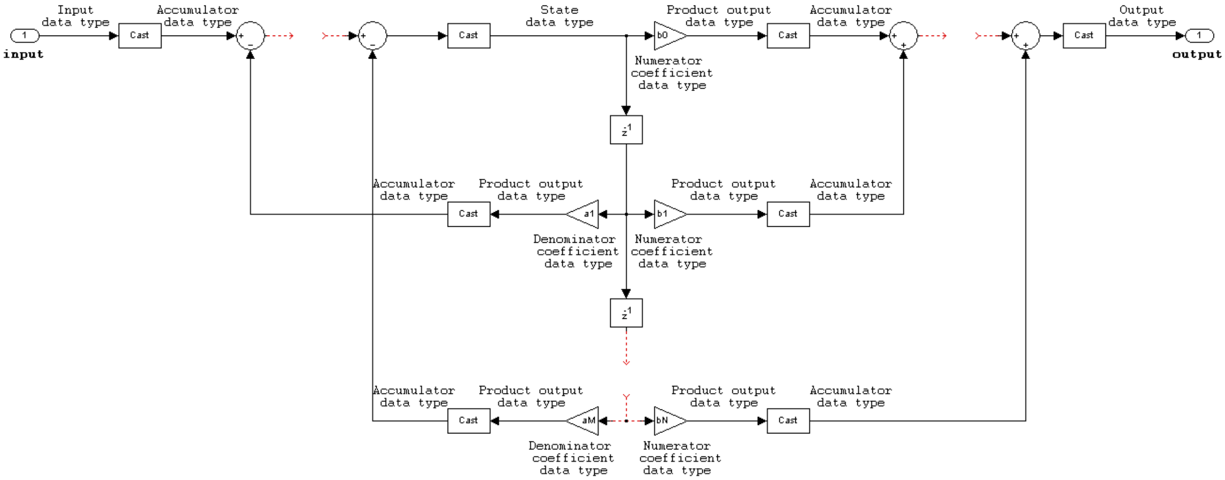


The following constraints are applicable when processing a fixed-point signal with this filter structure:

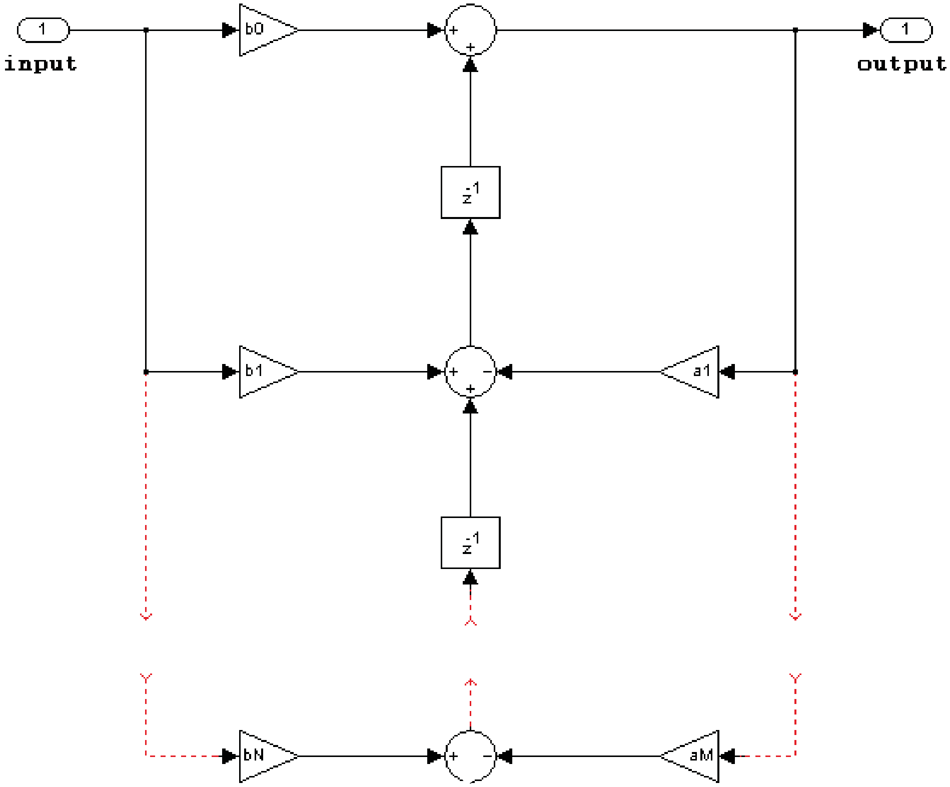
- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics.
  - When the numerator and denominator coefficients are specified using input ports and have different complexities from each other, you get an error.

  - When the numerator and denominator coefficients are specified in the dialog box and have different complexities from each other, the block does not error. Instead, it processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the inputs or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.


Input data type
Accumulator data type
1
input
Cast
State data type
Product output data type
Accumulator data type
Output data type
output
b0
Numerator coefficient data type
z-1
a1
b1
Denominator coefficient data type
aM
bN

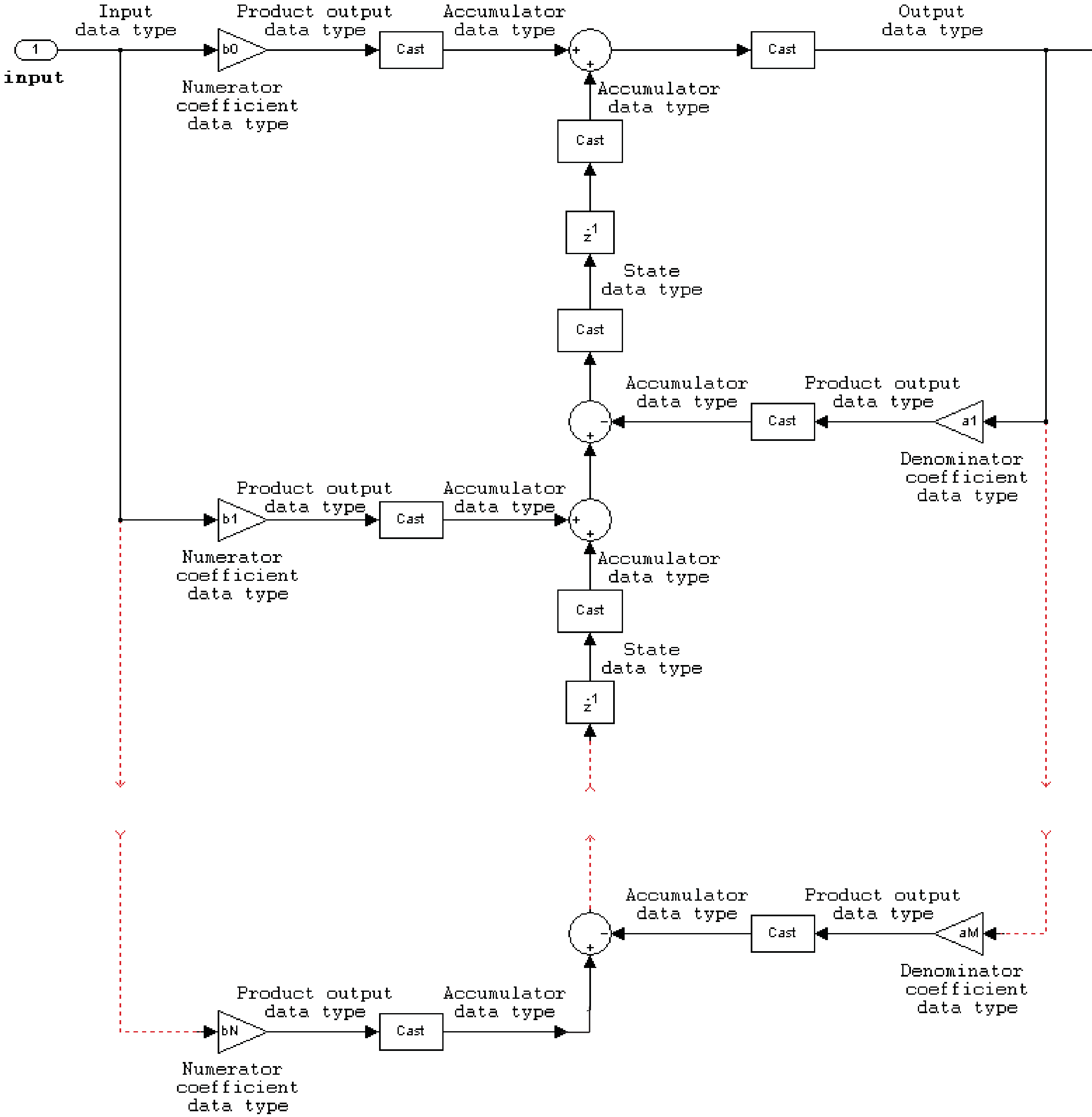
IIR direct form II transposed


1
input
b0
+
+
z⁻¹
b1
+
−
+
a1
z⁻¹
bN
+
−
aM
1
output

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics.
  - When the numerator and denominator coefficients are specified using input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog box and have different complexities from each other, the block does not error. Instead, it processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the inputs or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.


Input
data type
1
input
Product output
data type
b0
Numerator
coefficient
data type
Cast
Accumulator
data type
+
+
Cast
Output
data type
Accumulator
data type
Cast
z-1
State
data type
Cast
Accumulator
data type
Product output
data type
Cast
a1
-
+
Denominator
coefficient
data type
Product output
data type
b1
Numerator
coefficient
data type
Cast
Accumulator
data type
+
+
Accumulator
data type
Cast
State
data type
z-1
Accumulator
data type
Product output
data type
Cast
aM
-
+
Denominator
coefficient
data type
Product output
data type
bN
Cast
Accumulator
data type
Numerator
coefficient
data type

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

Allpole Filter	DSP System Toolbox
Digital Filter Design	DSP System Toolbox
Digital Filter	DSP System Toolbox
Discrete FIR Filter	Simulink
Filter Realization Wizard	DSP System Toolbox
`dfilt`	DSP System Toolbox
`fdatool`	DSP System Toolbox
`fvtool`	Signal Processing Toolbox

Discrete FIR Filter

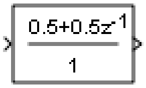
Purpose

Model FIR filters

Library

Discrete

Description



The Discrete FIR Filter block independently filters each channel of the input signal with the specified digital FIR filter. The block can implement static filters with fixed coefficients, as well as time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an independent channel (sample-based processing), or each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal those of the input, except when you specify a matrix of filter taps for the **Coefficients** parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

The outputs of this block numerically match the outputs of the DSP System Toolbox Digital Filter Design block and of the Signal Processing Toolbox™ `dfilt` object.

This block supports the Simulink state logging feature. See “States” in the *Simulink User’s Guide* for more information.

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.

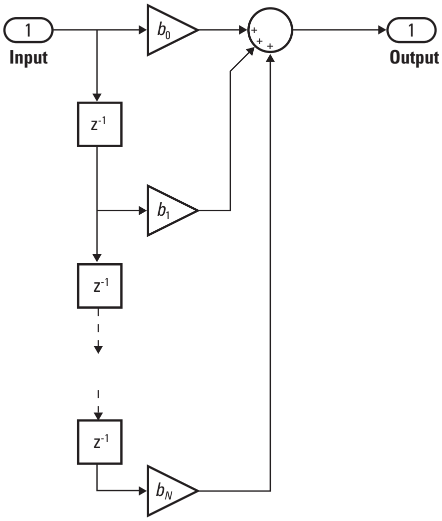
Data Type Support

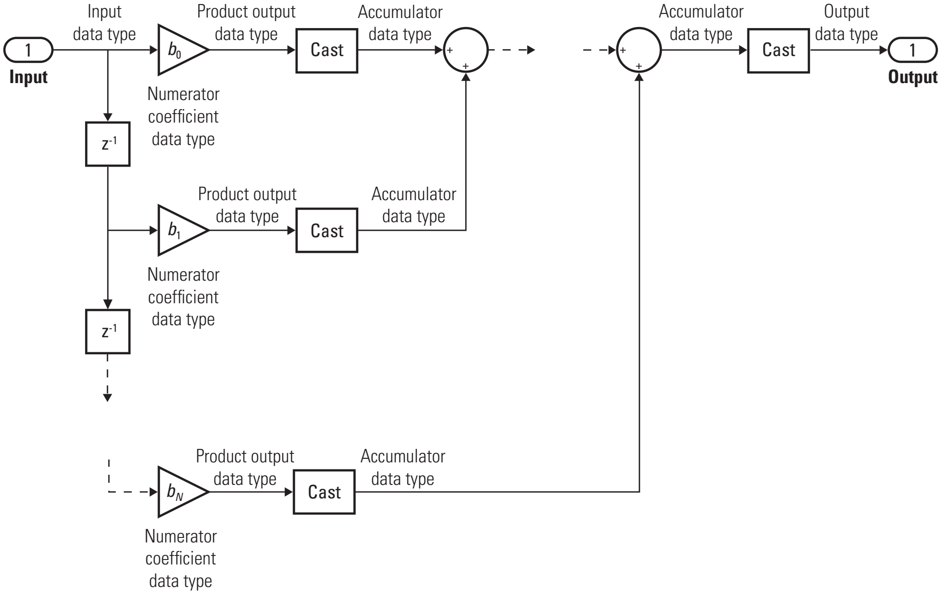
The Discrete FIR Filter block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.

The following diagrams show the filter structure and the data types used within the Discrete FIR Filter block for fixed-point signals.

Direct Form

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.

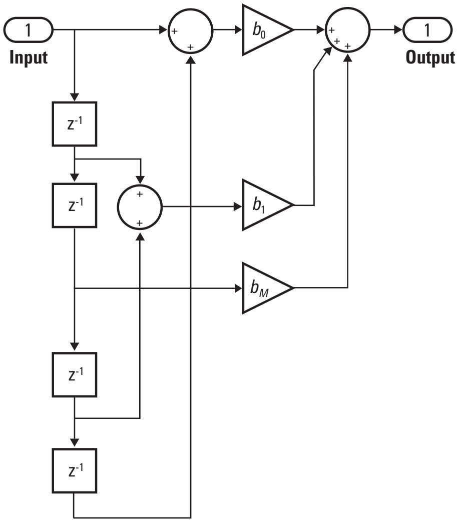



Input
data type
Product output
data type
Accumulator
data type
Accumulator
data type
Output
data type
1
Input
b_0
Cast
+
+
+
+
Cast
1
Output
Numerator
coefficient
data type
z-1
Product output
data type
Accumulator
data type
b_1
Cast
Numerator
coefficient
data type
z-1
Product output
data type
Accumulator
data type
b_N
Cast
Numerator
coefficient
data type

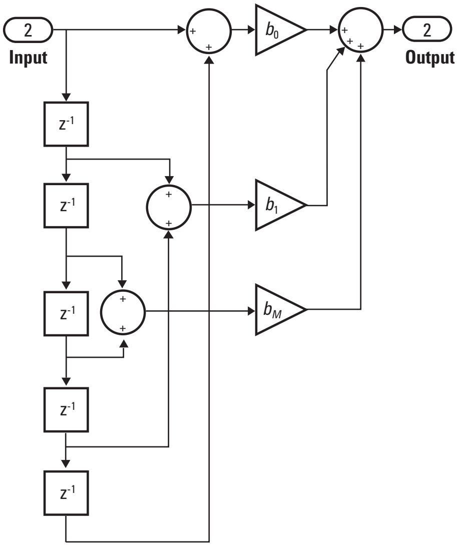
Direct Form Symmetric

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.

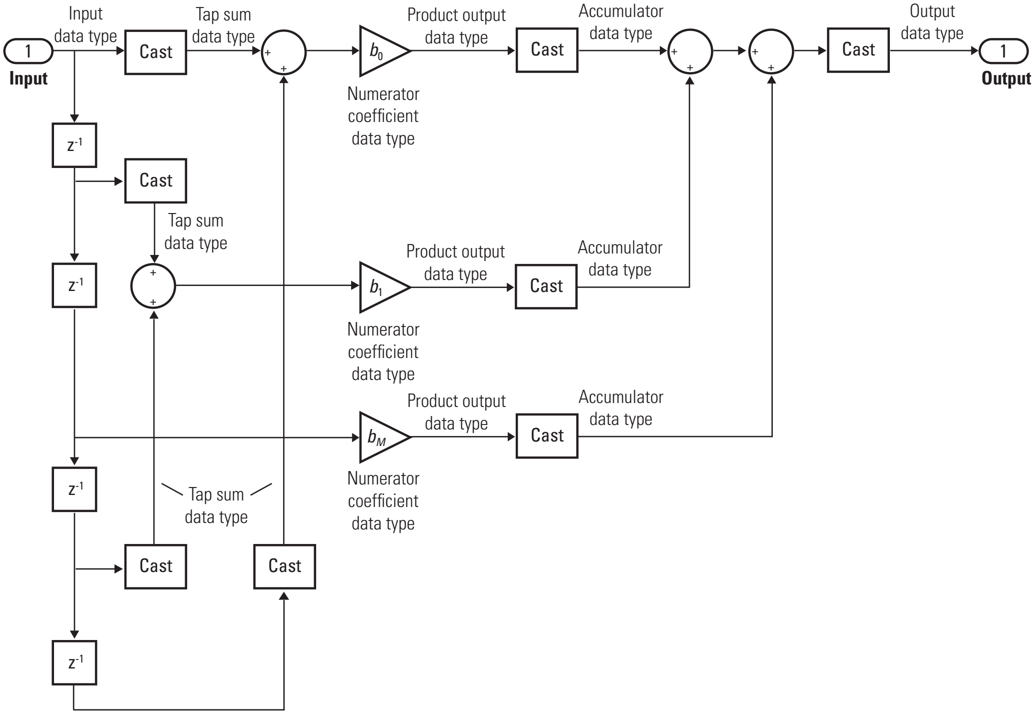
It is assumed that the filter coefficients are symmetric. The block only uses the first half of the coefficients for filtering.


1
Input
b_0
1
Output
z^{-1}
z^{-1}
b_1
b_M
z^{-1}
z^{-1}

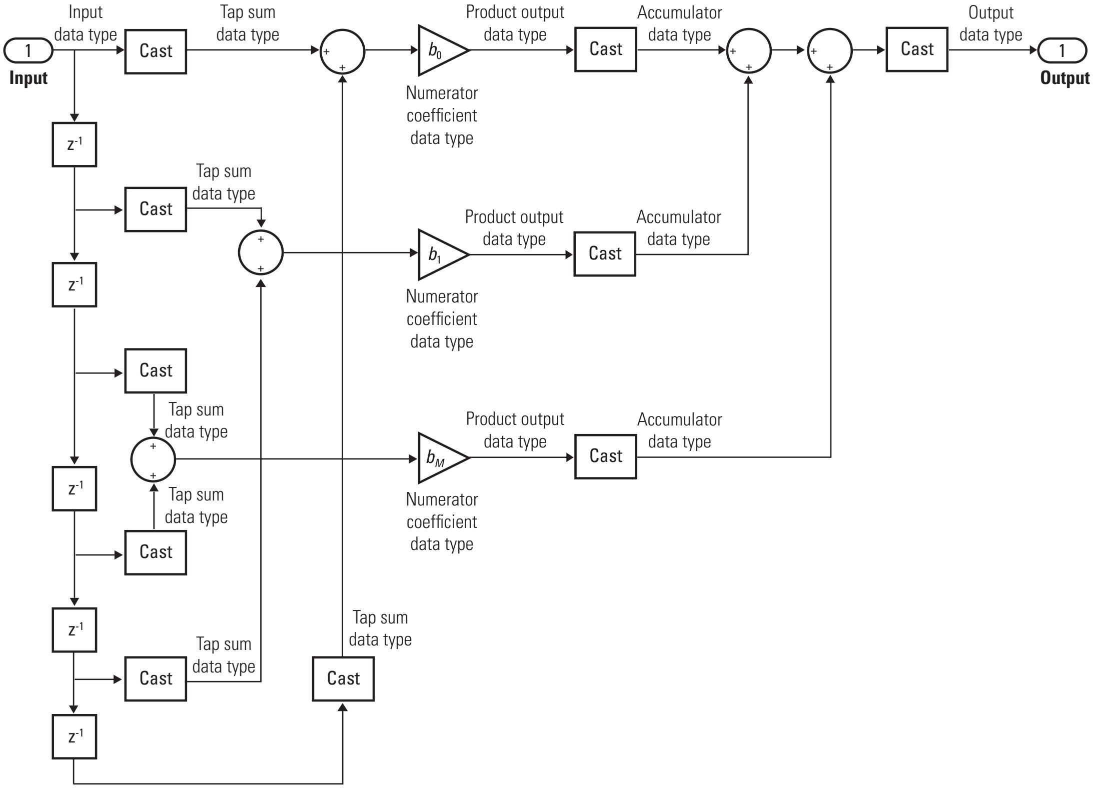
Even Order - Type I


2
Input
b_0
2
Output
z^{-1}
z^{-1}
b_1
z^{-1}
b_M
z^{-1}
z^{-1}

Odd Order - Type II


Input
data type
1
Input
Cast
Tap sum
data type
+
+
b_0
Product output
data type
Cast
Accumulator
data type
+
+
+
+
Cast
Output
data type
1
Output
Numerator
coefficient
data type
z^{-1}
Cast
Tap sum
data type
z^{-1}
+
+
b_1
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
b_M
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
z^{-1}
Tap sum
data type
Cast
Cast
z^{-1}

Even Order - Type I

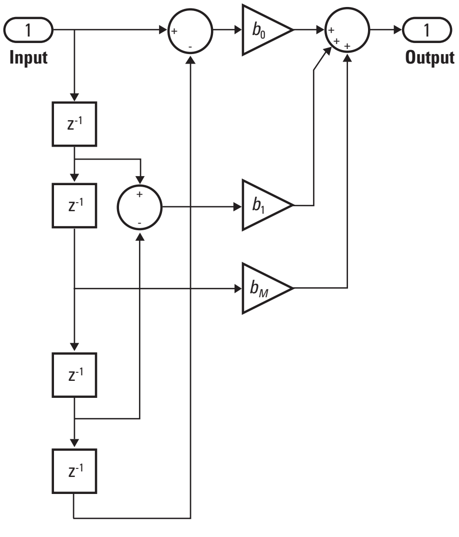

Input
data type
1
Input
Cast
Tap sum
data type
b_0
Product output
data type
Cast
Accumulator
data type
Cast
Output
data type
1
Output
Numerator
coefficient
data type
z^{-1}
Cast
Tap sum
data type
b_1
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
z^{-1}
Cast
Tap sum
data type
b_M
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
z^{-1}
Tap sum
data type
Cast
z^{-1}
Tap sum
data type
Cast
Tap sum
data type
Cast
z^{-1}

Odd Order - Type II

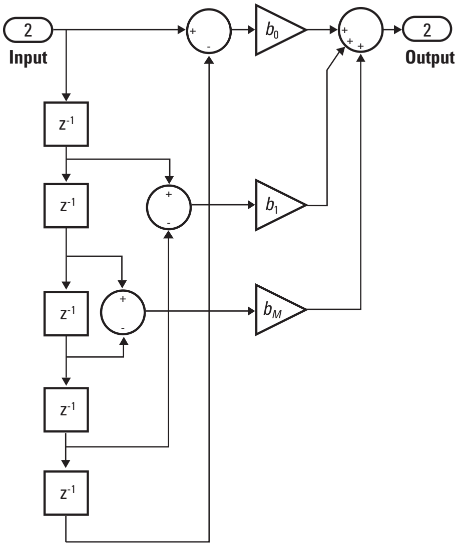
Direct Form Antisymmetric

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.

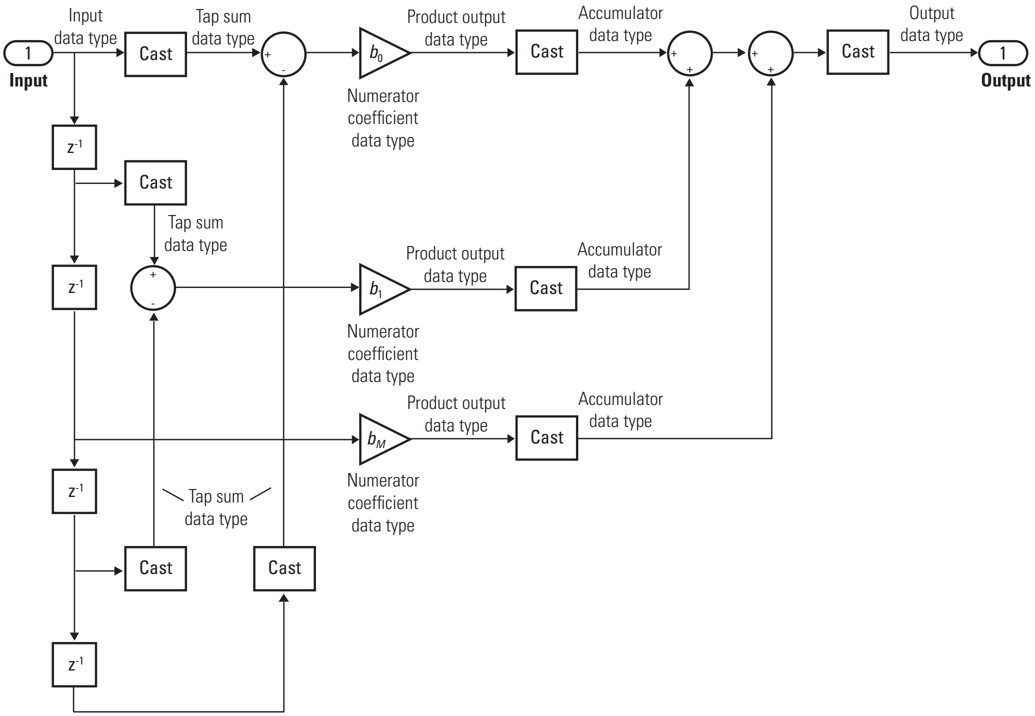
It is assumed that the filter coefficients are antisymmetric. The block only uses the first half of the coefficients for filtering.


1
Input
z^{-1}
z^{-1}
z^{-1}
z^{-1}
b_0
b_1
b_M
1
Output

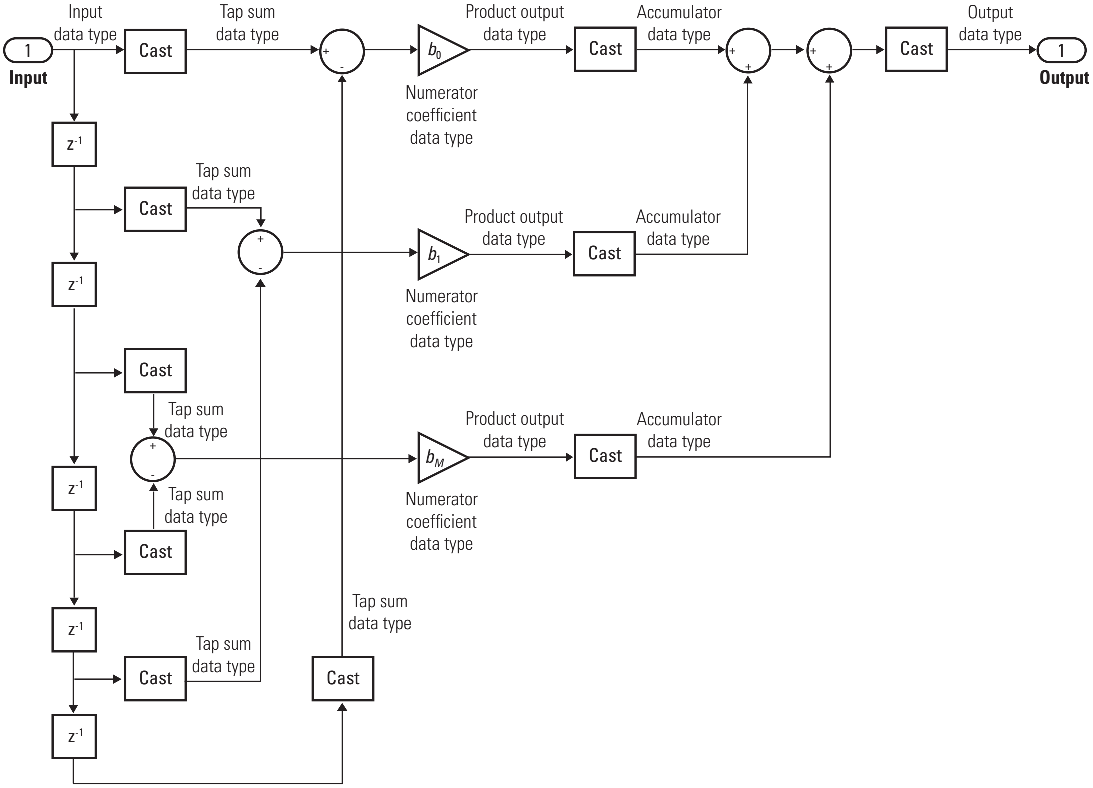
Even Order - Type III


2
Input
z^{-1}
z^{-1}
z^{-1}
z^{-1}
z^{-1}
b_0
b_1
b_M
2
Output

Odd Order - Type IV


Input
data type
1
Input
Cast
Tap sum
data type
b_0
Product output
data type
Cast
Accumulator
data type
Cast
Output
data type
1
Output
Numerator
coefficient
data type
z^{-1}
Cast
Tap sum
data type
z^{-1}
b_1
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
b_M
Product output
data type
Cast
Accumulator
data type
z^{-1}
Tap sum
data type
Numerator
coefficient
data type
Cast
Cast
z^{-1}

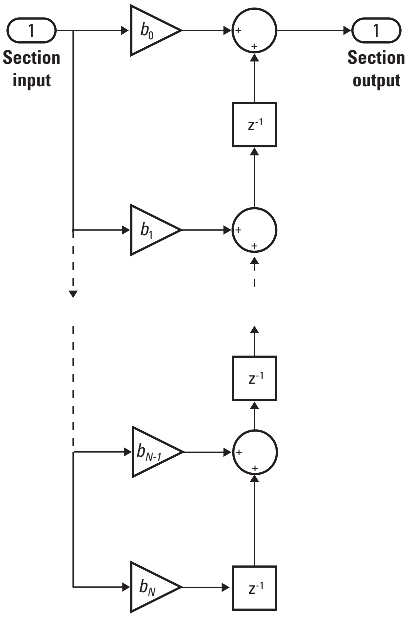
Even Order - Type III

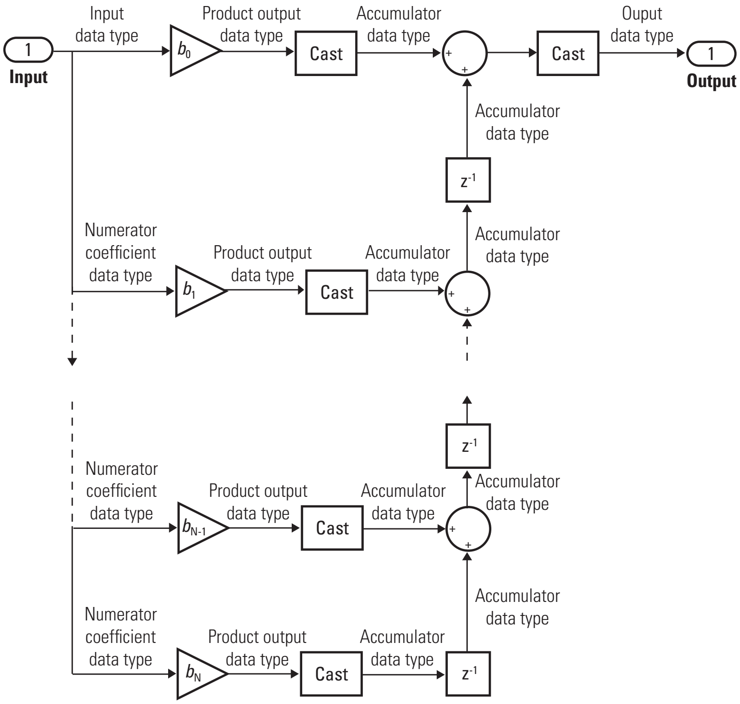

Input
data type
1
Input
Cast
Tap sum
data type
+
-
b_0
Product output
data type
Cast
Accumulator
data type
+
+
+
+
Cast
Output
data type
1
Output
Numerator
coefficient
data type
z^{-1}
Cast
Tap sum
data type
+
-
b_1
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
z^{-1}
Cast
Tap sum
data type
+
-
b_M
Product output
data type
Cast
Accumulator
data type
Numerator
coefficient
data type
z^{-1}
Tap sum
data type
Cast
z^{-1}
Tap sum
data type
Cast
Tap sum
data type
Cast
z^{-1}

Odd Order - Type IV

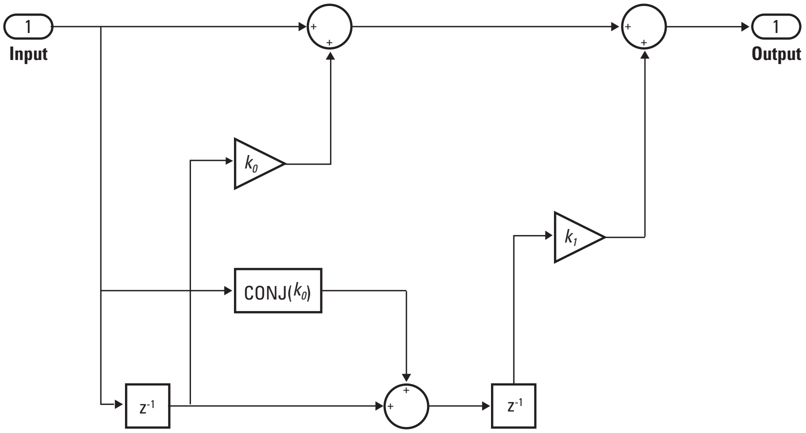
Direct Form Transposed

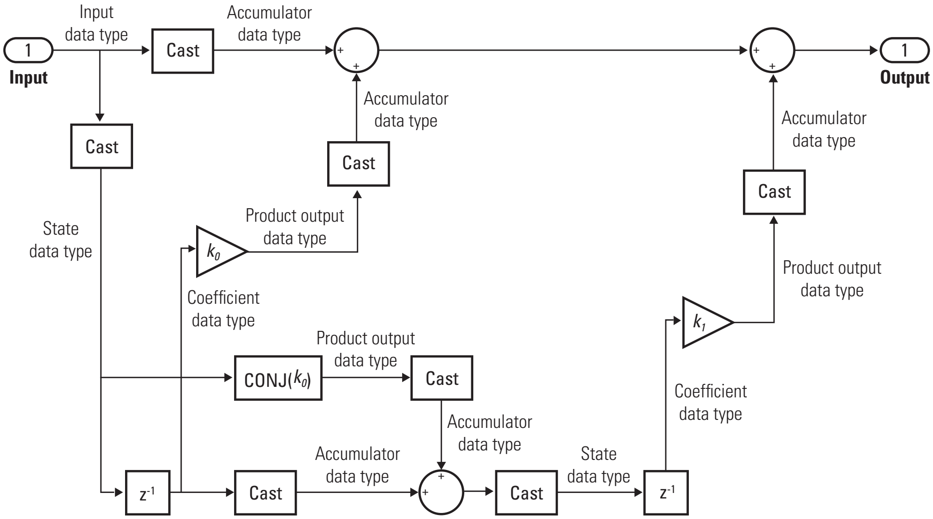
States are complex when either the inputs or the coefficients are complex.


1
Section input
b_0
+
+
1
Section output
z^{-1}
b_1
+
+
z^{-1}
b_{N-1}
+
+
b_N
z^{-1}


Input
data type
1
Input
b_0
Product output
data type
Cast
Accumulator
data type
+
+
Cast
Ouput
data type
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
Accumulator
data type
Numerator
coefficient
data type
b_N
Product output
data type
Cast
Accumulator
data type
z-1

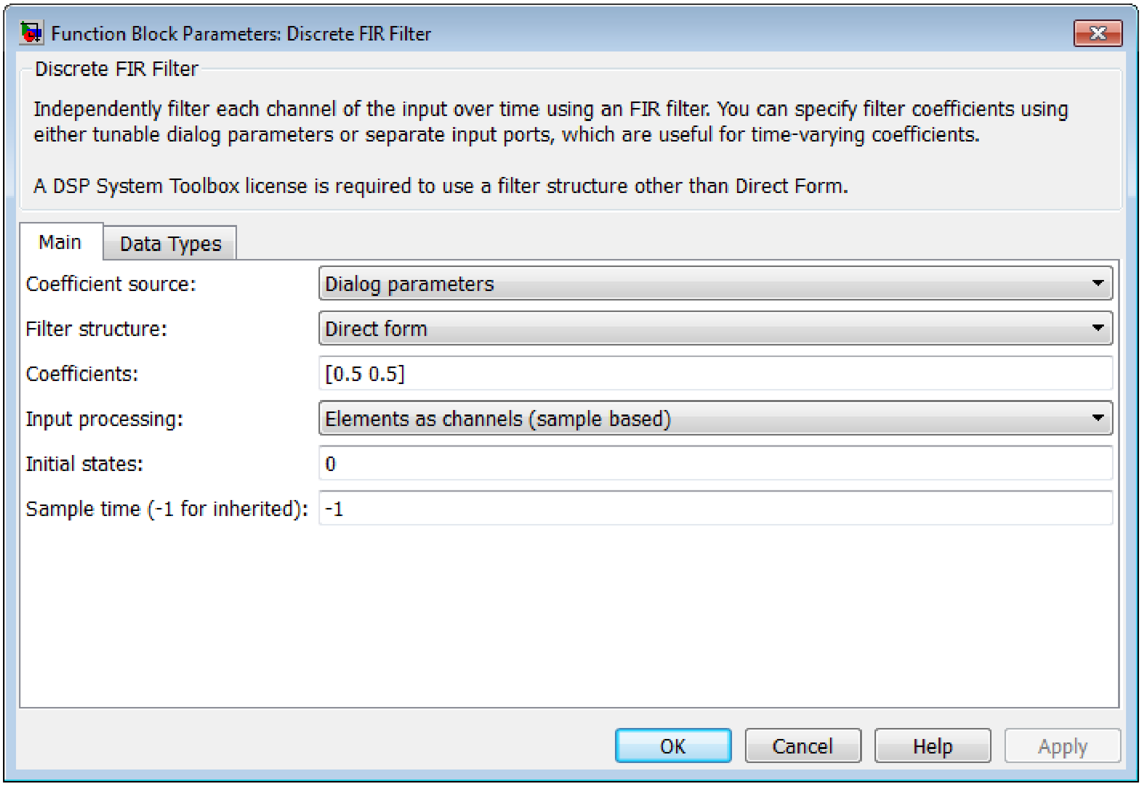
Lattice MA


1
Input
k_0
CONJ(k_0)
z^{-1}
z^{-1}
k_1
1
Output


1
Input
Input
data type
Cast
Accumulator
data type
Cast
State
data type
Accumulator
data type
Cast
Product output
data type
k_0
Coefficient
data type
Product output
data type
CONJ(k_0)
Cast
Accumulator
data type
z^{-1}
Cast
Accumulator
data type
Cast
State
data type
z^{-1}
k_1
Coefficient
data type
Product output
data type
Cast
Accumulator
data type
1
Output

Parameters and Dialog Box

The **Main** pane of the Discrete FIR Filter block dialog box appears as follows.



Discrete FIR Filter

Independently filter each channel of the input over time using an FIR filter. You can specify filter coefficients using either tunable dialog parameters or separate input ports, which are useful for time-varying coefficients.

A DSP System Toolbox license is required to use a filter structure other than Direct Form.

Coefficient source: Dialog parameters

Filter structure: Direct form

Coefficients: [0.5 0.5]

Input processing: Elements as channels (sample based)

Initial states: 0

Sample time (-1 for inherited): -1

Coefficient source

Select whether you want to specify the filter coefficients on the block mask or through an input port.

Filter structure

Select the filter structure you want the block to implement. You must have an available DSP System Toolbox license to run a model with a Discrete FIR Filter block that implements any filter structure other than direct form.

Coefficients

Specify the vector coefficients of the filter's transfer function. Filter coefficients must be specified as a row vector. When you specify a row vector of filter taps, the block applies a single filter to the input. To apply multiple filters to the same input, specify a matrix of coefficients, where each row represents a different set of filter taps. This parameter is visible only when **Coefficient source** is set to `Dialog parameters` . For multiple filter, **Filter structure** must be `Direct form`, and the input must be a scalar.

Input processing

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

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

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

For more information, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

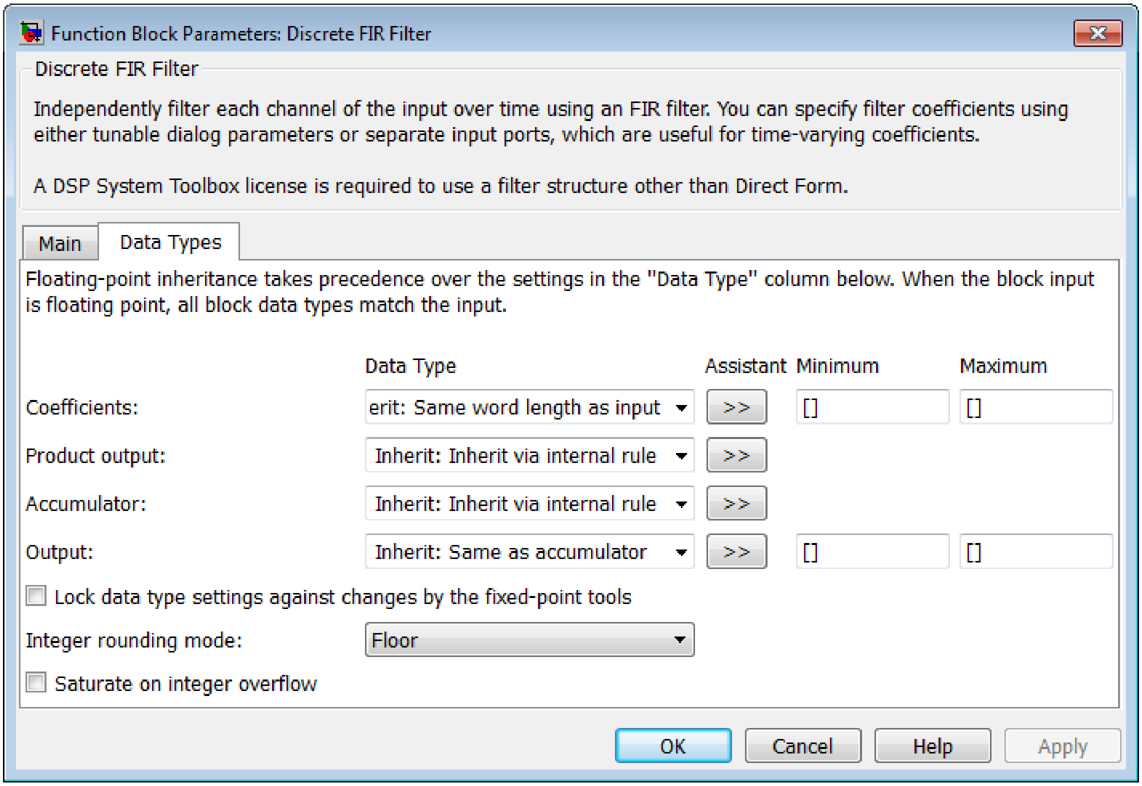
Initial states

Specify the initial conditions of the filter states. To learn how to specify initial states, see "Specifying Initial States" on page 1-427.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in “How Simulink Works” in the *Simulink User’s Guide*.

The **Data Types** pane of the Discrete FIR Filter block dialog box appears as follows.



Function Block Parameters: Discrete FIR Filter

Discrete FIR Filter

Independently filter each channel of the input over time using an FIR filter. You can specify filter coefficients using either tunable dialog parameters or separate input ports, which are useful for time-varying coefficients.

A DSP System Toolbox license is required to use a filter structure other than Direct Form.

Floating-point inheritance takes precedence over the settings in the "Data Type" column below. When the block input is floating point, all block data types match the input.

	Data Type	Assistant	Minimum	Maximum
Coefficients:	erit: Same word length as input	>>	[]	[]
Product output:	Inherit: Inherit via internal rule	>>		
Accumulator:	Inherit: Inherit via internal rule	>>		
Output:	Inherit: Same as accumulator	>>	[]	[]

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

Integer rounding mode: Floor

☐ Saturate on integer overflow

Tap sum

Specify the tap sum data type of a direct form symmetric or direct form antisymmetric filter, which is the data type the filter uses when it sums the inputs prior to multiplication by the coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is either `Direct form symmetric` or `Direct form antisymmetric`.

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

See "Specify Data Types Using Data Type Assistant" for more information.

Coefficients

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

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Coefficients minimum

Specify the minimum value that a filter coefficient should have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Coefficients maximum

Specify the maximum value that a filter coefficient should have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Product output

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

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

Accumulator

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

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” for more information.

State

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

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is `Lattice MA`.

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

See “Specify Data Types Using Data Type Assistant” for more information.

Output

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

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Block Output Data Types” in the *Simulink User’s Guide* for more information.

Output minimum

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

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

Output maximum

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

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

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

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Locking the Output Data Type Setting"in the Fixed-Point Designer documentation.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see "Rounding". in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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,	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.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
	see “Checking 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. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

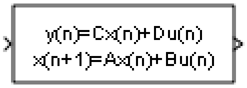
Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of initial states
States	See “Specifying Initial States” on page 1-427
Dimensionalized	Yes
Zero-Crossing Detection	No

Discrete State-Space

Purpose Implement discrete state-space system

Library Discrete

Description



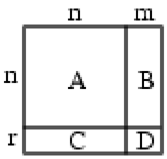
Block Behavior for Non-Empty Matrices

The Discrete State-Space block implements the system described by

$$\begin{aligned} x(n+1) &= Ax(n) + Bu(n) \\ y(n) &= Cx(n) + Du(n), \end{aligned}$$

where u is the input, x is the state, and y is the output. The matrix coefficients must have these characteristics, as illustrated in the following diagram:

- **A** must be an n-by-n matrix, where n is the number of states.
- **B** must be an n-by-m matrix, where m is the number of inputs.
- **C** must be an r-by-n matrix, where r is the number of outputs.
- **D** must be an r-by-m matrix.



The block accepts one input and generates one output. The width of the input vector is the number of columns in the **B** and **D** matrices. The width of the output vector is the number of rows in the **C** and **D** matrices. To define the initial state vector, use the **Initial conditions** parameter.

To specify a vector or matrix of zeros for **A**, **B**, **C**, **D**, or **Initial conditions**, use the `zeros` function.

Block Behavior for Empty Matrices

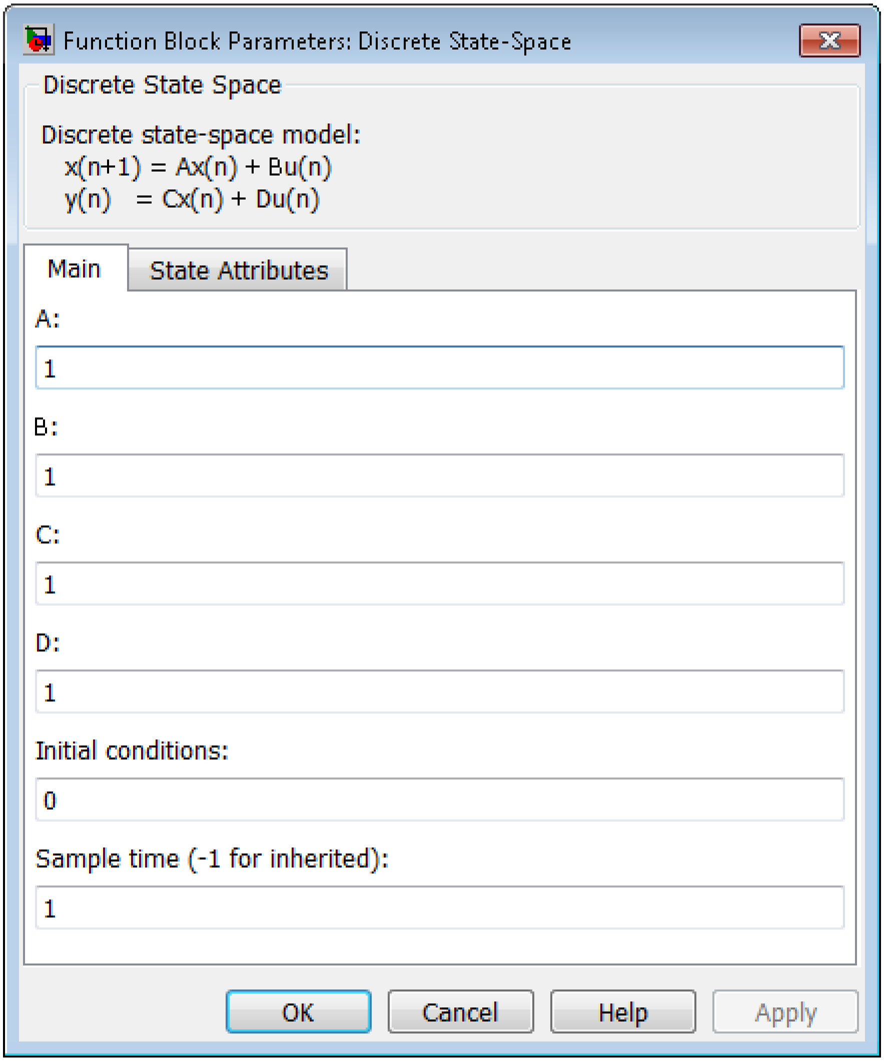
When the matrices **A**, **B**, and **C** are empty (for example, `[]`), the functionality of the block becomes `y(n) = Du(n)`. If the **Initial conditions** vector is also empty, the block uses an initial state vector of zeros.

Data Type Support

The Discrete State Space block accepts and outputs a real signal of type `single` or `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** tab of the Discrete State-Space block dialog box appears as follows:



Function Block Parameters: Discrete State-Space

Discrete State Space

Discrete state-space model:
x(n+1) = Ax(n) + Bu(n)
y(n)   = Cx(n) + Du(n)

Main
State Attributes

A: 1

B: 1

C: 1

D: 1

Initial conditions: 0

Sample time (-1 for inherited): 1

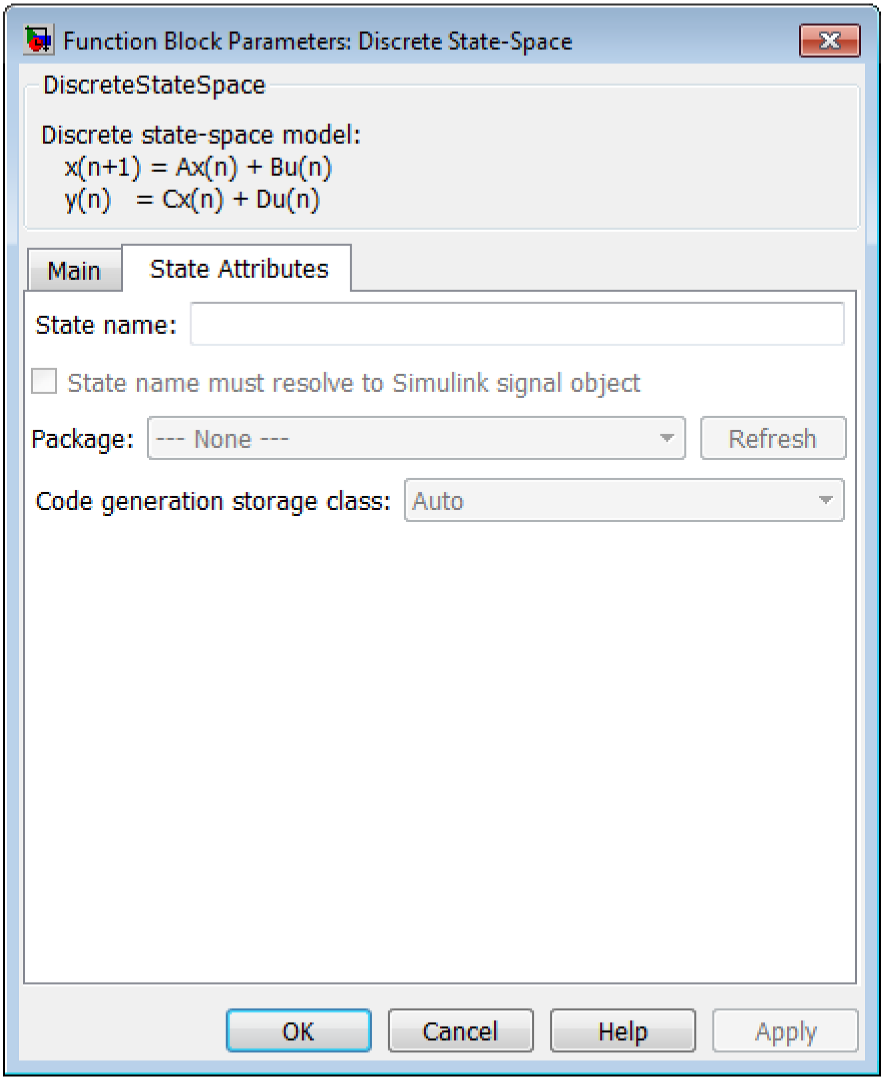
OK
Cancel
Help
Apply

A, B, C, D
Specify the matrix coefficients, as defined in the Description section.

Initial conditions
Specify the initial state vector. The default value is `0`. Simulink does not allow the initial states of this block to be `inf` or `NaN`.

Sample time (–1 for inherited)
Specify the time interval between samples. See "Specify Sample Time" in the Simulink documentation.

The **State Attributes** tab of the Discrete State-Space block dialog box appears as follows:



Function Block Parameters: Discrete State-Space

DiscreteStateSpace

Discrete state-space model:
x(n+1) = Ax(n) + Bu(n)
y(n)   = Cx(n) + Du(n)

Main
State Attributes

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Refresh

Code generation storage class: Auto

OK
Cancel
Help
Apply

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “States” in the Simulink Coder documentation.

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Package

Select a package that defines the custom storage class you want to apply. If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Code generation storage class

From the list, select a state storage class.

`Auto`

Auto is the storage class to use for states that do not need to interface to external code.

`ExportedGlobal`

The state is stored in a global variable.

`ImportedExtern`
`model``_private.h` declares the state as an extern variable.

`ImportedExternPointer`
`model``_private.h` declares the state as an extern pointer.

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Code generation storage type qualifier

Specify a Simulink Coder storage type qualifier. The default is `' '`. When this field is blank, no qualifier is assigned.

The Simulink Coder product does not check this string for errors. Thus, whatever value you enter appears automatically in the variable declaration.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

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 “States” in the Simulink Coder documentation.

Characteristics

Direct Feedthrough	Only if $D \neq 0$
Sample Time	Specified in the **Sample time** parameter

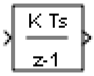
Scalar Expansion	Yes, of the initial conditions
States	Determined by the size of A
Dimensionalized	Yes
Zero-Crossing Detection	No

Discrete-Time Integrator

Purpose Perform discrete-time integration or accumulation of signal

Library Discrete

Description



Capabilities of the Discrete-Time Integrator Block

You can use the Discrete-Time Integrator block in place of the Integrator block to create a purely discrete system. With the Discrete-Time Integrator block, you can:

- Define initial conditions on the block dialog box or as input to the block.
- Define an input gain (K) value.
- Output the block state.
- Define upper and lower limits on the integral.
- Reset the state depending on an additional reset input.

Integration and Accumulation Methods

The block can integrate or accumulate using the Forward Euler, Backward Euler, and Trapezoidal methods. Assume that `u` is the input, `y` is the output, and `x` is the state. For a given step `n`, Simulink updates `y(n)` and `x(n+1)`. In integration mode, `T` is the block sample time (delta `T` in the case of triggered sample time). In accumulation mode, `T = 1`; the block sample time determines when the output is computed but not the output value. `K` is the gain value. Values clip according to upper or lower limits.

- Forward Euler method (the default), also known as Forward Rectangular, or left-hand approximation.

  For this method, `1/s` is approximated by `T/(z-1)`. The resulting expression for the output of the block at step `n` is:

  ```
  y(n) = y(n-1) + K*T*u(n-1)
  ```

Let `x(n+1) = x(n) + K*T*u(n)`. The block uses the following steps to compute its output:

```
Step 0:    y(0)   = x(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)
```

With this method, input port 1 does not have direct feedthrough.

- Backward Euler method, also known as Backward Rectangular or right-hand approximation.

  For this method, `1/s` is approximated by `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 the following steps to compute its output

  ```
  Step 0:   y(0)   = x(0) = IC (clipped if necessary)
     x(1)   = y(0)
  ```

  or, depending on **Use initial condition as initial and reset value for** parameter:

  ```
  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)
  ```

With this method, input port 1 has direct feedthrough.

- Trapezoidal method. For this method, `1/s` is approximated by

  ```
  T/2*(z+1)/(z-1)
  ```

  When `T` is fixed (equal to the sampling period), let

  ```
  x(n) = y(n-1) + K*T/2 * u(n-1)
  ```

  The block uses the following steps to compute its output

  ```
  Step 0:   x(0) = IC (clipped if necessary)
     x(1)   = y(0) + K*T/2 * u(0)
  ```

  or, depending on **Use initial condition as initial and reset value for** parameter:

  ```
  Step 0:   y(0)   = x(0) = IC (clipped if necessary)
     x(1)   = y(0) = x(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 the sense that `x(n) != y(n)`.

  If `T` is variable (for example, obtained from the triggering times), the block uses the following algorithm to compute the outputs

  ```
  Step 0:   y(0)   = x(0) = IC (clipped if necessary)
     x(1)   = y(0)
  ```

  or, depending on **Use initial condition as initial and reset value for** parameter:

  ```
  Step 0:   y(0)   = x(0) = IC (clipped if necessary)
  ```

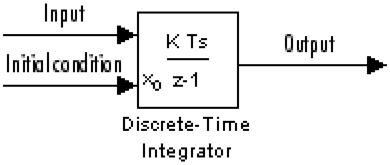
```
   x(1)   = y(0) = x(0) + K*T/2*u(0)

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)
```

With this method, input port 1 has direct feedthrough.

How to 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, specify the **Initial condition source** parameter as `internal` and enter the value in the **Initial condition** parameter 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:



When to Use the State Port

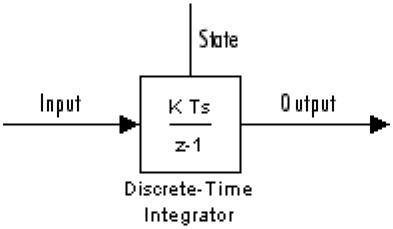
In two situations, you must use the state port instead of the output port:

- 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 of this situation, 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 of this situation, 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. You output the block state by selecting the **Show state port** check box.

By default, the state port appears on the top of the block:

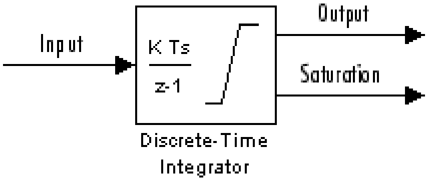


How to Limit the Integral

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. Doing so 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...	The output is...
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 saturation port appears below the block output port:

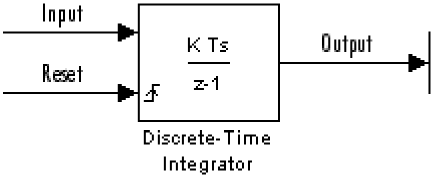


The signal has one of three values:

- 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.

How to Reset the State

The block can reset its state to the initial condition you specify, based on an external signal. To cause the block to reset its state, select one of the **External reset** parameter choices. A trigger port appears below the block input port to indicate the trigger type:



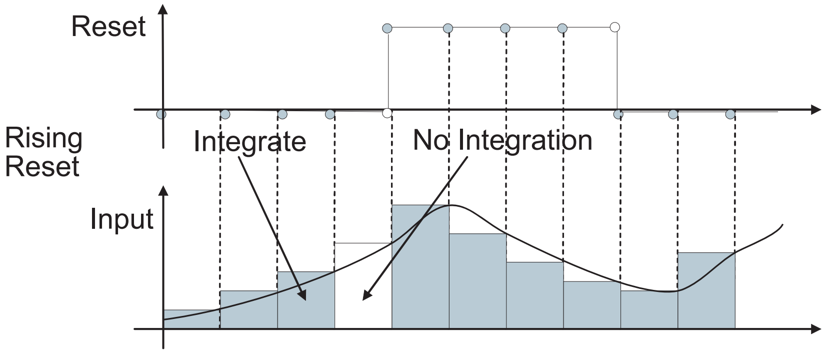
The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results. To resolve this loop, feed the output of the block state port into the reset port instead. To access the block state, 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:

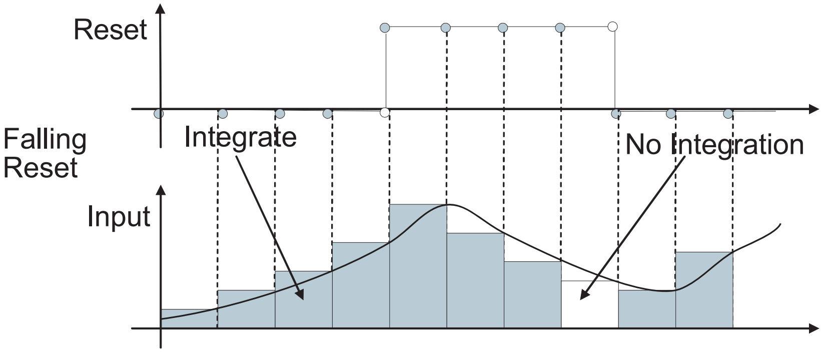
- `rising`

  Resets the state when the reset signal has a rising edge. For example, the following figure shows the effect that a rising reset trigger has on backward Euler integration.

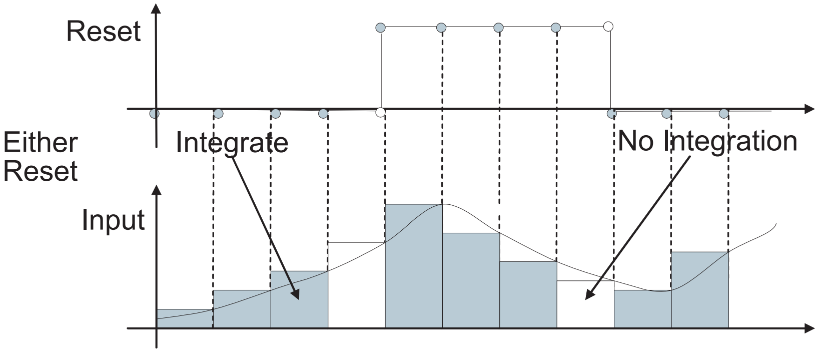
- `falling`

Resets the state when the reset signal has a falling edge. For example, the following figure shows the effect that a falling reset trigger has on backward Euler integration.


Reset
Falling
Reset
Integrate
No Integration
Input

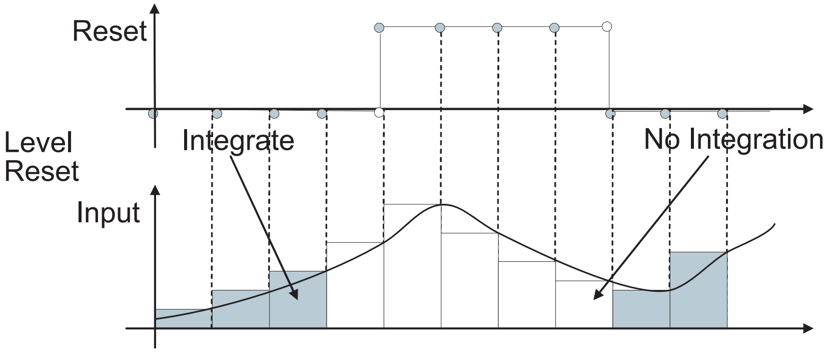
- `either`

  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.


Reset
Either
Reset
Integrate
No Integration
Input

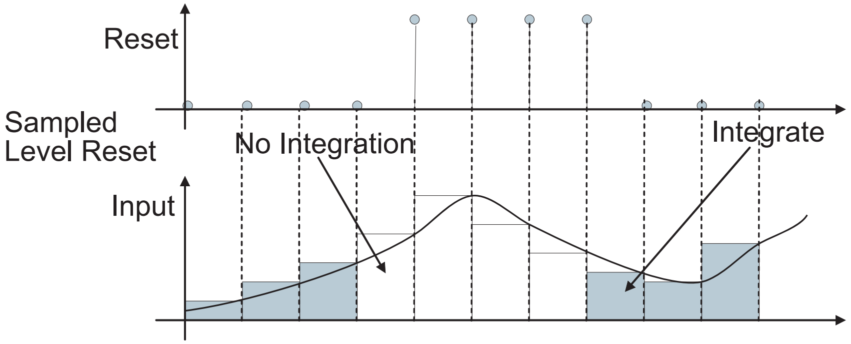
- `level`

Resets and holds the output to the initial condition while the reset signal is nonzero. For example, the following figure shows the effect that a level reset trigger has on backward Euler integration.


Reset
Level
Reset
Integrate
No Integration
Input

- `sampled level`

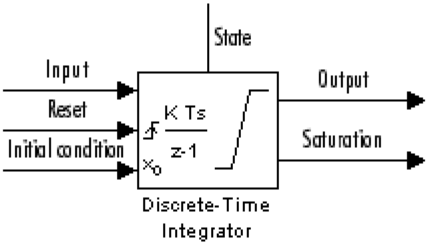
  Resets the output to the initial condition when the reset signal is nonzero. For example, the following figure shows the effect that a sampled level reset trigger has on backward Euler integration.


Reset
Sampled
Level Reset
No Integration
Integrate
Input

Note The `sampled level` reset option requires fewer computations and hence is more efficient than the `level` reset option. However, the `sampled level` reset option might introduce a discontinuity when integration resumes.

Block Icon When Selecting All Options

When you select all options, the icon looks like this:



Behavior When Using Simplified Initialization Mode

If you use simplified initialization mode, the behavior of the Discrete-Time Integrator block differs from its behavior in 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 results using explicit sample time, where the block is triggered at the same rate as the explicit sample time

In addition, the simplified initialization behavior makes it easier to convert 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”.

Initial Conditions in Simplified Initialization Mode

When you use simplified initialization mode, the **Initial condition** parameter applies only to the integrator output.

In addition, the **Use initial condition as initial and reset value for** parameter is disabled. The block uses the initial condition as the initial and reset value for the output.

Input-Output Equations in Simplified Initialization Mode

When you use simplified initialization mode, the block starts from first time step `n = 0` with initial output `y(0) = IC` (clipped if necessary).

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 input-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`.

Enable and Disable Behavior in Simplified Initialization Mode

When you use simplified initialization mode, the enable and disable behavior of the block is simplified as follows:

At disable time t_d:

$$y(t_d) = y(t_d-1)$$

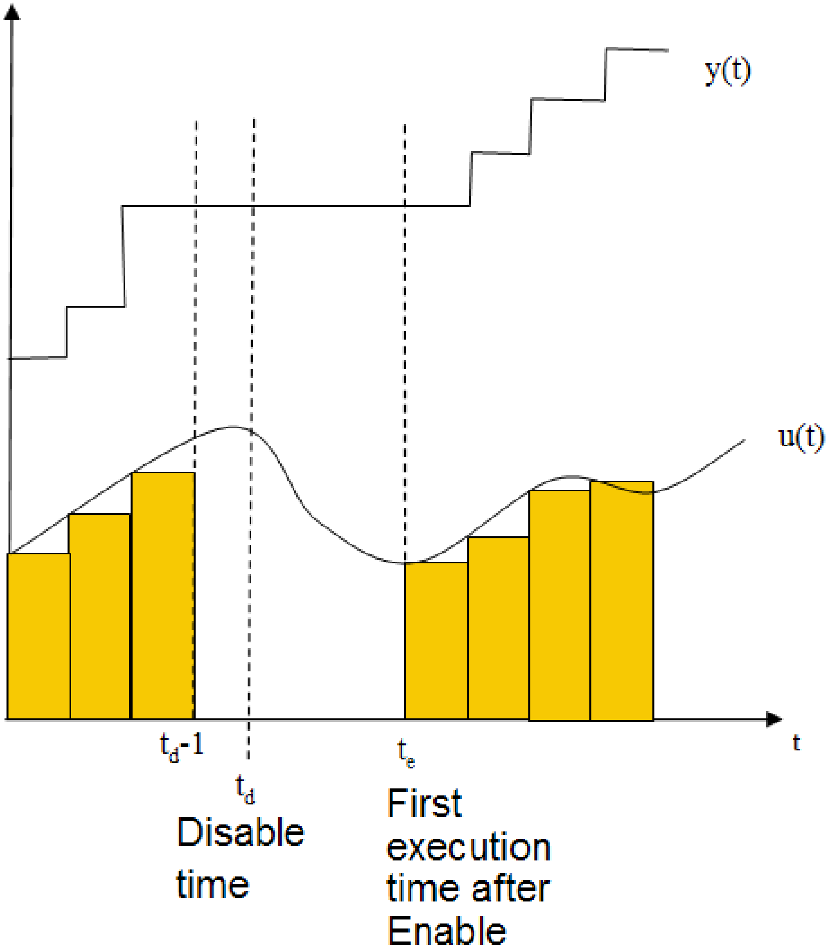
At enable time t_e:

- If parent subsystem resets on enable:

  $y(t_e) = IC$

- In all other cases (see below):

  $y(t_e) = y(t_d)$



Iterator Subsystems

When using simplified initialization mode, you cannot place the Discrete-Time Integrator block in an Iterator Subsystem.

In simplified initialization mode, Iterator subsystems do not maintain elapsed time, so Simulink reports an error if any block needing elapsed time, such as the Discrete-Time Integrator, is placed inside an Iterator Subsystem block.

Behavior When Used in an Enabled Subsystem Inside a Function-Call Subsystem

Suppose that you have a function-call subsystem that contains an enabled subsystem, which contains a Discrete-Time Integrator block. The following behavior applies.

Integrator Method	Sample Time Type of Function-Call Trigger Port	Value of `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`

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
			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`.

Data Type Support

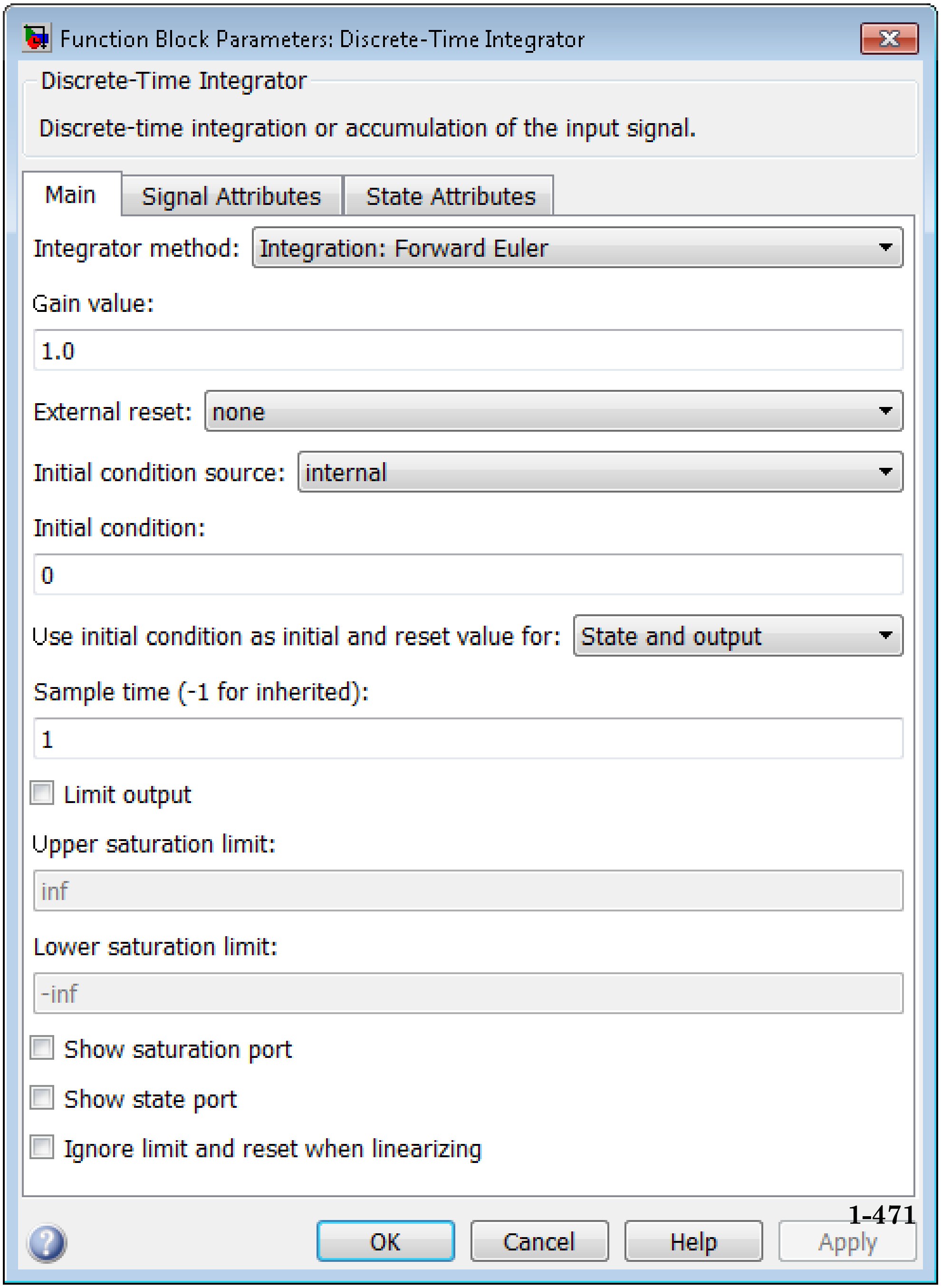
The Discrete-Time Integrator block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box

The **Main** pane of the Discrete-Time Integrator block dialog box appears as follows:


Function Block Parameters: Discrete-Time Integrator
Main
Signal Attributes
State Attributes
OK
Cancel
Help
Apply

Discrete-Time Integrator

Discrete-time integration or accumulation of the input signal.

Integrator method: Integration: Forward Euler

Gain value: 1.0

External reset: none

Initial condition source: internal

Initial condition: 0

Use initial condition as initial and reset value for: State and output

Sample time (-1 for inherited): 1

☐ Limit output

Upper saturation limit: inf

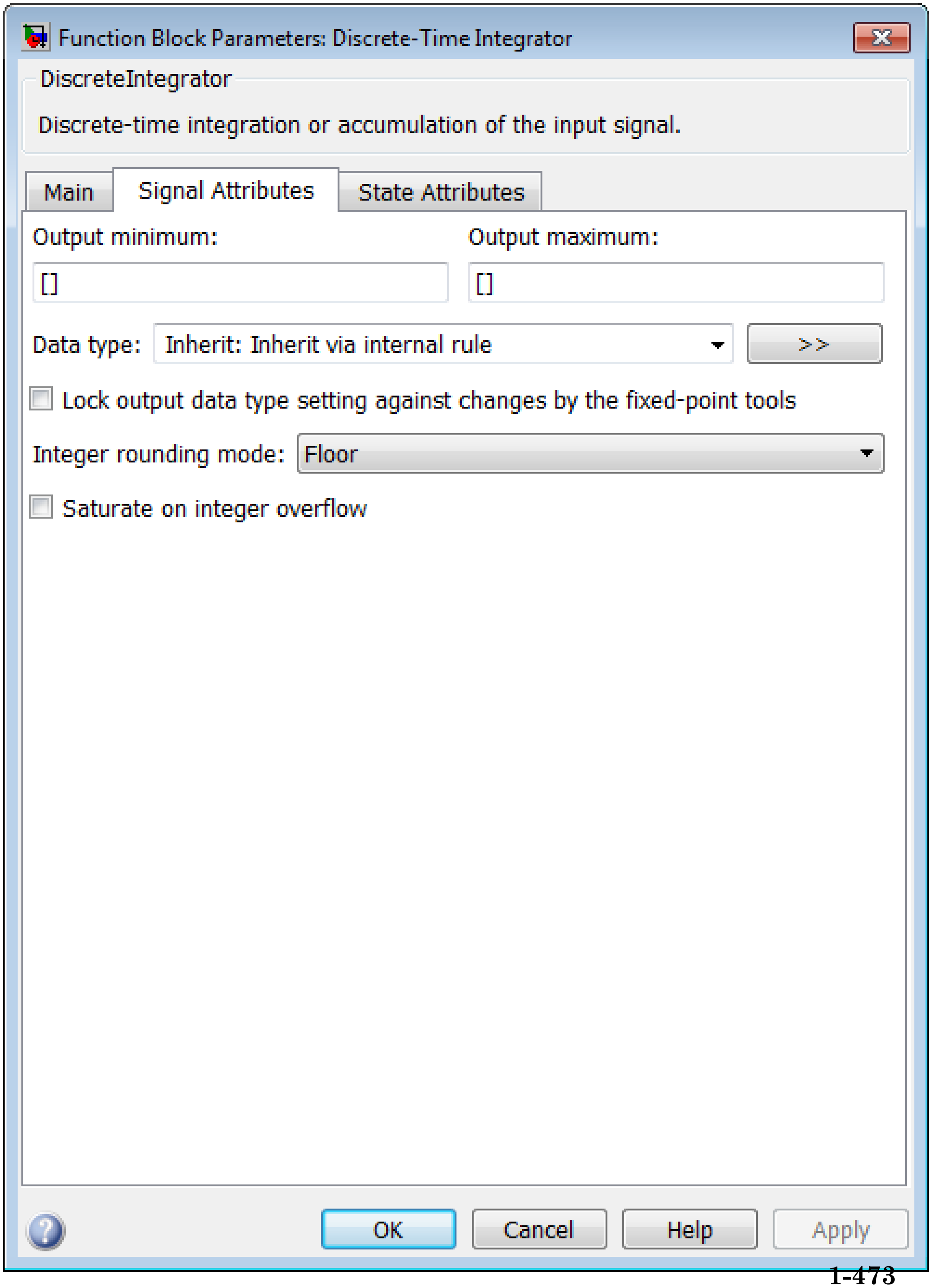
Lower saturation limit: -inf

☐ Show saturation port

☐ Show state port

☐ Ignore limit and reset when linearizing

The **Signal Attributes** pane of the Discrete-Time Integrator block dialog box appears as follows:



Function Block Parameters: Discrete-Time Integrator
DiscreteIntegrator
Main
Signal Attributes
State Attributes
>>
OK
Cancel
Help
Apply

Discrete-time integration or accumulation of the input signal.

Output minimum: []

Output maximum: []

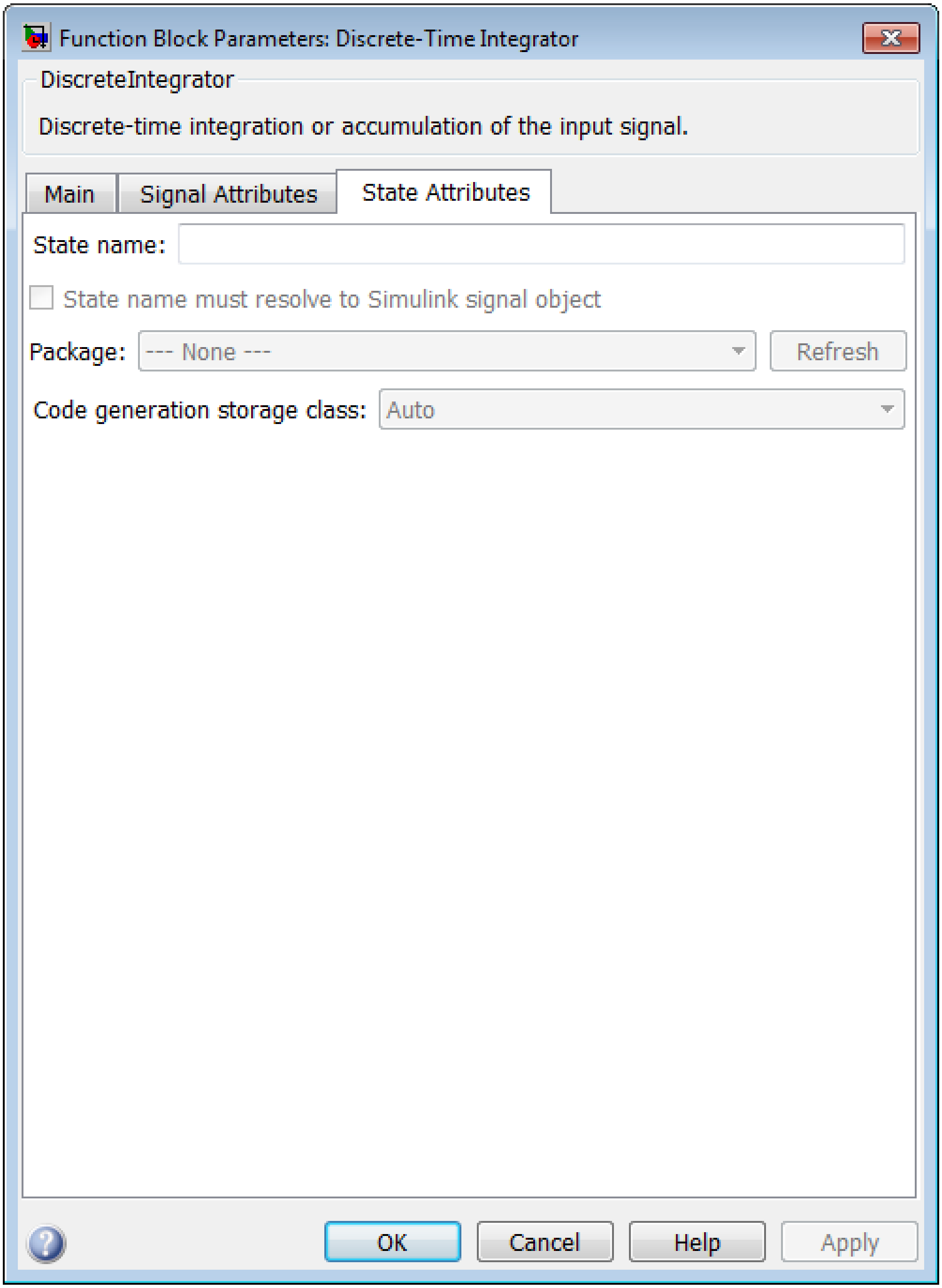
Data type: Inherit: Inherit via internal rule

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

Integer rounding mode: Floor

☐ Saturate on integer overflow

The **State Attributes** pane of the Discrete-Time Integrator block dialog box appears as follows:


Function Block Parameters: Discrete-Time Integrator

DiscreteIntegrator

Discrete-time integration or accumulation of the input signal.

Main
Signal Attributes
State Attributes

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Refresh

Code generation storage class: Auto

OK
Cancel
Help
Apply

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 “States” in the Simulink Coder documentation.

- “Show data type assistant” on page 1-1672
- “Integrator method” on page 1-478
- “Gain value” on page 1-479
- “External reset” on page 1-480
- “Initial condition source” on page 1-827
- “Initial condition” on page 1-482
- “Use initial condition as initial and reset value for” on page 1-483
- “Sample time (-1 for inherited)” on page 1-485
- “Limit output” on page 1-829
- “Upper saturation limit” on page 1-830
- “Lower saturation limit” on page 1-831
- “Show saturation port” on page 1-832
- “Show state port” on page 1-833
- “Ignore limit and reset when linearizing” on page 1-491
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1721
- “Integer rounding mode” on page 1-1722
- “Saturate on integer overflow” on page 1-1724
- “State name” on page 1-1975
- “State name must resolve to Simulink signal object” on page 1-1976

- “Package” on page 1-1977
- “Code generation storage class (when Package is selected)” on page 1-1979
- “Code generation storage class (when Package is selected)” on page 1-1979
- “Code generation storage type qualifier” on page 1-1981
- “Output minimum” on page 1-1727
- “Output maximum” on page 1-1728
- “Output data type” on page 1-506
- “Mode” on page 1-508
- “Data type override” on page 1-1926
- “Signedness” on page 1-1927
- “Word length” on page 1-1930
- “Scaling” on page 1-1736
- “Fraction length” on page 1-1931
- “Slope” on page 1-1932
- “Bias” on page 1-1932

Show data type assistant

Display the **Data Type Assistant**.

Settings

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

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Integrator method

Specify the integration or accumulation method.

Settings

Default: `Integration:  Forward Euler`

`Integration:  Forward Euler`
: Integrator method is Forward Euler.

`Integration:  Backward Euler`
: Integrator method is Backward Euler.

`Integration:  Trapezoidal`
: Integrator method is Trapezoidal.

`Accumulation:  Forward Euler`
: Accumulation method is Forward Euler.

`Accumulation:  Backward Euler`
: Accumulation method is Backward Euler.

`Accumulation:  Trapezoidal`
: Accumulation method is Trapezoidal.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Gain value

Specify a scalar, vector, or matrix by which to multiply the integrator input. Each element of the gain must be a positive real number.

Settings

Default: `1.0`

- Specifying a value other than 1.0 (the default) is semantically equivalent to connecting a Gain block to the input of the integrator.
- Valid entries include:
  - `double(1.0)`
  - `single(1.0)`
  - `[1.1 2.2 3.3 4.4]`
  - `[1.1 2.2; 3.3 4.4]`
- Using this parameter to specify the input gain eliminates a multiplication operation in the generated code. Realizing this benefit, however, requires that this parameter be nontunable. Accordingly, the Simulink Coder software generates a warning during code generation if the Model Parameter Configuration dialog box for this model declares this parameter to be tunable. 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

External reset

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

Settings

Default: `none`

`none`
: Do not reset the state to initial conditions.

`rising`
: Reset the state when the reset signal has a rising edge.

`falling`
: Reset the state when the reset signal has a falling edge.

`either`
: Reset the state when the reset signal rises or falls.

`level`
: Reset and holds the output to the initial condition while the reset signal is nonzero.

`sampled level`
: Reset the output to the initial condition when the reset signal is nonzero.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Initial condition source

Get the initial conditions of the states.

Settings

Default: `internal`

`internal`
: Get the initial conditions of the states from the **Initial condition** parameter.

`external`
: Get the initial conditions of the states from an external block.

Tips

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Initial condition

Specify the states' initial conditions.

Settings

Default: `0`

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Use initial condition as initial and reset value for

Specify whether to apply the initial condition as the initial and reset value for the state and output, or the state only.

Note If you are using simplified initialization mode, this parameter is disabled. The initial condition is always used as the initial and reset value for the output. For more information, see “Underspecified initialization detection”.

Settings

Default: `State and output`

`State and output`

Set the following for initial

`y(0) = IC`

`x(0) = IC`

or at reset

`y(n) = IC`

`x(n) = IC`

`State only (most efficient)`

Set the following for initial

`x(0) = IC`

or at reset

`x(n) = IC`

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits.

Settings

Default: 1

By default, the block uses a discrete sample time of 1. To set a different sample time, enter another discrete value, such as 0.1.

See also "Specify Sample Time" in the online documentation for more information.

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Limit output

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

Settings

Default: Off

☑ On

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

☐ Off

Do not limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

Dependencies

This parameter enables **Upper saturation limit**.

This parameter enables **Lower saturation limit**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Upper saturation limit

Specify the upper limit for the integral.

Settings

Default: `inf`

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Dependencies

Limit output enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Lower saturation limit

Specify the lower limit for the integral.

Settings

Default: `-inf`

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Dependencies

Limit output enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Show saturation port

Add a saturation output port to the block.

Settings

Default: Off

☑ On
Add a saturation output port to the block.

☐ Off
Do not add a saturation output port to the block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Show state port

Add an output port to the block for the block's state.

Settings

Default: Off

☑ On

Add an output port to the block for the block's state.

☐ Off

Do not add an output port to the block for the block's state.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Ignore limit and reset when linearizing

Cause Simulink linearization commands to treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

Settings

Default: Off

☑ On

Cause Simulink linearization commands to treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

☐ Off

Do not cause Simulink linearization commands to treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

Tips

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

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

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

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
: Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`
: Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`
: Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`
: Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`
: Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`
: Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`
: Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Rounding" in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see “Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

State name

Use this parameter to assign a unique name to each state.

Settings

Default: `' '`

- If left blank, no name is assigned.

Tips

- 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.

Dependency

This parameter enables **State name must resolve to Simulink signal object** when you click the **Apply** button.

For more information, see “States” in the Simulink Coder documentation.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Settings

Default: Off

☑ On

Require that state name resolve to Simulink signal object.

☐ Off

Do not require that state name resolve to Simulink signal object.

Dependencies

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Package

Select a package that defines the custom storage class you want to apply.

Settings

Default: `---None---`

`---None---`
  Sets internal storage class attributes.

`mpt`
  Applies the built-in `mpt` package.

`Simulink`
  Applies the built-in `Simulink` package.

Dependencies

If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Code generation storage class

Select state storage class.

Settings

Default: `Auto`

`Auto`
Auto is the appropriate storage class for states that you do not need to interface to external code.

`ExportedGlobal`
State is stored in a global variable

`ImportedExtern`
`*model*_private.h` declares the state as an extern variable.

`ImportedExternPointer`
`*model*_private.h` declares the state as an extern pointer.

Dependencies

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“State Storage Classes” in the Simulink Coder documentation.

Code generation storage class (when Package is selected)

Select custom storage class for state.

Settings

Default: `Auto`

`Auto`
: Auto is the appropriate storage class for states that you do not need to interface to external code.

`SimulinkGlobal`
: *`model_P`* initializes the state to its corresponding value in the workspace.

`ExportedGlobal`
: State is stored in a global variable

`ImportedExtern`
: *`model`*`_private.h` declares the state as an extern variable.

`ImportedExternPointer`
: *`model`*`_private.h` declares the state as an extern pointer.

`Default`
: A non-editable placeholder storage class is created.

`BitField`
: A `struct` declaration is created that embeds Boolean data.

`Volatile`
: Volatile type qualifier is used in state declaration.

`ExportToFile`
: Header `(.h)` file containing global variable declarations is generated with user-specified name.

`ImportFromFile`
: Predefined header `(.h)` files containing global variable declarations are included.

`FileScope`
A static qualifier is generated in front of the state declaration to make the state visible only to the current file.

`Struct`
A `struct` declaration is created to encapsulate parameter or signal object data.

`StructVolatile`
Volatile type qualifier is used in `struct` declaration.

`GetSet`
Supports specialized function calls to read and write memory.

Dependencies

State name enables this parameter.

The list of valid storage classes differs based on the **Package** selection.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“State Storage Classes” in the Simulink Coder documentation.

Code generation storage type qualifier

Specify the Simulink Coder storage type qualifier.

Settings

Default: `'  '`

If left blank, no qualifier is assigned.

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the minimum to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`double`

Output data type is `double`.

`single`

Output data type is `single`.

`int8`

Output data type is `int8`.

`uint8`
: Output data type is `uint8`.

`int16`
: Output data type is `int16`.

`uint16`
: Output data type is `uint16`.

`int32`
: Output data type is `int32`.

`uint32`
: Output data type is `uint32`.

`fixdt(1,16,0)`
: Output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
: Output data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
: Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specify Block Output Data Types".

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule` (default)
- `Inherit via back propagation`

`Built in`

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

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

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

Dependency

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

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

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

Tip

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

Dependency

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

Signedness

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

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Word length

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

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Scaling

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

Settings

Default: `Best precision`

`Binary point`
Specify binary point location.

`Slope and bias`
Enter slope and bias.

`Best precision`
Specify best-precision values.

Dependencies

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

Selecting `Binary point` enables:

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

Selecting `Slope and bias` enables:

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

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

Command-Line Information

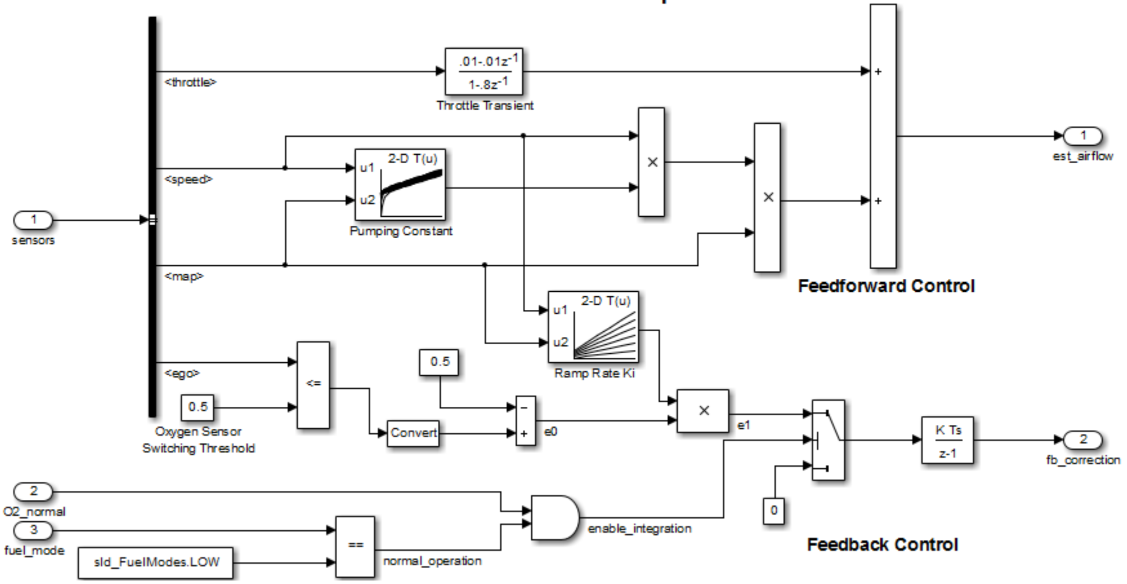
See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Examples

The `sldemo_fuelsys` model uses a Discrete-Time Integrator block in the `fuel_rate_control/airflow_calc` subsystem. This block uses the Forward Euler integration method.


Intake Airflow Estimation and Closed-Loop Correction
<throttle>
.01-.01z-1
1-.8z-1
Throttle Transient
<speed>
2-D T(u)
u1
u2
Pumping Constant
1
sensors
<map>
×
×
+
+
1
est_airflow
Feedforward Control
2-D T(u)
u1
u2
Ramp Rate Ki
<ego>
<=
0.5
0.5
Oxygen Sensor Switching Threshold
Convert
-
+
e0
×
e1
K Ts
z-1
2
fb_correction
0
2
O2_normal
3
fuel_mode
sld_FuelModes.LOW
==
normal_operation
enable_integration
Feedback Control

When the Switch block feeds a nonzero value into the Discrete-Time Integrator block, integration occurs. Otherwise, integration does not occur.

Characteristics

Direct Feedthrough	Yes, of the reset and external initial condition source ports. The input has direct feedthrough for every integration method except Forward Euler and accumulation Forward Euler.
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of parameters

States	Inherited from driving block and parameter
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Integrator

Discrete Transfer Fcn

Purpose

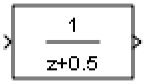
Implement discrete transfer function

Library

Discrete

Description

The Discrete Transfer Fcn block implements the *z*-transform transfer function:



$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^m + num_1 z^{n-1} + \ldots + num_m}{den_0 z^n + den_1 z^{n-1} + \ldots + den_n}$$

where *m+1* and *n+1* are the number of numerator and denominator coefficients, respectively. *num* and *den* contain the coefficients of the numerator and denominator in descending powers of *z*. *num* can be a vector or matrix, *den* must be a vector, and you specify both as parameters on the block dialog box. The order of the denominator must be greater than or equal to the order of the numerator.

Specify the coefficients of the numerator and denominator polynomials in descending powers of *z*. This block lets you use polynomials in *z* to represent a discrete system, a method that control engineers typically use. Conversely, the Discrete Filter block lets you use polynomials in z^{-1} (the delay operator) to represent a discrete system, a method that signal processing engineers typically use. The two methods are identical when the numerator and denominator polynomials have the same length.

The Discrete Transfer Fcn block applies the *z*-transform transfer function to each independent channel of the input. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an individual channel (sample-based processing), or each column of the input as an individual channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

Specifying Initial States

Use the **Initial states** parameter to specify initial filter states. To determine the number of initial states you must specify and how to specify them, see the following tables.

Frame-Based Processing

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
• 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)	• Scalar • Column vector (*M*-by-1)

Sample-Based Processing (Continued)

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
		• Row vector (1-by-M)	• 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)	$K \times N$	• Scalar • Column vector (M-by-1) • Row vector (1-by-M) • Matrix (M-by-($K \times N$))	• Scalar

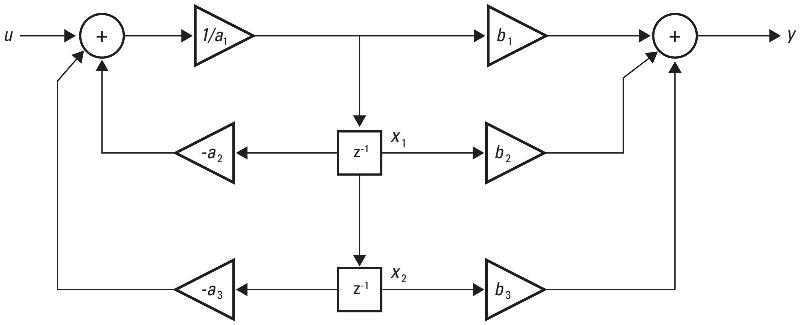
When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. Enter `0` to initialize all states to zero. When the **Initial states** is a vector or a matrix, each vector or matrix element specifies a unique initial state for a corresponding delay element 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:

$$y_1 = b_1\left[\frac{(u_1 - a_2x_1 - a_3x_2)}{a_1}\right] + b_2x_1 + b_3x_2$$



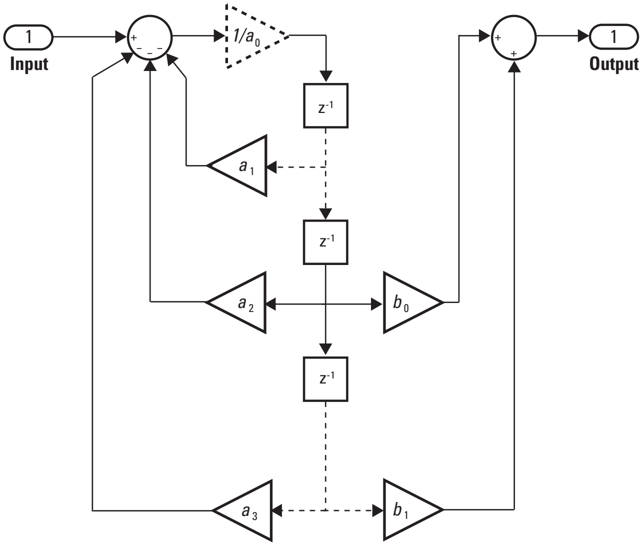
Data Type Support

The Discrete Transfer Function block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

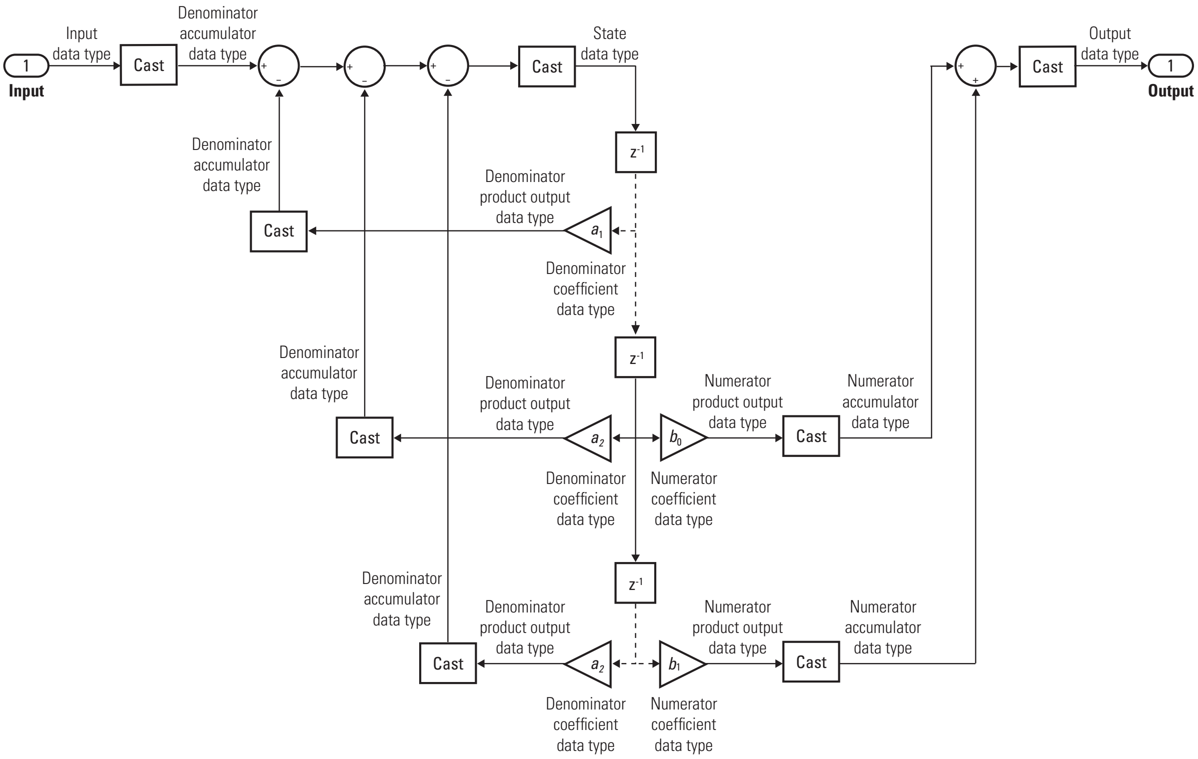
Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

States are complex when either the input or the coefficients are complex.

The following diagrams show the filter structure and the data types that the block uses for floating-point and fixed-point signals.

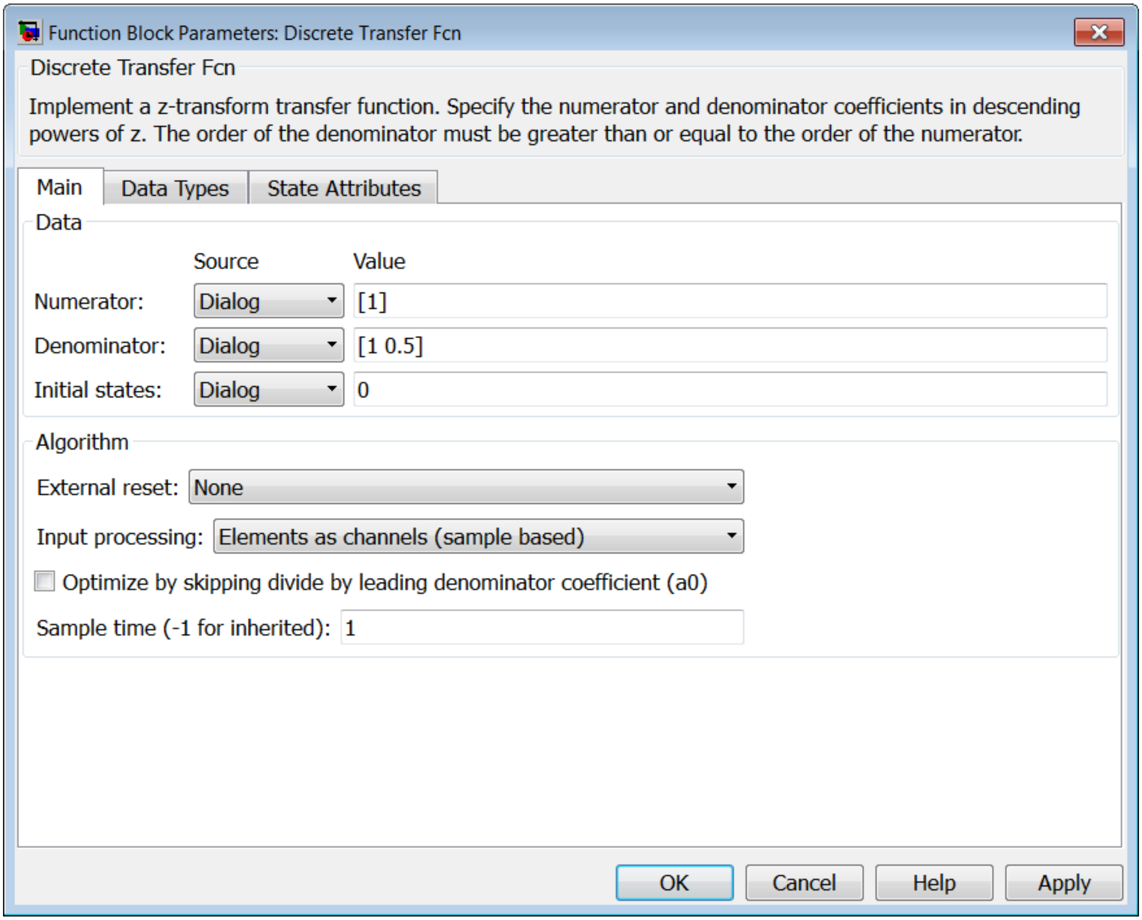

1
Input
+
_
_
1/a0
z-1
a1
z-1
a2
b0
z-1
a3
b1
+
+
1
Output

The block omits the dashed divide when you select the **Optimize by skipping divide by leading denominator coefficient (a0)** parameter.


1
Input
Input data type
Cast
Denominator accumulator data type
Cast
State data type
z-1
Denominator accumulator data type
Cast
Denominator product output data type
a1
Denominator coefficient data type
z-1
Denominator accumulator data type
Cast
Denominator product output data type
a2
b0
Numerator product output data type
Cast
Numerator accumulator data type
Denominator coefficient data type
Numerator coefficient data type
z-1
Denominator accumulator data type
Cast
Denominator product output data type
a2
b1
Numerator product output data type
Cast
Numerator accumulator data type
Denominator coefficient data type
Numerator coefficient data type
Cast
Output data type
1
Output

Parameters and Dialog Box

The **Main** pane of the Discrete Transfer Fcn block dialog box appears as follows.


Function Block Parameters: Discrete Transfer Fcn
Main
Data Types
State Attributes
OK
Cancel
Help
Apply

Discrete Transfer Fcn

Implement a z-transform transfer function. Specify the numerator and denominator coefficients in descending powers of z. The order of the denominator must be greater than or equal to the order of the numerator.

Data

	Source	Value
Numerator:	Dialog	[1]
Denominator:	Dialog	[1 0.5]
Initial states:	Dialog	0

Algorithm

External reset: None

Input processing: Elements as channels (sample based)

☐ Optimize by skipping divide by leading denominator coefficient (a0)

Sample time (-1 for inherited): 1

Numerator

Numerator coefficients of the discrete transfer function. To specify the coefficients, set the **Source** to `Dialog`. Then enter the coefficients in **Value** as descending powers of z. Use a row vector to specify the coefficients for a single numerator polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps.

Denominator

Denominator coefficients of the discrete transfer function. To specify the coefficients, set the **Source** to `Dialog`. Then, enter the coefficients in **Value** as descending powers of z. Use a row vector to specify the coefficients for a single denominator polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps.

Initial states

If the **Source** is `Dialog`, then, in **Value**, specify the initial states of the filter states. To learn how to specify initial states, see "Specifying Initial States" on page 1-518.

If the **Source** is `Input port`, then there is nothing to be specified for **Value**.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
`None`	No reset.
`Rising`	Reset on a rising edge.
`Falling`	Reset on a falling edge.
`Either`	Reset on either a rising or falling edge.
`Level`	Reset in either of these cases: • when there is a nonzero at the current time step • when the time step value changes from nonzero at the previous time step to zero at the current time step
`Level hold`	Reset when nonzero at the current time step.

The reset signal must be scalar.

Input processing

Specify whether the block performs sample- or frame-based processing.

- `Elements as channels (sample based)` — Process each element of the input as an independent channel.
- `Columns as channels (frame based)` — Process each column of the input as an independent channel.

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

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

Optimize by skipping divide by leading denominator coefficient (a0)

Select when the leading denominator coefficient, a_0, equals one. This parameter optimizes your code.

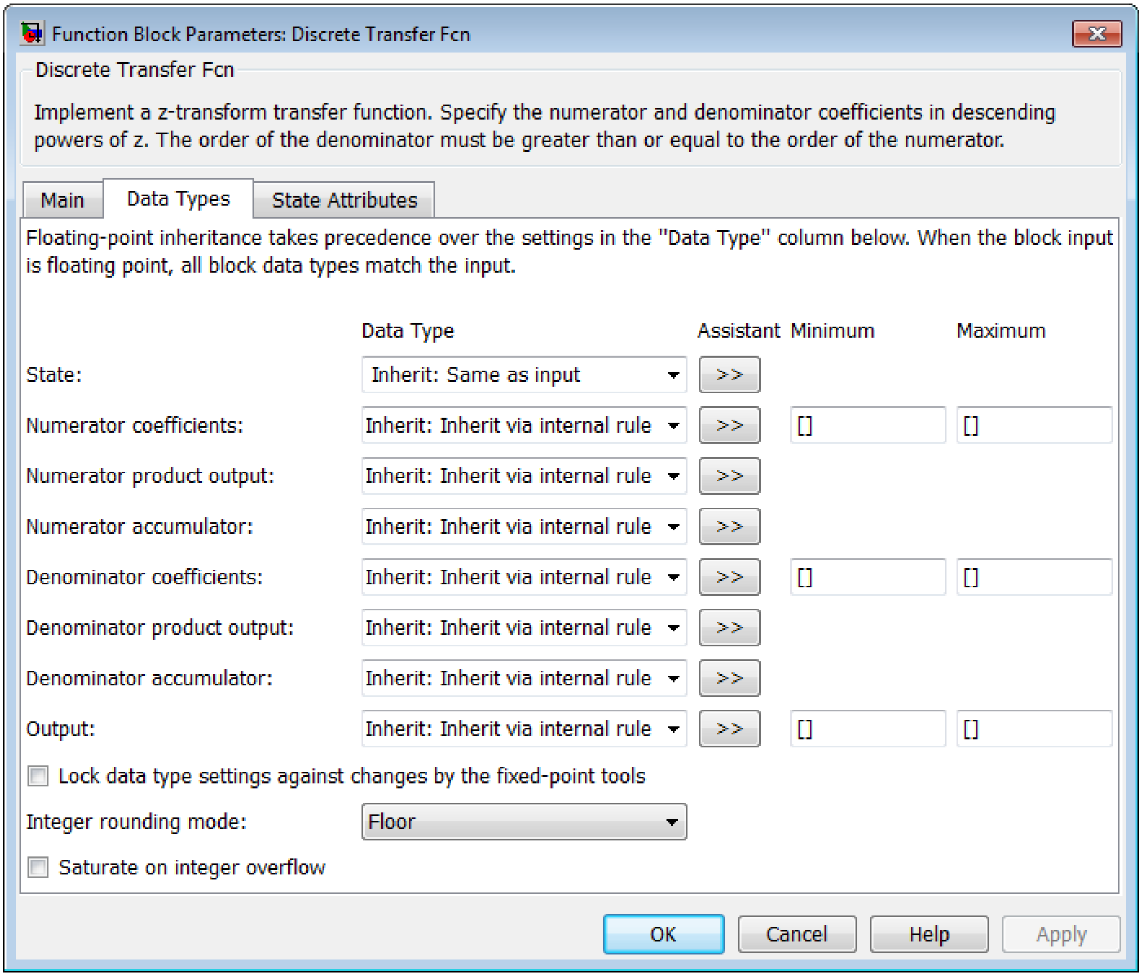
When you select this check box, the block does not perform a divide-by-a_0 either in simulation or in the generated code. An error occurs if a_0 is not equal to one.

When you clear this check box, the block is fully tunable during simulation, and performs a divide-by-a_0 in both simulation and code generation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in “How Simulink Works” in the *Simulink User’s Guide*.

The **Data Types** pane of the Discrete Transfer Function block dialog box appears as follows.



Discrete Transfer Fcn

Implement a z-transform transfer function. Specify the numerator and denominator coefficients in descending powers of z. The order of the denominator must be greater than or equal to the order of the numerator.

Floating-point inheritance takes precedence over the settings in the "Data Type" column below. When the block input is floating point, all block data types match the input.

	Data Type	Assistant	Minimum	Maximum
State:	Inherit: Same as input	>>		
Numerator coefficients:	Inherit: Inherit via internal rule	>>	[]	[]
Numerator product output:	Inherit: Inherit via internal rule	>>		
Numerator accumulator:	Inherit: Inherit via internal rule	>>		
Denominator coefficients:	Inherit: Inherit via internal rule	>>	[]	[]
Denominator product output:	Inherit: Inherit via internal rule	>>		
Denominator accumulator:	Inherit: Inherit via internal rule	>>		
Output:	Inherit: Inherit via internal rule	>>	[]	[]

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

Integer rounding mode: Floor

☐ Saturate on integer overflow

State

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

- A rule that inherits a data type, for example, `Inherit: Same as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Numerator coefficients

Specify the numerator coefficient data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Numerator coefficient minimum

Specify the minimum value that a numerator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Numerator coefficient maximum

Specify the maximum value that a numerator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Numerator product output

Specify the product output data type for the numerator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Numerator accumulator

Specify the accumulator data type for the numerator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Denominator coefficients

Specify the denominator coefficient data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Denominator coefficient minimum

Specify the minimum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Denominator coefficient maximum

Specify the maximum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Automatic scaling of fixed-point data types

Denominator product output

Specify the product output data type for the denominator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Data Types Using Data Type Assistant” in theSimulink User’s Guide for more information.

Denominator accumulator

Specify the accumulator data type for the denominator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See "Specify Data Types Using Data Type Assistant" in the *Simulink User's Guide* for more information.

Output

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

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See “Specify Block Output Data Types” in the *Simulink User’s Guide* for more information.

Output minimum

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

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

Output maximum

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

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

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

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Locking the Output Data Type Setting”in the Fixed-Point Designer documentation.

Integer rounding mode

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

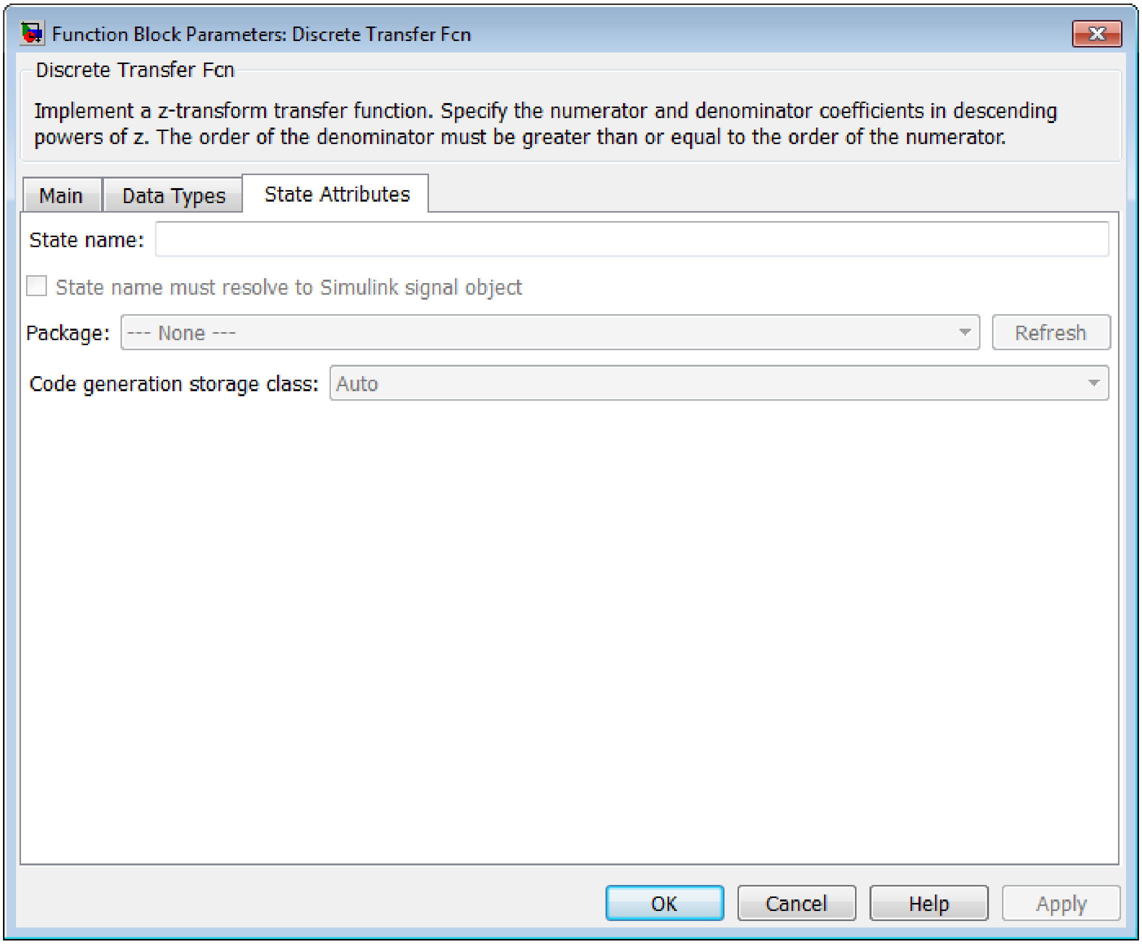
Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit	Overflows saturate to either the minimum or maximum value	An overflow associated with a signed 8-bit

Action	Reasons for Taking This Action	What Happens for Overflows	Example
	saturation protection in the generated code.	that the data type can represent.	integer can saturate to –128 or 127.
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 "Checking 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.

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.

The **State Attributes** pane of the Discrete Filter block dialog box appears as follows.



Function Block Parameters: Discrete Transfer Fcn

Discrete Transfer Fcn

Implement a z-transform transfer function. Specify the numerator and denominator coefficients in descending powers of z. The order of the denominator must be greater than or equal to the order of the numerator.

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Code generation storage class: Auto

State name

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “States” in the Simulink Coder documentation.

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Package

Select a package that defines the custom storage class you want to apply. If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Code generation storage class

From the list, select a state storage class.

`Auto`

Auto is the storage class to use for states that do not need to interface to external code.

`ExportedGlobal`

The state is stored in a global variable.

`ImportedExtern`

`*model*_private.h` declares the state as an extern variable.

`ImportedExternPointer`

`*model*_private.h` declares the state as an extern pointer.

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Code generation storage type qualifier

Specify a Simulink Coder storage type qualifier. The default is `' '`. When this field is blank, no qualifier is assigned.

The Simulink Coder product does not check this string for errors. Thus, whatever value you enter appears automatically in the variable declaration.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name resolves
- Minimum and maximum values of the signal object

For more information, see “States” in the Simulink Coder documentation.

Characteristics

Direct Feedthrough	Only when the leading numerator coefficient is not equal to zero and the numerator order equals the denominator order
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of initial states
States	See “Specifying Initial States” on page 1-518

Dimensionalized	Yes
Zero-Crossing Detection	No

Purpose

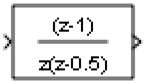
Model system defined by zeros and poles of discrete transfer function

Library

Discrete

Description

The Discrete Zero-Pole block models a discrete system defined by the zeros, poles, and gain of a *z*-domain transfer function. This block assumes that the transfer function has the following form:



$$H(z) = K\frac{Z(z)}{P(z)} = K\frac{(z-Z_1)(z-Z_2)\ldots(z-Z_m)}{(z-P_1)(z-P_2)\ldots(z-P_n)},$$

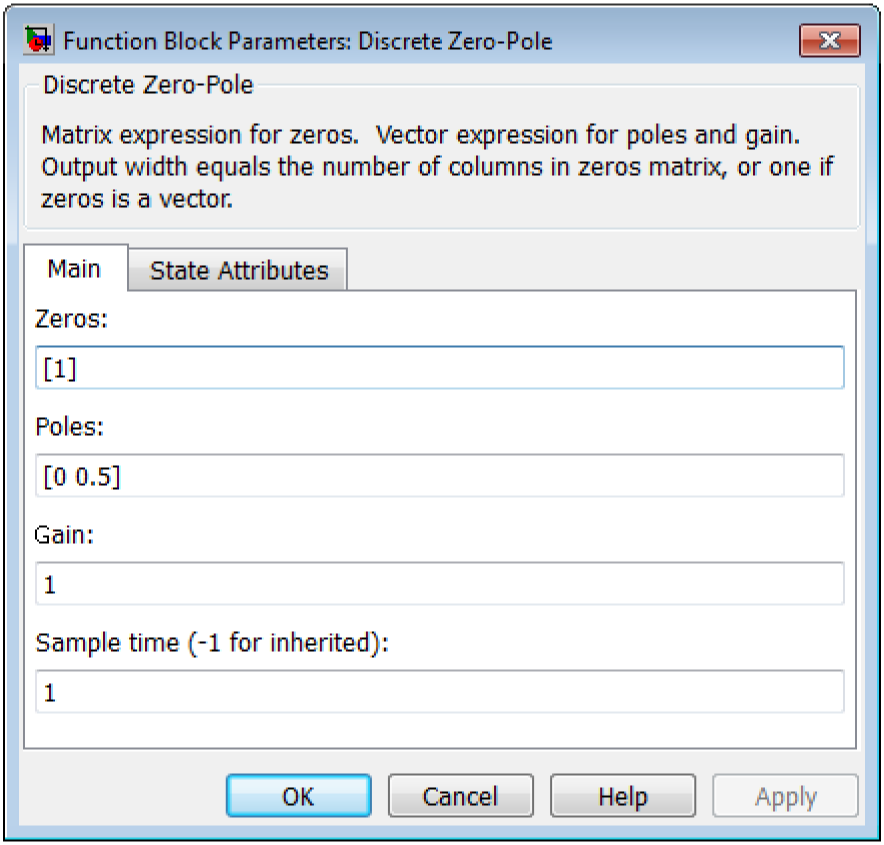
where Z represents the zeros vector, P the poles vector, and K the gain. The number of poles must be greater than or equal to the number of zeros ($n \geq m$). If the poles and zeros are complex, they must be complex conjugate pairs.

The block displays the transfer function depending on how the parameters are specified. See Zero-Pole for more information.

Data Type Support

The Discrete Zero-Pole block accepts and outputs real signals of type `double` and `single`. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box


Function Block Parameters: Discrete Zero-Pole
Main
State Attributes
OK
Cancel
Help
Apply

Discrete Zero-Pole

Matrix expression for zeros. Vector expression for poles and gain. Output width equals the number of columns in zeros matrix, or one if zeros is a vector.

Zeros: [1]

Poles: [0 0.5]

Gain: 1

Sample time (-1 for inherited): 1

Zeros
Specify the matrix of zeros. The default is `[1]`.

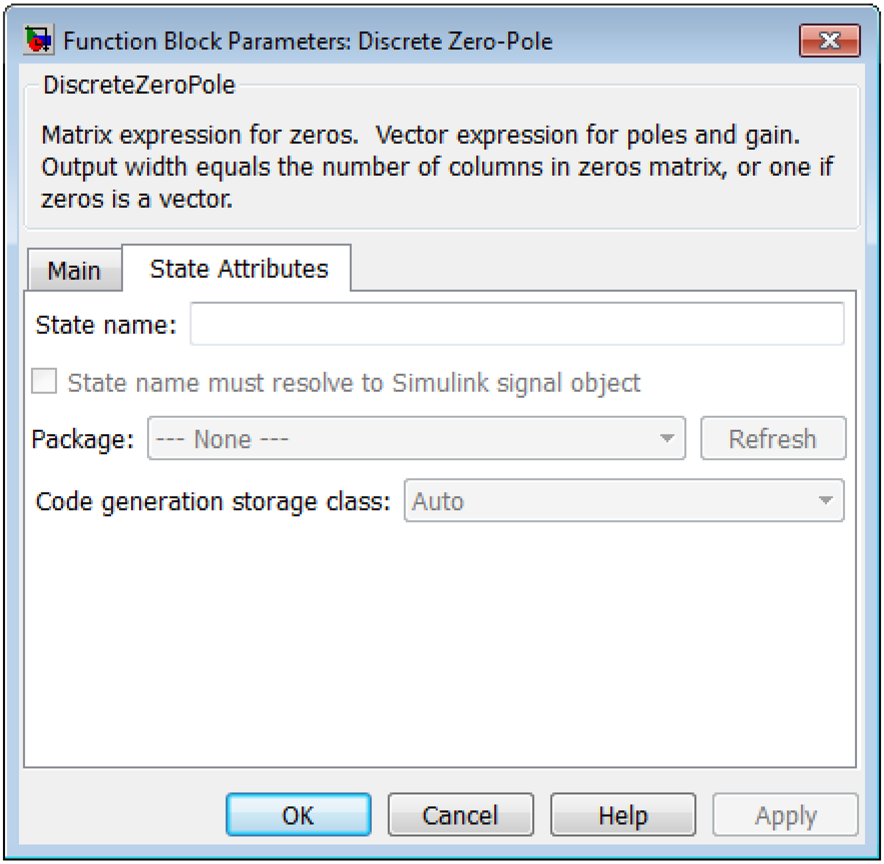
Poles
Specify the vector of poles. The default is `[0 0.5]`.

Gain
Specify the gain. The default is `1`.

Sample time

Specify the time interval between samples. See Specifying Sample Time in the "How Simulink Works" chapter of the Simulink documentation.

The **State Attributes** pane of the Discrete Zero-Pole block dialog box appears as follows:



Function Block Parameters: Discrete Zero-Pole

DiscreteZeroPole

Matrix expression for zeros. Vector expression for poles and gain. Output width equals the number of columns in zeros matrix, or one if zeros is a vector.

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Code generation storage class: Auto

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “States” in the Simulink Coder documentation.

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Package

Select a package that defines the custom storage class you want to apply. If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Code generation storage class

From the list, select a state storage class.

`Auto`

Auto is the storage class to use for states that do not need to interface to external code.

`ExportedGlobal`

The state is stored in a global variable.

`ImportedExtern`
`model``_private.h` declares the state as an extern variable.

`ImportedExternPointer`
`model``_private.h` declares the state as an extern pointer.

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Code generation storage type qualifier
Specify a Simulink Coder storage type qualifier. The default is `' '`. When this field is blank, no qualifier is assigned.

The Simulink Coder product does not check this string for errors. Thus, whatever value you enter appears automatically in the variable declaration.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

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 "States" in the Simulink Coder documentation.

Characteristics

Direct Feedthrough	Yes, if the number of zeros and poles are equal
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
States	Length of **Poles** vector
Dimensionalized	No
Zero Crossing	No

Purpose

Show value of input

Library

Sinks

Description

Format Options

You control the display format using the **Format** parameter:

If you select...	The block displays...
`short`	A 5-digit scaled value with fixed decimal point
`long`	A 15-digit scaled value with fixed decimal point
`short_e`	A 5-digit value with a floating decimal point
`long_e`	A 16-digit value with a floating decimal point
`bank`	A value in fixed dollars and cents format (but with no $ or commas)
`hex (Stored Integer)`	The stored integer value of a fixed-point input in hexadecimal format
`binary (Stored Integer)`	The stored integer value of a fixed-point input in binary format

If you select...	The block displays...
`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 “About Simulink Enumerations” and “Define Simulink Enumerations”):

- The block displays enumerated values, not the values of underlying integers.
- Setting **Format** to any of the `Stored Integer` settings causes an error.

Display Abbreviations

The following abbreviations appear on the Display block to help you identify the format of the value.

When you see...	The value that appears is...
`(SI)`	The stored integer value **Note** `(SI)` does not appear when the signal is of an integer data type.
`hex`	In hexadecimal format
`bin`	In binary format
`oct`	In octal format

Frequency of Data Display

The amount of data that appears and the time steps at which the data appears depend on the **Decimation** block parameter and the `SampleTime` property:

- The **Decimation** parameter enables you to display data at every *n*th sample, where *n* is the decimation factor. The default decimation, `1`, displays data at every time step.

  Note The Display block updates its display at the initial time, even when the **Decimation** value is greater than one.

- The `SampleTime` property, which you can set with `set_param`, enables you to specify a sampling interval at which to display points. This property is useful when you are using a variable-step solver where the interval between time steps is not the same. The default sample time, `-1`, causes the block to ignore the sampling interval when determining the points to display.

Resizing Options

If the block input is an array, you can resize the block to show more than just the first element. You can resize the block vertically or horizontally, and the block adds display fields in the appropriate direction. A black triangle indicates that the block is not displaying all input array elements.

The Display block shows the first 200 elements of a vector signal and the first 20 rows and 10 columns of a matrix signal.

Floating Display

To use the block as a floating display, select the **Floating display** check box. The block input port disappears and the block displays the value of the signal on a selected line.

If you select **Floating display**:

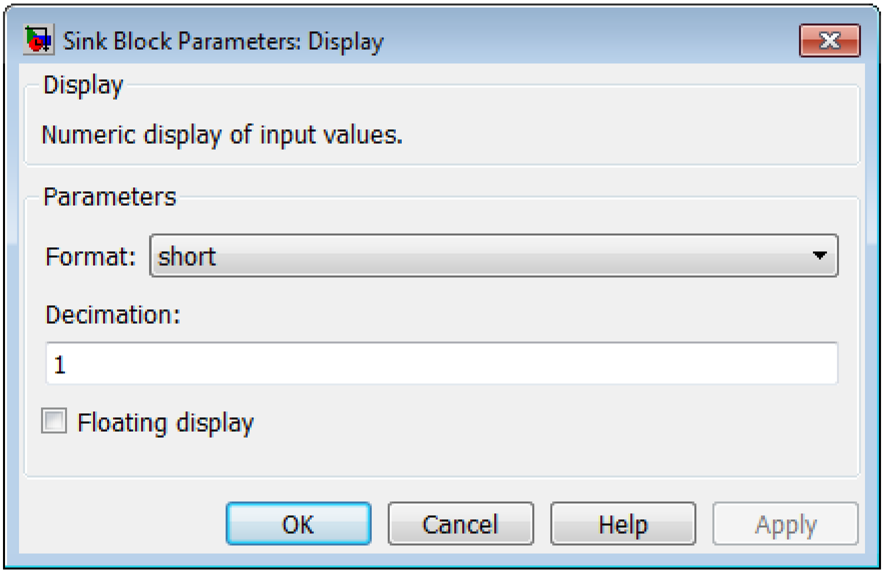
- Turn off signal storage reuse for your model. See “Signal storage reuse” in the Simulink documentation for more information.
- Do not connect a multidimensional signal to a floating display. Otherwise, you get a simulation error because the block does not support multidimensional signals.

Data Type Support

The Display block accepts real or complex signals of the following data types:

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

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



Sink Block Parameters: Display

Display

Numeric display of input values.

Parameters

Format: short

Decimation:

1

☐ Floating display

Parameters and Dialog Box

Format

Specify the format of the data that appears, as discussed in "Format Options" on page 1-545. The default is `short`.

Decimation

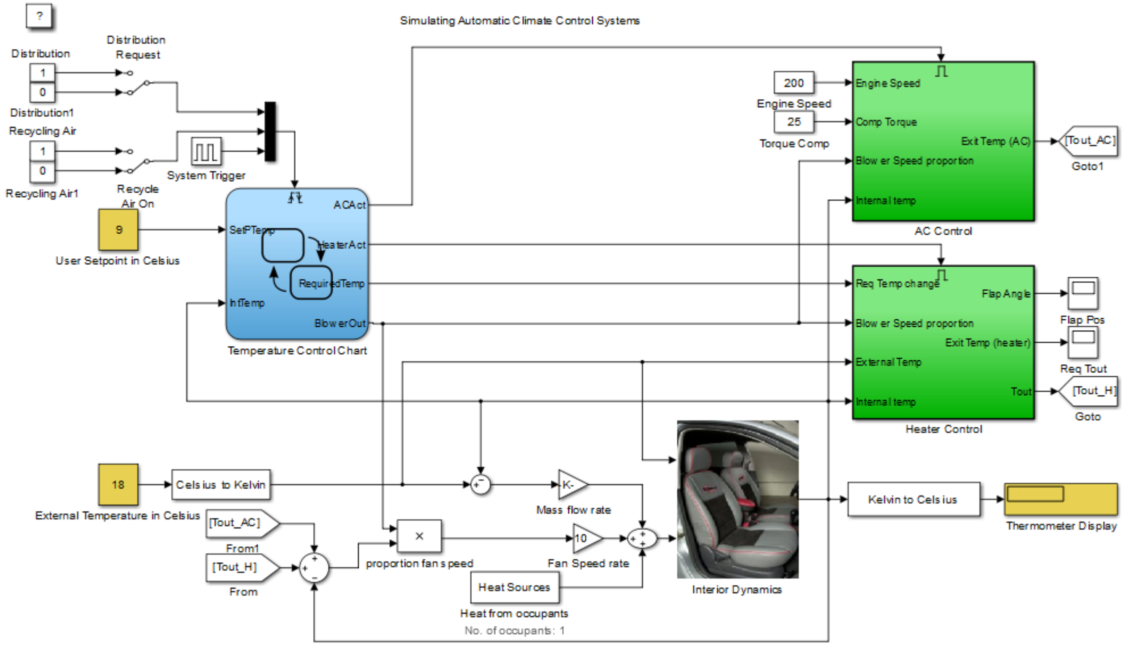
Specify how often to display data, as discussed in "Frequency of Data Display" on page 1-547. The default is `1`.

Floating display

Select to use the block as a floating display, as discussed in "Floating Display" on page 1-547.

Examples

The `sldemo_auto_climatecontrol` model shows how you can use the Display block.


Simulating Automatic Climate Control Systems
?
Distribution
Distribution Request
1
0
Distribution1
Recycling Air
1
0
Recycling Air1
Recycle Air On
System Trigger
9
User Setpoint in Celsius
ACAct
SetPTemp
HeaterAct
RequiredTemp
IntTemp
BlowerOut
Temperature Control Chart
200
Engine Speed
25
Torque Comp
Engine Speed
Comp Torque
Exit Temp (AC)
Blower Speed proportion
Internal temp
AC Control
[Tout_AC]
Goto1
Req Temp change
Flap Angle
Blower Speed proportion
Exit Temp (heater)
External Temp
Tout
Internal temp
Heater Control
Flap Pos
Req Tout
[Tout_H]
Goto
18
External Temperature in Celsius
Celsius to Kelvin
-K-
Mass flow rate
[Tout_AC]
From1
[Tout_H]
From
proportion fan speed
10
Fan Speed rate
Heat Sources
Heat from occupants
No. of occupants: 1
Interior Dynamics
Kelvin to Celsius
Thermometer Display

Characteristics

Sample Time	Use `set_param` to specify the `SampleTime` property
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

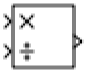
Scope

Purpose Divide one input by another

Library Math Operations

Description

Supported Block Operations



The Divide block outputs the result of dividing its first input by its second. The inputs can be scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The Divide block is functionally a Product block that has two block parameter values preset:

- **Multiplication:** `Element-wise(.*)`
- **Number of Inputs:** `*/`

Setting non-default values for either of those parameters can change a Divide block to be functionally equivalent to a Product block or a Product of Elements block. See the documentation of those two blocks for more information.

Expected Differences Between Simulation and Code Generation

If any of the Divide block inputs contains a `NaN` or `inf` value, or if the block generates `NaN` or `inf` during execution, you might see different results when you compare the block simulation results with the generated code. This difference is due to the nonfinite `NaN` or `inf` values. In such cases, inspect your model configuration and eliminate the conditions that produce `NaN` or `inf`.

Code Optimizations

The Simulink Coder build process provides efficient code for matrix inverse and division operations. The following summary describes the benefits and when each benefit is available:

Benefit	**Small matrices (2-by-2 to 5-by-5)**	**Medium matrices (6-by-6 to 20-by-20)**	**Large matrices (larger than 20-by-20)**
Faster code execution time, compared to R2011a and earlier releases	Yes	No	Yes
Reduced ROM and RAM usage, compared to R2011a and earlier releases	Yes, for real values	Yes, for real values	Yes, for real values
Reuse of variables	Yes	Yes	Yes
Dead code elimination	Yes	Yes	Yes
Constant folding	Yes	Yes	Yes
Expression folding	Yes	Yes	Yes
Consistency with MATLAB Coder results	Yes	Yes	Yes

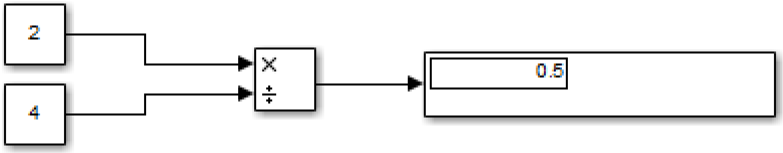
For blocks that have three or more inputs of different dimensions, the code might include an extra buffer to store temporary variables for intermediate results.

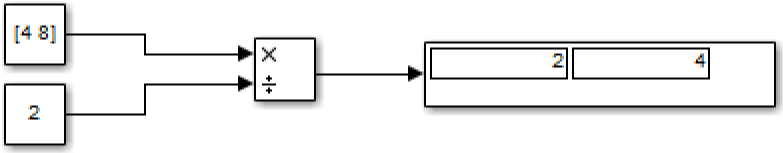
Parameters and Dialog Box

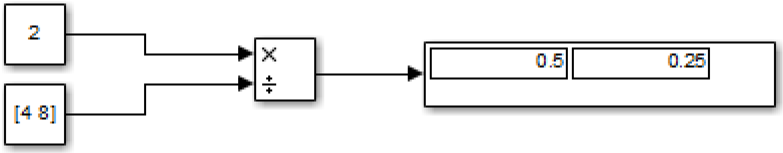
The Divide block has the same parameters and dialog box as the Product block. If all you need is to divide two inputs to create an output, you can use the Divide block with default parameter values. If you need additional capabilities, see the Product block documentation, which also describes the capabilities of the Divide block's "Signal Attributes Pane" on page 1-1310.

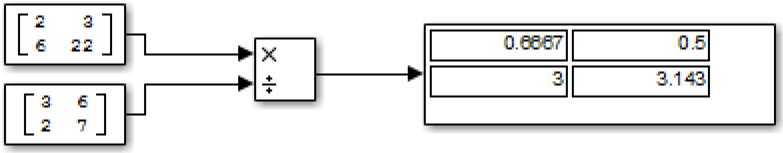
Examples

The following examples show the output of the Divide block for some typical inputs using default block parameter values.









DocBlock

Purpose

Create text that documents model and save text with model

Library

Model-Wide Utilities

Description

The DocBlock allows you to create and edit text that documents a model, and save that text with the model. Double-clicking an instance of the block creates a temporary file containing the text associated with this block and opens the file in an editor. Use the editor to modify the text and save the file. Simulink software stores the contents of the saved file in the model file.

The DocBlock supports HTML, Rich Text Format (RTF), and ASCII text document types. The default editors for these different document types are

- HTML — Microsoft® Word (if available). Otherwise, the DocBlock opens HTML documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- RTF — Microsoft Word (if available). Otherwise, the DocBlock opens RTF documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- Text — The DocBlock opens text documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.

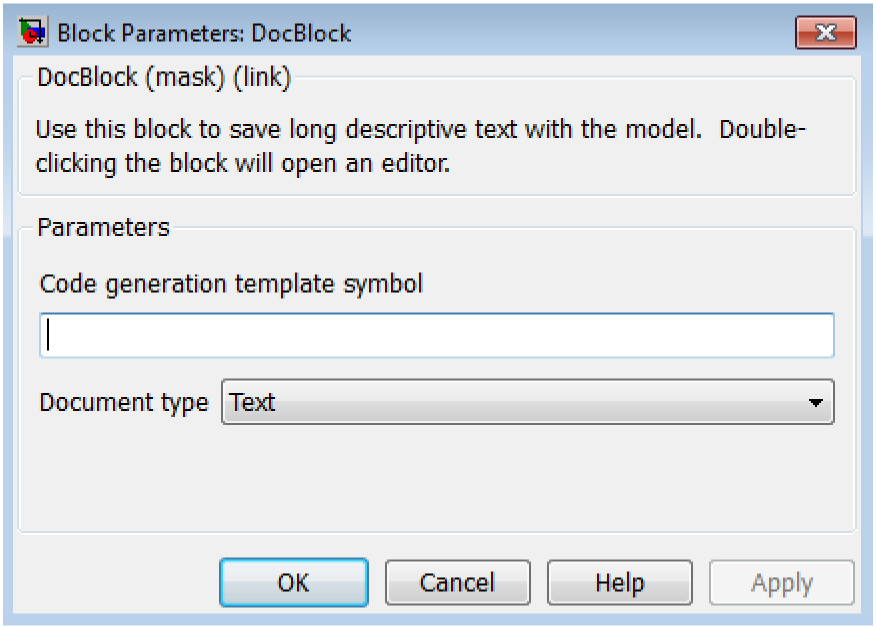
Use the `docblock` command to change the default editors.

Data Type Support

Not applicable.

Parameters and Dialog Box

Double-clicking an instance of the DocBlock opens an editor. To access the DocBlock parameter dialog box, select the block in the Model Editor and then select **Mask Parameters** from either the **Edit** menu or the block's context menu.



DocBlock (mask) (link)

Use this block to save long descriptive text with the model. Double-clicking the block will open an editor.

Parameters

Code generation template symbol

Document type: Text

Code generation template symbol (Embedded Coder® license required)

Enter a template symbol name in this field. Embedded Coder software uses this symbol to add comments to the code generated from the model. For more information, see "Add Global Comments".

Document type

Select the type of document associated with the DocBlock. The options are:

- `Text` (the default)

- `RTF`
- `HTML`

Note If you are using a DocBlock to add comments to your code during code generation, ensure that you set the **Document Type** as `Text`. If you set the **Document Type** as `RTF` or `HTML`, your comments will not appear in the code.

Characteristics Not applicable

Purpose

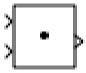
Generate dot product of two vectors

Library

Math Operations

Description

The Dot Product block generates the dot product of the vectors at its inputs. The scalar output, `y`, is equal to the MATLAB operation



```
y = sum(conj(u1) .* u2 )
```

where `u1` and `u2` represent the vectors at the block's top and bottom inputs, respectively. (See "How to Rotate a Block" in the Simulink documentation for a description of the port order for various block orientations.) The inputs can be vectors, column vectors (single-column matrices), or scalars. If both inputs are vectors or column vectors, they must be the same length. If `u1` and `u2` are both column vectors, the block outputs the equivalent of the MATLAB expression `u1'*u2`.

The elements of the input vectors can be real- or complex-valued signals. The signal type (complex or real) of the output depends on the signal types of the inputs.

Input 1	Input 2	Output
real	real	real
real	complex	complex
complex	real	complex
complex	complex	complex

To perform element-by-element multiplication without summing, use the Product block.

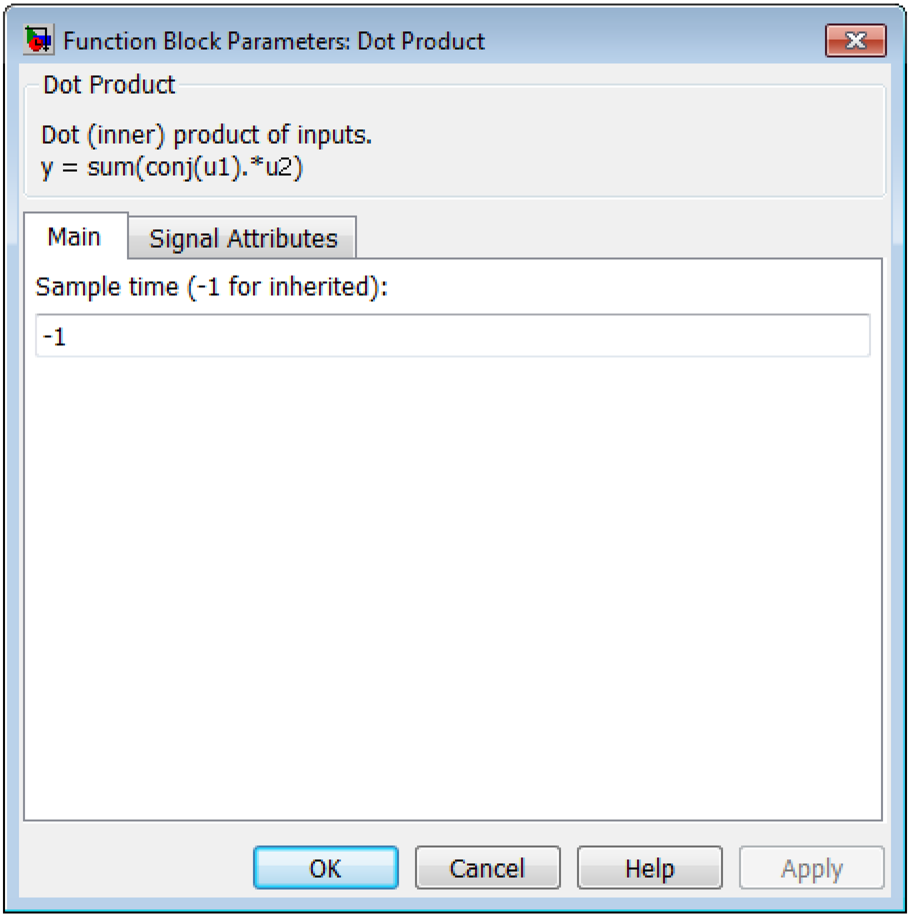
Data Type Support

The Dot Product block accepts and outputs signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see "Data Types Supported by Simulink".

Parameters and Dialog Box

The **Main** pane of the Dot Product block dialog box appears as follows:


Function Block Parameters: Dot Product

Dot Product

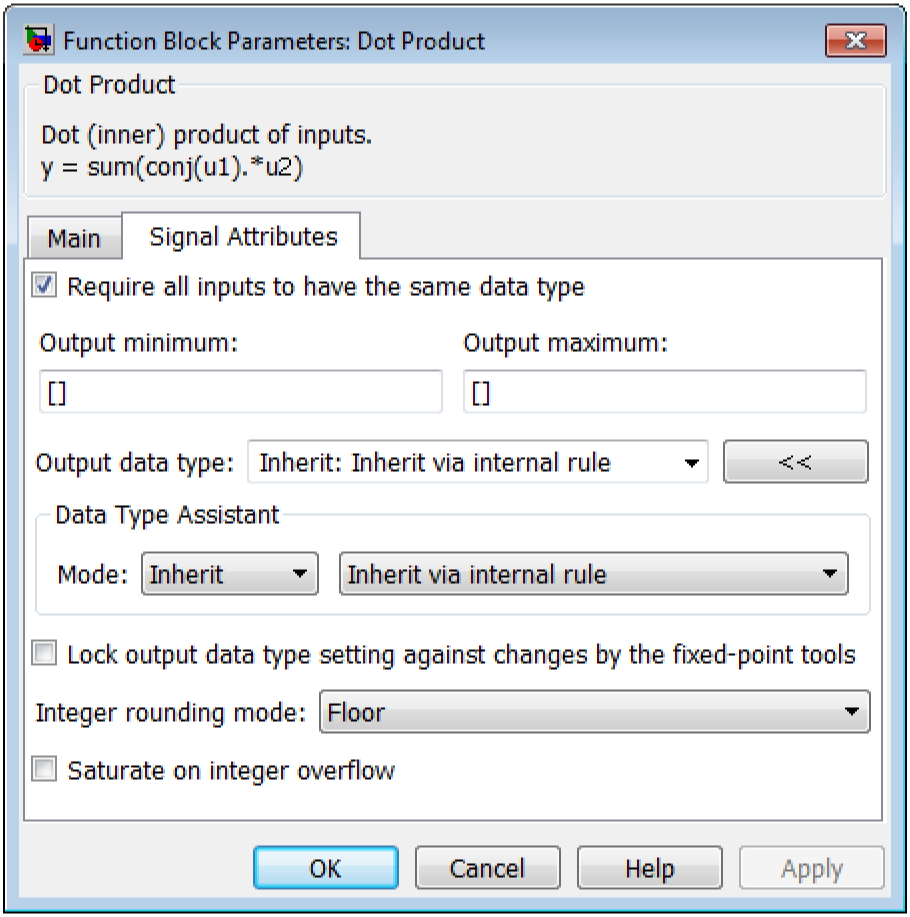
Dot (inner) product of inputs.
y = sum(conj(u1).*u2)

Main
Signal Attributes

Sample time (-1 for inherited): -1

OK
Cancel
Help
Apply

The **Signal Attributes** pane of the Dot Product block dialog box appears as follows:



Function Block Parameters: Dot Product

Dot Product

Dot (inner) product of inputs.
y = sum(conj(u1).*u2)

☑ Require all inputs to have the same data type

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

Data Type Assistant

Mode: Inherit — Inherit via internal rule

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

Integer rounding mode: Floor

☐ Saturate on integer overflow

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” in the Simulink documentation for more information.

Require all inputs to have same data type

Select to require all inputs to have the same data type.

Output minimum

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

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

Output maximum

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

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

Output data type

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

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

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

See “Specify Block Output Data Types” for more information.

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

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

Integer rounding mode

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

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 “Checking 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.

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

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
States	0
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also

Product

Purpose

Add enabling port to system

Library

Ports & Subsystems

Description

Adding an Enable block to a subsystem or at the root level of a model makes it an enabled system. A subsystem can contain no more than one Enable block. An enabled system executes while the input received at the Enable port is greater than zero.

At the start of a simulation, Simulink software initializes the states of blocks inside an enabled system to their initial conditions.

If you use an enable port for a root-level model:

- For multi-rate enabled models, set the solver to single-tasking.
- If the enabled model has a fixed-step size, at least one block in that model must run at that fixed-step size rate.

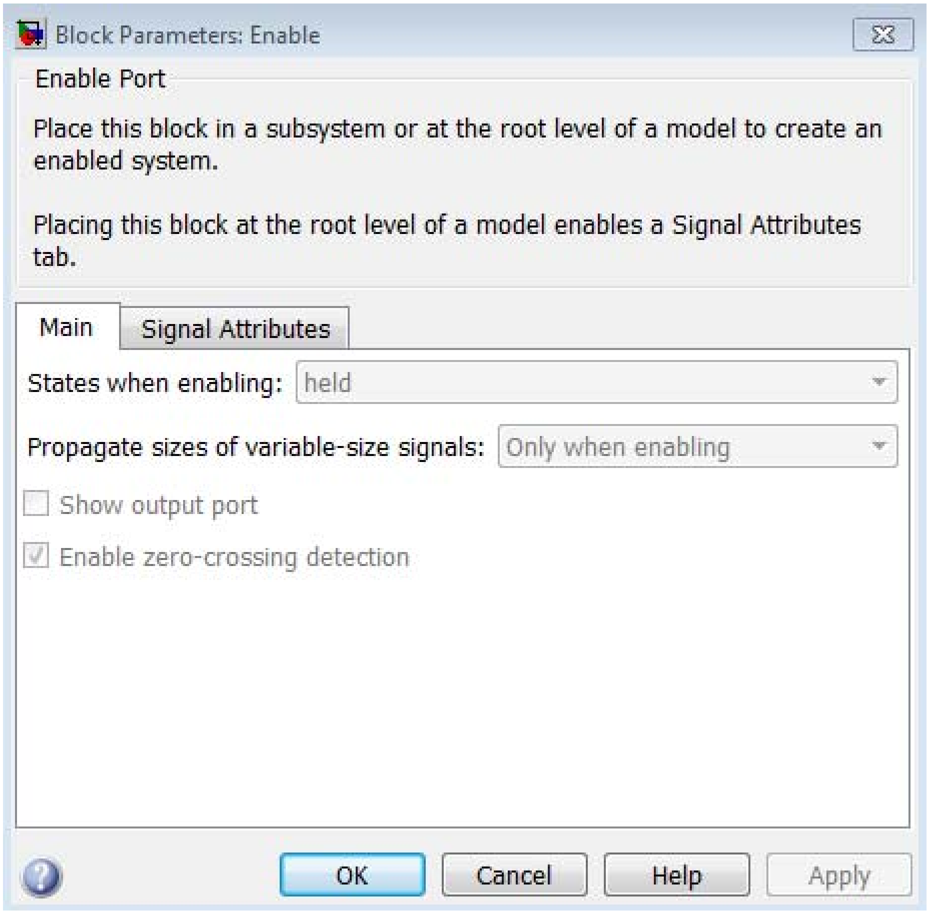
The Enable block supports signal label propagation.

Data Type Support

The Enable block accepts signals of supported Simulink numeric data types, including fixed-point data types. For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box

The **Main** pane of the Enable block dialog box appears as follows:



Enable Port

Place this block in a subsystem or at the root level of a model to create an enabled system.

Placing this block at the root level of a model enables a Signal Attributes tab.

States when enabling: held

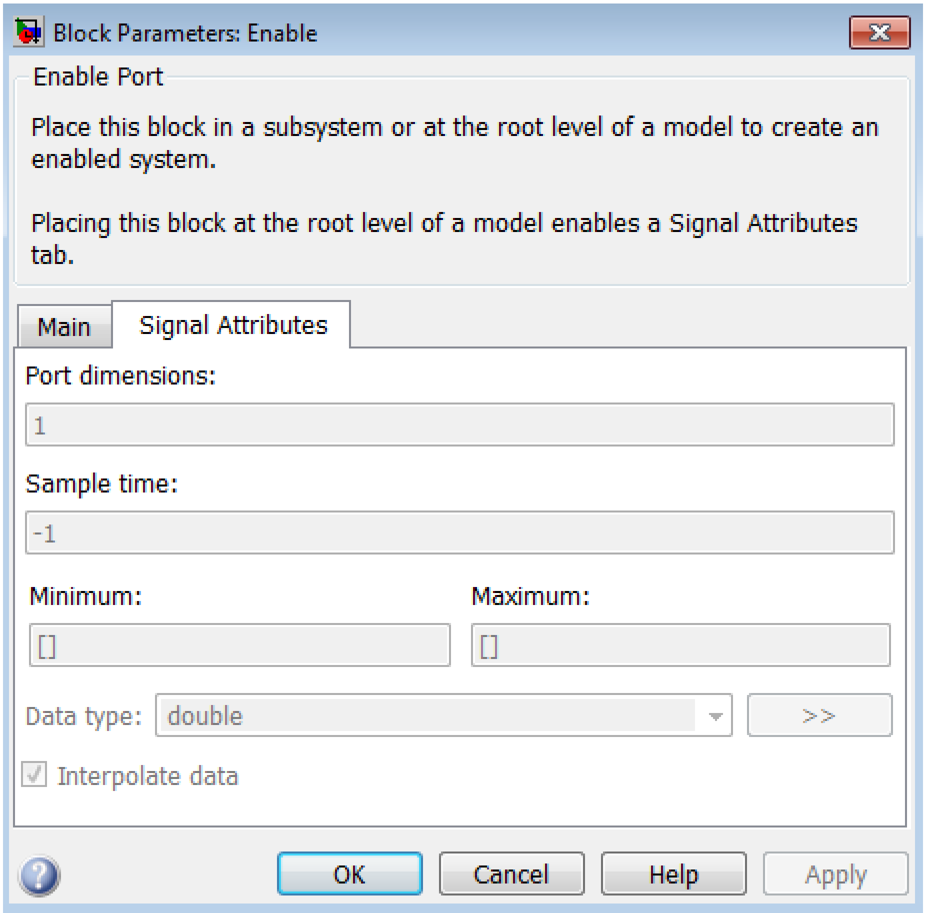
Propagate sizes of variable-size signals: Only when enabling

☐ Show output port

☑ Enable zero-crossing detection

Block Parameters: Enable
Main
Signal Attributes
OK
Cancel
Help
Apply

Placing the Enable block at the root of a model enables the **Signal Attributes** pane:



Block Parameters: Enable

Enable Port

Place this block in a subsystem or at the root level of a model to create an enabled system.

Placing this block at the root level of a model enables a Signal Attributes tab.

Port dimensions: 1

Sample time: -1

Minimum: []

Maximum: []

Data type: double

☑ Interpolate data

- “States when enabling” on page 1-567
- “Propagate sizes of variable-size signals” on page 1-568
- “Show output port” on page 1-569
- “Enable zero-crossing detection” on page 1-570
- “Port dimensions” on page 1-571

- “Sample time” on page 1-572
- “Minimum” on page 1-1920
- “Maximum” on page 1-1921
- “Data type” on page 1-575
- “Show data type assistant” on page 1-1672
- “Mode” on page 1-578
- “Interpolate data” on page 1-580

States when enabling

At the instant when an enabled system is being disabled, specify what happens to the states of blocks in the enabled system.

Settings

Default: `held`

`held`

Holds the states at their previous values.

`reset`

Resets the states to their initial conditions (zero if not defined).

Command-Line Information

Parameter: `StatesWhenEnabling`
Type: string
Value: `'held'` | `'reset'`
Default: `'held'`

Propagate sizes of variable-size signals

Specify when to propagate a variable-size signal.

Settings

Default: `Only when enabling`

`Only when enabling`
: Propagates variable-size signals only when reenabling the system containing the Enable Port block. When you select this option, sample time must be periodic.

`During execution`
: Propagates variable-size signals at each time step.

Command-Line Information

Parameter: `PropagateVarSize`
Type: string
Value: `'Only when enabling' | 'During execution'`
Default: `'Only when enabling'`

Show output port

Select this check box to output the enabling signal.

Settings

Default: On

☑ On

Shows the Enable block output port and outputs the enabling signal. Selecting this option allows the system to process the enabling signal.

☐ Off

Removes the output port from the Enable block.

Command-Line Information

Parameter: `ShowOutputPort`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Enable zero-crossing detection

Select this check box to enable zero-crossing detection.

Settings

Default: On

☑ On

Detect zero crossings.

☐ Off

Do not detect zero crossings.

Command-Line Information

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Port dimensions

Specify the dimensions of the input signal to the block.

Settings

Default: `1`

Valid values are:

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time

Specify the time interval between samples.

Settings

Default: `-1`

See “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Minimum

Specify the minimum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

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

Simulink software uses this value to perform:

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Maximum

Specify the maximum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

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

Simulink software uses this value to perform:

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Data type

Specify the output data type of the external input.

Settings

Default: `double`

`double`
: Data type is `double`.

`single`
: Data type is `single`.

`int8`
: Data type is `int8`.

`uint8`
: Data type is `uint8`.

`int16`
: Data type is `int16`.

`uint16`
: Data type is `uint16`.

`int32`
: Data type is `int32`.

`uint32`
: Data type is `uint32`.

`boolean`
: Data type is `boolean`.

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

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

`<data type expression>`
: The name of a data type object, for example `Simulink.NumericType`

Do not specify a bus object as the expression.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

Settings

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

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `double`

`Built in`

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

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

`Fixed point`

Fixed-point data types.

`Expression`

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

Do not specify a bus object as the expression.

Dependency

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Interpolate data

Cause the block to interpolate or extrapolate output at time steps for which no corresponding workspace data exists when loading data from the workspace.

Settings

Default: On

☑ On

Cause the block to interpolate or extrapolate output at time steps for which no corresponding workspace data exists when loading data from the workspace.

☐ Off

Do not cause the block to interpolate or extrapolate output at time steps for which no corresponding workspace data exists when loading data from the workspace.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

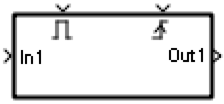
Characteristics

Sample Time	Determined by the signal at the enable port
Dimensionalized	Yes
Virtual	Yes, if not connected directly to an Outport block For more information, see “Virtual Blocks” in the Simulink documentation.
Zero-Crossing Detection	Yes, if enabled

Purpose Represent subsystem whose execution is enabled and triggered by external input

Library Ports & Subsystems

Description



This block is a Subsystem block that is preconfigured to serve as the starting point for creating an enabled and triggered subsystem. For more information, see “Triggered and Enabled Subsystems” in the online Simulink help.

Purpose Represent subsystem whose execution is enabled by external input

Library Ports & Subsystems

Description This block is a Subsystem block that is preconfigured to serve as the starting point for creating an enabled subsystem. For more information, see “Enabled Subsystems” in the “Creating a Model” chapter of the Simulink documentation.

Purpose

Generate enumerated constant value

Library

Sources

Description

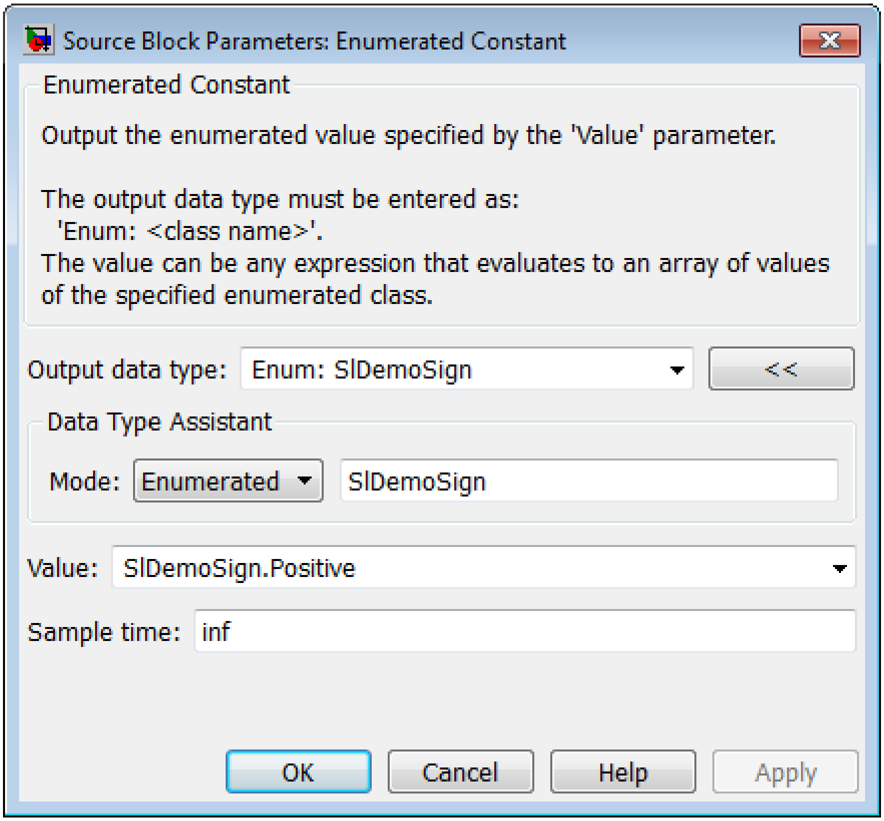
The Enumerated Constant block outputs a scalar, array, or matrix of enumerated values. You can also use the Constant block to output enumerated values, but it provides block parameters that do not apply to enumerated types, such as **Output minimum** and **Output maximum**. When you need a block that outputs only constant enumerated values, preferably use Enumerated Constant rather than Constant. For more information, see “About Simulink Enumerations”.

Data Type Support

The Enumerated Constant block supports only enumerated data types. Use the Constant block to output constant data of other types. For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box

The Enumerated Constant block dialog box initially appears as follows:



Source Block Parameters: Enumerated Constant
Enumerated Constant

Output the enumerated value specified by the 'Value' parameter.

The output data type must be entered as:
'Enum: <class name>'.
The value can be any expression that evaluates to an array of values of the specified enumerated class.

Output data type: Enum: SlDemoSign

<<
Data Type Assistant

Mode: Enumerated SlDemoSign

Value: SlDemoSign.Positive

Sample time: inf

OK
Cancel
Help
Apply

Output data type

The **Output data type** field specifies the enumerated type from which you want the block to output one or more values. The initial value, `Enum: SlDemoSign`, is a dummy enumerated type that prevents a newly cloned block from causing an error. To specify the desired enumerated type, select it from the pulldown or enter `Enum:` *`ClassName`* in the **Output data type** field, where *`ClassName`* is the name of the MATLAB class that defines the type.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Value

The **Value** field specifies the value(s) that the block outputs. The output of the block has the same dimensions and elements as the **Value** parameter. The initial value, `SlDemoSign.Positive`, is a dummy enumerated value that prevents a newly cloned block from causing an error.

To specify the desired enumerated value(s), select from the pulldown or enter any MATLAB expression that evaluates to the desired result, including an expression that uses tunable parameters. All values specified must be of the type indicated by the **Output data type**. To specify an array that includes every value in the enumerated type, use the `enumeration` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sample time

Specify the interval between times that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

Settings

Default: `inf`

This setting indicates that the block output can never change. This setting speeds simulation and generated code by avoiding the need to recompute the block output. See “Specify Sample Time” for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Characteristics

Direct Feedthrough	N/A
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

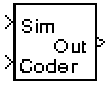
Purpose

Create branches of block diagram that apply only to simulation or only to code generation

Library

Signal Routing

Description



This block outputs the signal at its Sim port only if the model that contains it is being simulated. It outputs the signal at its Coder port only if code is being generated from the model. This option enables you to create branches of a block diagram that apply only to simulation or code generation. The table below describes various scenarios where either the Sim or Coder port applies.

Scenario	Output
Normal mode simulation	Sim
Accelerator mode simulation	Sim
Rapid Accelerator mode simulation	Coder
Simulation of a referenced model (Normal or Accelerator modes)	Sim
Simulation of a referenced model (Processor-in-the-loop (PIL) mode)	Coder (uses the same code generated for a referenced model)
External mode simulation	Coder
Standard code generation	Coder
Code generation of a referenced model	Coder

Simulink Coder software does not generate code for blocks connected to the Sim port if these conditions hold:

- You select the **Inline parameters** check box on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.
- The blocks connected to the Sim port do not have external signals.
- The Sim port input path does not contain a MATLAB S-function or an Interpreted MATLAB Function block.

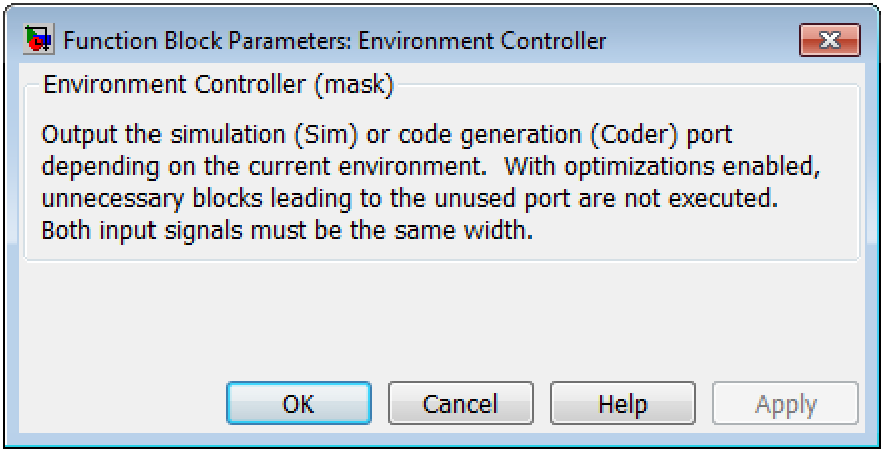
If you enable block reduction optimization, Simulink eliminates blocks in the branch connected to the Coder port when compiling the model for simulation. For information about block reduction, see "Block reduction" in the online Simulink documentation.

Note Simulink Coder code generation eliminates the blocks connected to the Sim branch only if the Sim branch has the same signal dimensions as the Coder branch. Regardless of whether it eliminates the Sim branch, Simulink Coder uses the sample times on the Sim branch as well as the Coder branch to determine the fundamental sample time of the generated code and might, in some cases, generate sample-time handling code that applies only to sample times specified on the Sim branch.

Data Type Support

The Environment Controller block accepts signals of any data type that Simulink supports. The output uses the same data type as the input.

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



Function Block Parameters: Environment Controller
Environment Controller (mask)
OK
Cancel
Help
Apply

Output the simulation (Sim) or code generation (Coder) port depending on the current environment. With optimizations enabled, unnecessary blocks leading to the unused port are not executed. Both input signals must be the same width.

Parameters and Dialog Box

Characteristics

Multidimensionalized	Yes

Purpose Output selection of contiguous bits from input signal

Library Logic and Bit Operations

Description

Extract Bits
Upper Half

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.

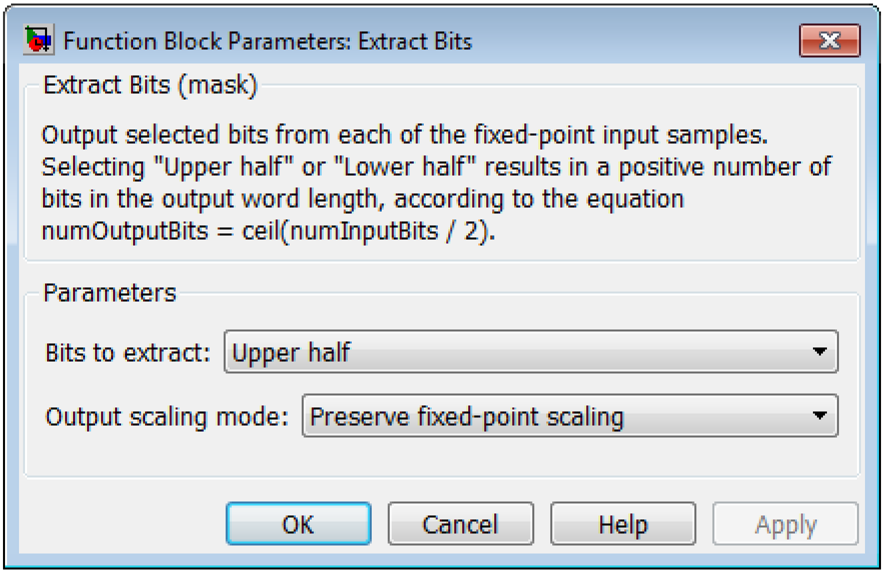
- Select `Range of bits` to indicate a series of contiguous bits of the input to output in the **Bit indices** parameter. You indicate the range in `[start end]` format, and the indices of the input bits are labeled contiguously starting at 0 for the least significant bit.

Data Type Support

The Extract Bits block accepts inputs of any numeric data type that Simulink supports, including fixed-point data types. Floating-point inputs are passed through the block unchanged. `Boolean` inputs are treated as `uint8` signals.

Note Performing bit operations on a signed integer is difficult. You can avoid difficulty by converting the data type of your input signals to unsigned integer types.

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



Function Block Parameters: Extract Bits

Extract Bits (mask)

Output selected bits from each of the fixed-point input samples. Selecting "Upper half" or "Lower half" results in a positive number of bits in the output word length, according to the equation numOutputBits = ceil(numInputBits / 2).

Parameters

Bits to extract: Upper half

Output scaling mode: Preserve fixed-point scaling

Parameters and Dialog Box

Bits to extract

Select the method for extracting bits from the input signal.

Number of bits

(Not shown on dialog above.) Select the number of bits to output from the input signal. Signed integer data types can have no less then two bits in them. Unsigned data integer types can be as short as a single bit.

This parameter is only visible if you select `Range starting with most significant bit` or `Range ending with least significant bit` for the **Bits to extract** parameter.

Bit indices

(Not shown on dialog above.) Specify a contiguous range of bits of the input signal to output. Specify the range in `[start end]` format. The indices are assigned to the input bits starting with 0 at the least significant bit.

This parameter is only visible if you select `Range of bits` for the **Bits to extract** parameter.

Output scaling mode

Select the scaling mode to use on the output bits selection:

- When you select `Preserve fixed-point scaling`, the fixed-point scaling of the input is used to determine the output scaling during the data type conversion.
- When you select `Treat bit field as an integer`, the fixed-point scaling of the input is ignored, and only the stored integer is used to compute the output data type.

Example

Consider an input signal that is represented in binary by `110111001`:

- If you select `Upper half` for the **Bits to extract** parameter, the output is `11011` in binary.
- If you select `Lower half` for the **Bits to extract** parameter, the output is `11001` in binary.
- If you select `Range starting with most significant bit` for the **Bits to extract** parameter, and specify `3` for the **Number of bits** parameter, the output is `110` in binary.
- If you select `Range ending with least significant bit` for the **Bits to extract** parameter, and specify `8` for the **Number of bits** parameter, the output is `10111001` in binary.
- If you select `Range of bits` for the **Bits to extract** parameter, and specify `[4 7]` for the **Bit indices** parameter, the output is `1011` in binary.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited
Scalar Expansion	N/A

States	None
Dimensionalized	Inherited
Multidimensionalized	Yes
Zero Crossing	No

Purpose

Apply specified expression to input

Library

User-Defined Functions

Description

The Fcn block applies the specified mathematical expression to its input. The expression can include one or more of these components:

- `u` — The input to the block. If `u` is a vector, `u(i)` represents the `i`th element of the vector; `u(1)` or `u` alone represents the first element.
- Numeric constants.
- Arithmetic operators (+ - * / ^).
- Relational operators (`== != > < >= <=`) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Logical operators (&& `||` !) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Parentheses.
- Mathematical functions — `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `hypot`, `ln`, `log`, `log10`, `pow`, `power`, `rem`, `sgn`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.

Note The Fcn block does not support `round` and `fix`. Use the Rounding Function block to apply these rounding modes.

- Workspace variables — Variable names that are not recognized in the preceding list of items are passed to MATLAB for evaluation. Matrix or vector elements must be specifically referenced (e.g., `A(1,1)` instead of `A` for the first element in the matrix).

The Fcn block observes the following rules of operator precedence:

1. `( )`
2. `^`
3. `+ -` (unary)
4. `!`
5. `* /`
6. `+ -`
7. `> < <= >=`
8. `== !=`
9. `&&`
10. `||`

The expression differs from a MATLAB expression in that the expression cannot perform matrix computations. Also, this block does not support the colon operator (:).

Block input can be a scalar or vector. The output is always a scalar. For vector output, consider using the Math Function block. If a block input is a vector and the function operates on input elements individually (for example, the `sin` function), the block operates on only the first vector element.

Limitations

The Fcn block has the following limitations:

- You cannot tune the expression during simulation in Normal or Accelerator mode (see “How Acceleration Modes Work”), or in generated code. The Fcn block doesn’t support tunable parameters if **Inline Parameters** is turned on. To implement tunable expressions, tune the expression outside the Fcn block. For example, use the Relational Operator block to evaluate the expression outside.

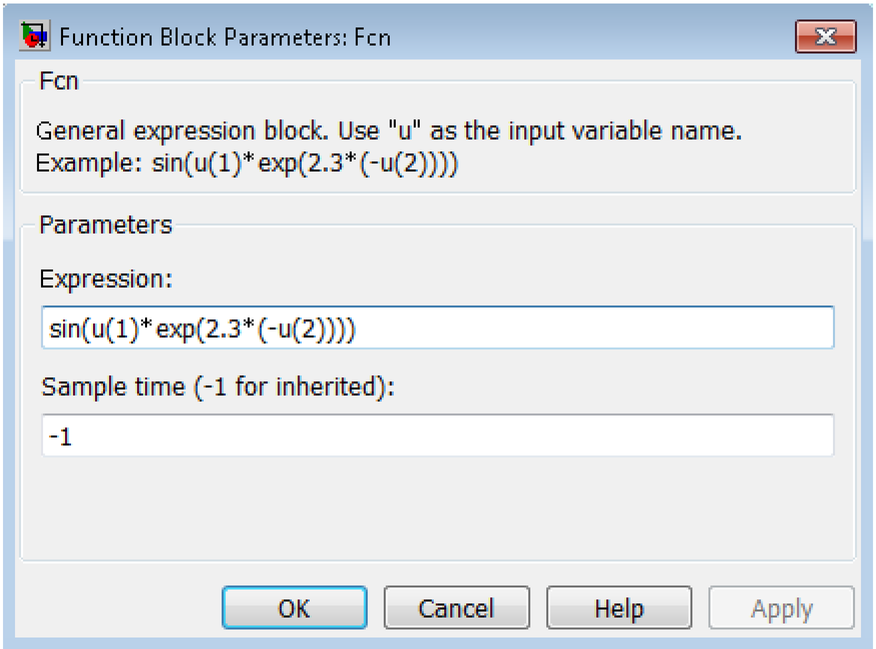
- The Fcn block does not support custom storage classes. See “Custom Storage Classes” in the Embedded Coder documentation.

Data Type Support

The Fcn block accepts and outputs signals of type `single` or `double`.

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

Parameters and Dialog Box



Function Block Parameters: Fcn

Fcn

General expression block. Use "u" as the input variable name.
Example: sin(u(1)*exp(2.3*(-u(2))))

Parameters

Expression:

sin(u(1)*exp(2.3*(-u(2))))

Sample time (-1 for inherited):

-1

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 1-596)

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Examples

The following example models show how to use the Fcn block:

- `sldemo_absbrake`
- `sldemo_enginewc` (Throttle & Manifold/Throttle subsystem)

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	No
Multidimensionalized	No
Zero-Crossing Detection	No

Purpose

Find nonzero elements in array

Library

Math Operations

Description

The Find block locates all nonzero elements of the input signal and returns the linear indices of those elements. If the input is a multidimensional signal, the Find block can also return the subscripts of the nonzero input elements. In both cases, you can show an output port with the nonzero input values.

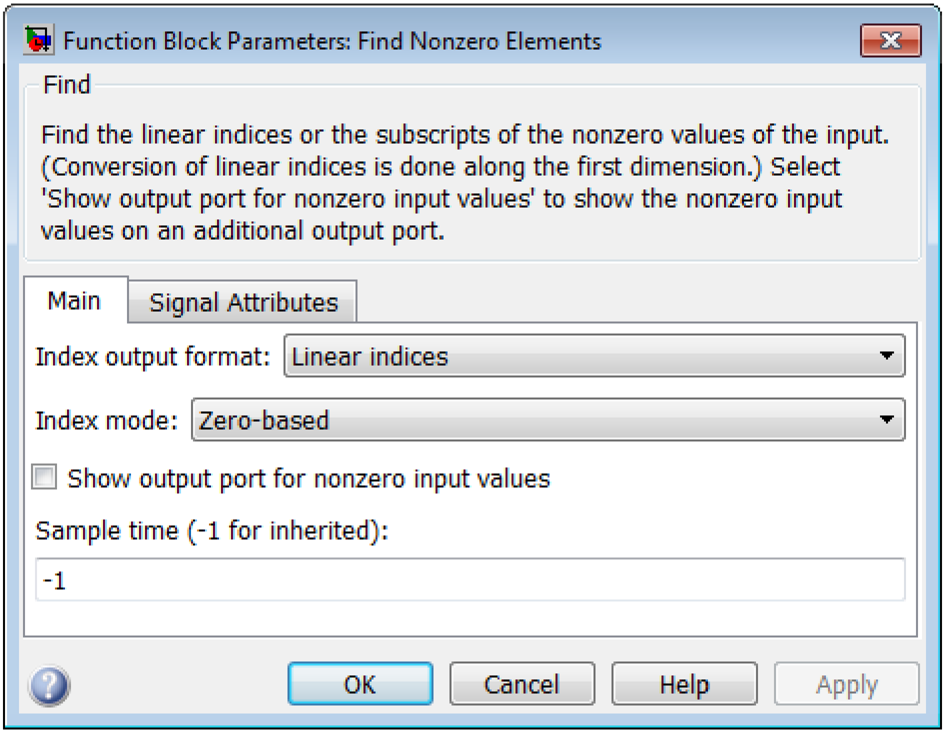
Data Type Support

The Find block accepts and outputs real values of any numeric data type that Simulink supports.

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

Parameters and Dialog Box

The **Main** pane of the Find block dialog box appears as follows:



Function Block Parameters: Find Nonzero Elements

Find

Find the linear indices or the subscripts of the nonzero values of the input. (Conversion of linear indices is done along the first dimension.) Select 'Show output port for nonzero input values' to show the nonzero input values on an additional output port.

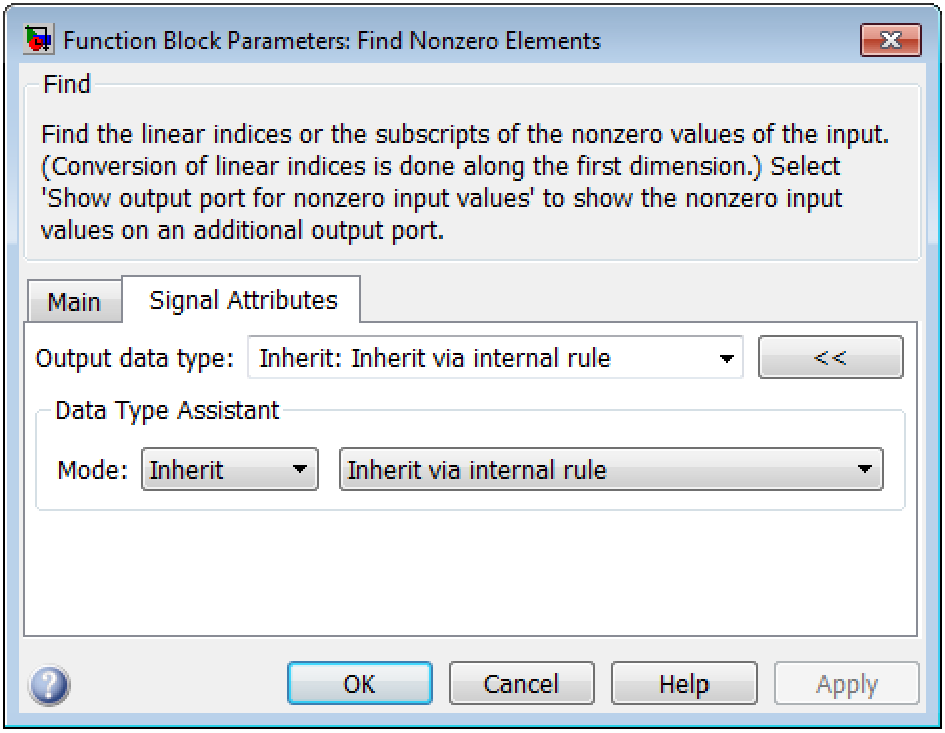
Index output format: Linear indices

Index mode: Zero-based

☐ Show output port for nonzero input values

Sample time (-1 for inherited): -1

The **Signal Attributes** pane of the Find block dialog appears as follows:


Function Block Parameters: Find Nonzero Elements
Main
Signal Attributes
<<
OK
Cancel
Help
Apply

Find

Find the linear indices or the subscripts of the nonzero values of the input. (Conversion of linear indices is done along the first dimension.) Select 'Show output port for nonzero input values' to show the nonzero input values on an additional output port.

Output data type: Inherit: Inherit via internal rule

Data Type Assistant

Mode: Inherit · Inherit via internal rule

- “Index output format” on page 1-604
- “Number of input dimensions” on page 1-605
- “Index mode” on page 1-606
- “Show output port for nonzero input values” on page 1-607
- “Sample time (-1 for inherited)” on page 1-608
- “Output data type” on page 1-609
- “Mode” on page 1-611
- “Data type override” on page 1-1926
- “Signedness” on page 1-614

- “Word length” on page 1-615
- “Scaling” on page 1-616

Index output format

Select the output format for the indices of the nonzero input values.

Settings

Default: `Linear indices`

`Linear indices`

Provides the element indices of any dimension signal in a vector form. For one dimension (vector) signals, indices correspond to the position of nonzero values within the vector. For signals with more than one dimension, the conversion of subscripts to indices is along the first dimension. You do not need to know the signal dimension of the input signal.

`Subscripts`

Provides the element indices of a two-dimension or larger signal in a subscript form. Because the block shows an output port for each dimension, this option requires you to know the number of dimensions for the input signal.

Dependencies

Selecting `Subscripts` from the **Index output format** list enables the **Number of input dimensions** parameter.

Command-Line Information

Parameter: `IndexOutputFormat`
Type: string
Value: `Linear indices` | `Subscripts`
Default: `Linear indices`

Number of input dimensions

Specify the number of dimensions for the input signal.

Settings

Default: `1`

Minimum: `1`

Maximum: 32

Dependencies

Selecting `Subscripts` from the **Index output format** list enables this parameter.

Command-Line Information

Parameter: `NumberOfInputDimensions`
Type: int
Value: positive integer value
Default: `1`

Index mode

Specify the indexing mode.

Settings

Default: `Zero-based`

`Zero-based`

An index of `0` specifies the first element of the input vector. An index of `1` specifies the second element, and so on.

`One-based`

An index of `1`specifies the first element of the input vector. An index of `2`, specifies the second element , and so on.

Command-Line Information

Parameter: `IndexMode`
Type: string
Value: `Zero-based` | `One-based`
Default: `Zero-based`

Show output port for nonzero input values

Show or hide the output port for nonzero input values.

Settings

Default: Off

☑ On

Display the output port for nonzero input values. The additional output port provides the values of the nonzero input elements.

☐ Off

Hide the output port for nonzero input values.

Command-Line Information

Parameter: `ShowOutputPortForNonzeroInputValues`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the “How Simulink Works” chapter of the Simulink documentation.

Command-Line Information

Parameter: `SampleTime`
Type: string
Value: `-1` (for inherited)| positive number
Default: `-1`

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Inherit via internal rule`

`Inherit:  Inherit via internal rule`

Output data type is defined by the target.

`int8`

Output data type is `int8`.

`uint8`

Output data type is `uint8`.

`int16`

Output data type is `int16`.

`uint16`

Output data type is `uint16`.

`int32`

Output data type is `int32`.

`uint32`

Output data type is `unt32`.

`fixdt(1,16)`

Output data type is fixed point, `fixdt(1,16)`.

`<data type expression>`

Use a data type object, for example, `Simulink.NumericType`.

Click the **Show data type assistant** button >> to display additional parameters for the **Output data type** parameter.

Command-Line Information

Parameter: `OutDataTypeStr`
Type: string

Value: `'Inherit:  Inherit via internal rule'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'`| `'fixdt(1,16)'`| `'<data type expression>'`

Default: `'Inherit:  Inherit via internal rule'`

See Also

"Specify Block Output Data Types", "Specify Data Types Using Data Type Assistant"

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second list of the possible values:

- `Inherit via internal rule` (Discrete-Time Integrator, Gain, Product, Sum, Switch block default)

`Built in`

Built-in data types. Selecting `Built in` enables a second list of the possible values:

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

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

Dependencies

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

Selecting `Fixed point` from the **Mode** list enables the following parameters:

- **Signed**

- **Scaling**
- **Word length**

See Also

"Specify Data Types Using Data Type Assistant"

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

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

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

Tip

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

Dependency

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

Signedness

Specify whether the fixed-point data is signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specifies the fixed-point data as signed.

`Unsigned`
: Specifies the fixed-point data as unsigned.

Dependency

Selecting `Fixed point` from the **Mode** list enables this parameter.

Word length

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

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Dependency

Selecting `Fixed point` from the **Mode** list enables this parameter.

Scaling

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

Settings

Default: `Integer`

`Integer`
: Specifies a binary point location for fixed-point data and sets the fraction length to 0.

The **Scaling** list has only one item for you to select.

Dependency

Selecting `Fixed point` from the **Mode** list enables this parameter.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

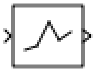
Purpose

Implement first-order sample-and-hold

Library

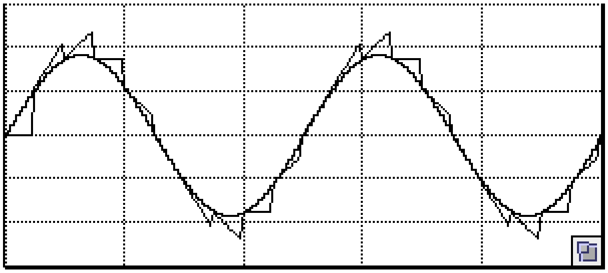
Discrete

Description



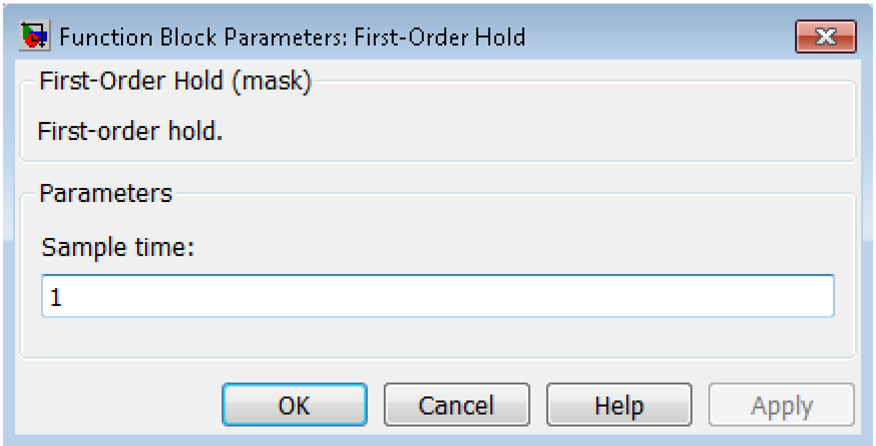
The First-Order Hold block implements a first-order sample-and-hold that operates at the specified sampling interval. This block has little value in practical applications and is included primarily for academic purposes.

This figure compares the output from a Sine Wave block and a First-Order Hold block.



Data Type Support

The First-Order Hold block accepts and outputs signals of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: First-Order Hold

First-Order Hold (mask)

First-order hold.

Parameters

Sample time:

1

Parameters and Dialog Box

Sample time

The time interval between samples. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	No
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
States	1 continuous and 1 discrete per input element
Dimensionalized	Yes
Zero Crossing	No

See Also

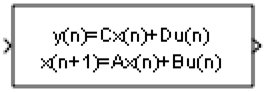
Zero-Order Hold

Fixed-Point State-Space

Purpose Implement discrete-time state space

Library 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.

The matrices A, B, C and D have the following characteristics:

- A must be an n-by-n matrix, where n is the number of states.
- B must be an n-by-m matrix, where m is the number of inputs.
- C must be an r-by-n matrix, where r is the number of outputs.
- D must be an r-by-m matrix.

In addition:

- The state x must be an n-by-1 vector.
- The input u must be an m-by-1 vector.
- The output y must be an r-by-1 vector.

The block accepts one input and generates one output. The block determines the input vector width by the number of columns in the B and D matrices. Similarly, the block determines the output vector width by the number of rows in the C and D matrices.

Data Type Support

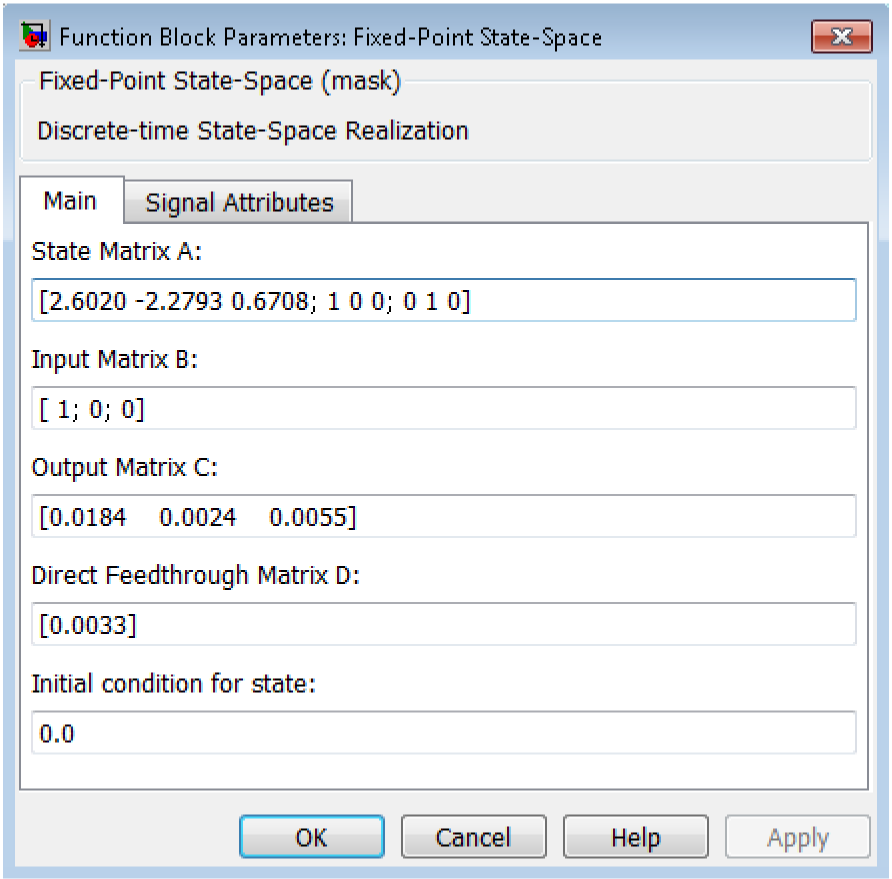
The Fixed-Point State-Space block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box

The **Main** pane of the Fixed-Point State-Space block dialog box appears as follows:



Function Block Parameters: Fixed-Point State-Space
Fixed-Point State-Space (mask)
Discrete-time State-Space Realization
Main
Signal Attributes
OK
Cancel
Help
Apply

State Matrix A: [2.6020 -2.2793 0.6708; 1 0 0; 0 1 0]

Input Matrix B: [1; 0; 0]

Output Matrix C: [0.0184   0.0024   0.0055]

Direct Feedthrough Matrix D: [0.0033]

Initial condition for state: 0.0

State Matrix A
Specify the matrix of states.

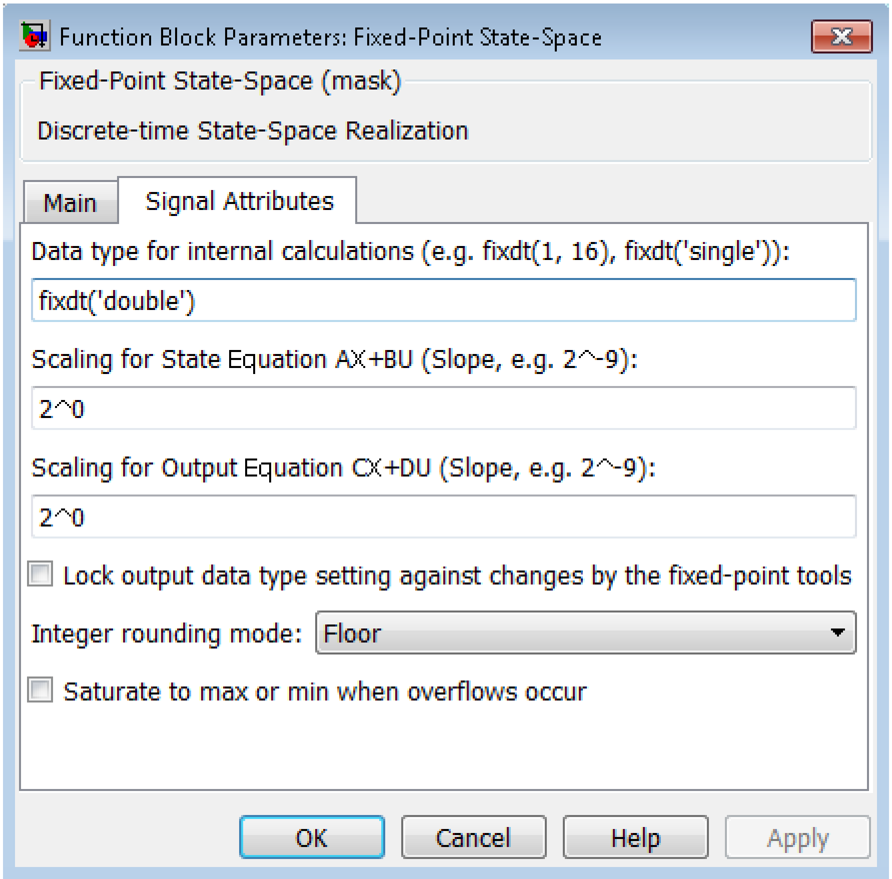
Input Matrix B
Specify the column vector of inputs.

Output Matrix C
Specify the column vector of outputs.

Direct Feedthrough Matrix D
Specify the matrix for direct feedthrough.

Initial condition for state
Specify the initial condition for the state.

The **Signal Attributes** pane of the Fixed-Point State-Space block dialog box appears as follows:


Function Block Parameters: Fixed-Point State-Space
Fixed-Point State-Space (mask)
Main
Signal Attributes
OK
Cancel
Help
Apply

Discrete-time State-Space Realization

Data type for internal calculations (e.g. fixdt(1, 16), fixdt('single')): fixdt('double')

Scaling for State Equation AX+BU (Slope, e.g. 2^-9): 2^0

Scaling for Output Equation CX+DU (Slope, e.g. 2^-9): 2^0

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

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Data type for internal calculations
Specify the data type for internal calculations.

Scaling for State Equation AX+BU
Specify the scaling for state equations.

Scaling for Output Equation CX+DU
Specify the scaling for output equations.

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

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

Integer rounding mode

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

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

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

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of initial conditions
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Discrete State-Space

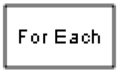
Purpose

Enable blocks inside For Each Subsystem to process elements or subarrays of input signal independently

Library

Ports & Subsystems

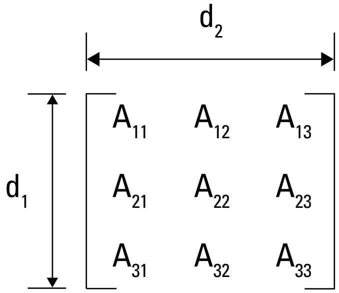
Description



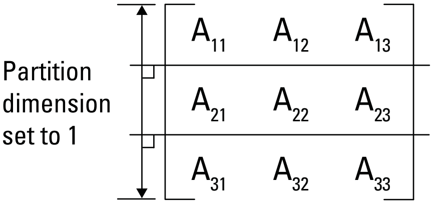
The For Each block serves as a control block for the For Each Subsystem block. Specifically, the For Each block enables the blocks inside the For Each Subsystem to process elements or subarrays of the input signals independently. Each block inside this subsystem that has states maintains a separate set of states for each element or subarray it processes. As the set of blocks in the subsystem process the elements (or subarrays), the subsystem concatenates the results to form output signals.

Using the For Each block, you specify how to decompose each input signal to the subsystem into elements or subarrays by setting the integer values of the "Partition Dimension" on page 1-632 and the "Partition Width" on page 1-633. Similarly, you define the dimension along which to concatenate the results by specifying the "Concatenation Dimension" on page 1-635.

As an illustration, consider an input signal A of the form:



The labels d_1 and d_2, respectively define dimension 1 and dimension 2. If you retain the default setting of 1 for both the partition dimension and the partition width, then Simulink slices perpendicular to partition dimension d_1 at a width equal to the partition width, one element.



Input signal A decomposes into the following three row vectors:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \end{bmatrix}$$

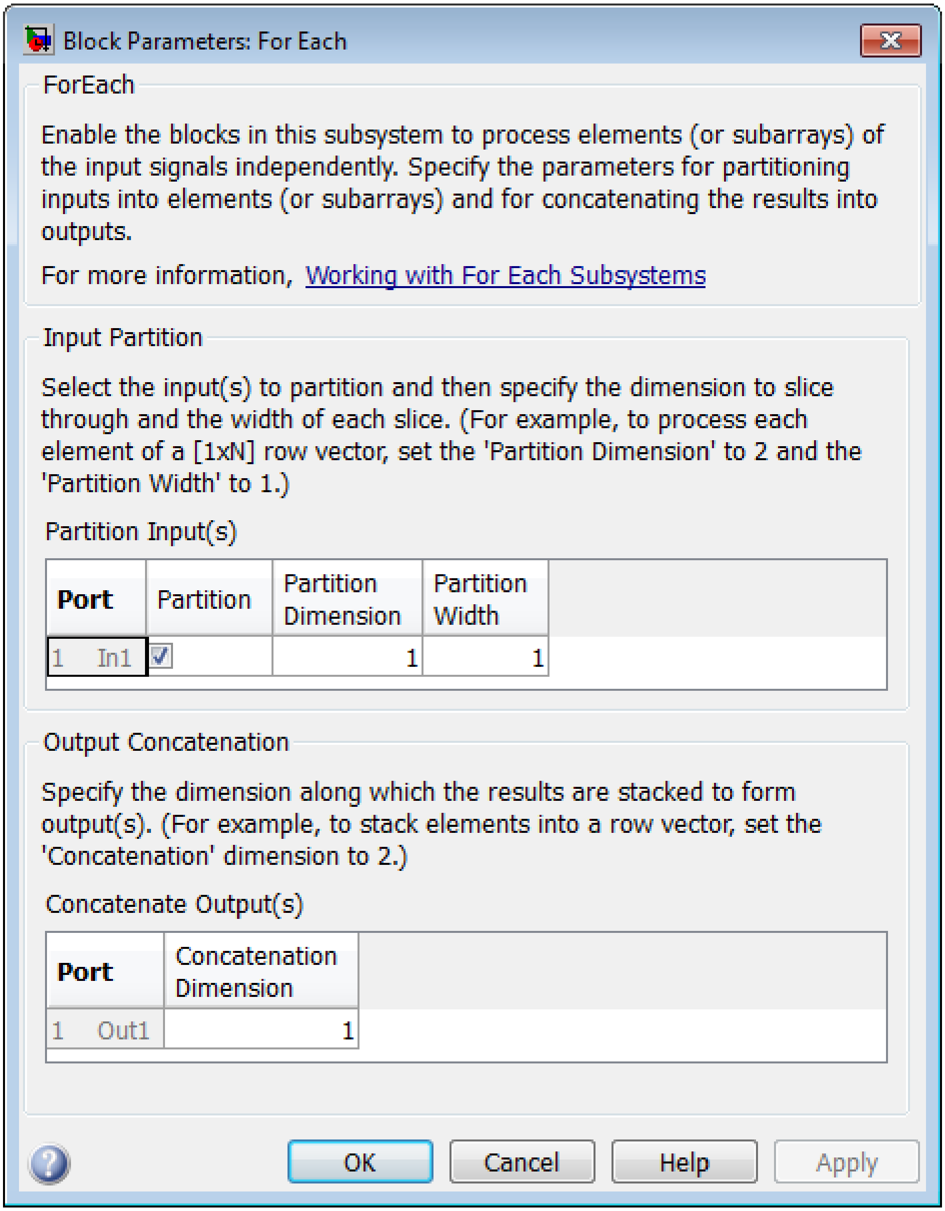
$$\begin{bmatrix} A_{21} & A_{22} & A_{23} \end{bmatrix}$$

$$\begin{bmatrix} A_{31} & A_{32} & A_{33} \end{bmatrix}$$

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 either case, 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.



Block Parameters: For Each

ForEach

Enable the blocks in this subsystem to process elements (or subarrays) of the input signals independently. Specify the parameters for partitioning inputs into elements (or subarrays) and for concatenating the results into outputs.

For more information, Working with For Each Subsystems

Input Partition

Select the input(s) to partition and then specify the dimension to slice through and the width of each slice. (For example, to process each element of a [1xN] row vector, set the 'Partition Dimension' to 2 and the 'Partition Width' to 1.)

Partition Input(s)

Port	Partition	Partition Dimension	Partition Width
1 In1	☑	1	1

Output Concatenation

Specify the dimension along which the results are stacked to form output(s). (For example, to stack elements into a row vector, set the 'Concatenation' dimension to 2.)

Concatenate Output(s)

Port	Concatenation Dimension
1 Out1	1

Parameters and Dialog Box

- "Partition Input(s)" on page 1-629
- "Port" on page 1-630

- “Partition” on page 1-631
- “Partition Dimension” on page 1-632
- “Partition Width” on page 1-633
- “Concatenate Output(s)” on page 1-634
- “Port” on page 1-634
- “Concatenation Dimension” on page 1-635

Partition Input(s)

Use this table to select each input signal that you need to partition and to specify the corresponding **Partition Dimension** and **Partition Width** parameters.

Port

The **Port** column displays the input index and the name of the input port connected to the For Each Subsystem block.

Tip

You can have any number of ports.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Partition

Select the check box beside each input signal that you want to partition into subarrays or elements.

Settings

Default: Off

☑ On
  Enables the partitioning of the input signal.

☐ Off
  Disables the partitioning of the input signal.

Dependencies

Selecting this check box enables the **Partition Dimension** and **Partition Width** parameters.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Partition Dimension

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.

Settings

Default: 1

Minimum: 1

Tip

Specify the corresponding width of each slice by setting the **Partition Width** parameter for a given input port.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Partition Width

Specify the width of each partition slice of the input signal.

Settings

Default: `1`

Minimum: `1`

Tips

- The default width of `1` represents a width of one element.
- Specify this parameter with the **Partition Dimension** parameter to define fully how to create the partitioned signal.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Concatenate Output(s)

For each output port, specify the dimension along which to stack (concatenate) the For Each Subsystem results.

Port

The **Port** column displays the output index and the name of the output port connected to the For Each Subsystem block.

Tip

You can have any number of ports.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Concatenation Dimension

Specify the dimension along which to stack the results of the For Each Subsystem.

Settings

Default: 1

Minimum: 1

Tips

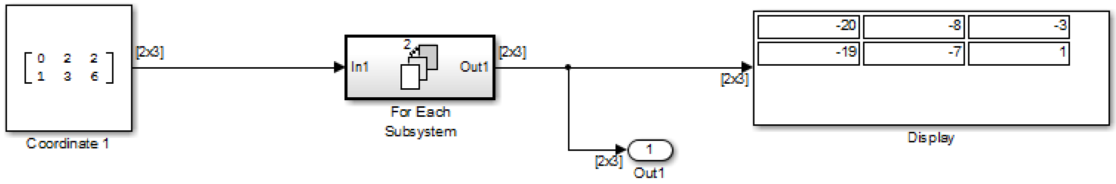
- If you specify the default, the results stack in the d_1 direction. Thus if the block generates column vectors, the concatenation process results in a single column vector.
- If you specify 2, the results stack in the d_2 direction. Thus if the block generates row vectors, the concatenation process results in a single row vector.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

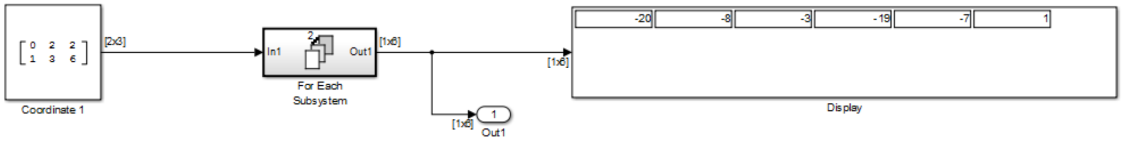
Examples

The following model demonstrates the partitioning of an input signal by a For Each block. Each row of this 2-by-3 input array contains three integers that represent the (*x*, *y*, *z*)-coordinates of a point. The goal is to translate each of these points based on a new origin at (–20, –10, –5) and to display the results.


0 2 2
1 3 6
Coordinate 1
[2x3]
2
In1
Out1
For Each Subsystem
[2x3]
-20
-8
-3
-19
-7
1
[2x3]
Display
[2x3]
1
Out1

By placing the process of summing an input signal and the new origin inside of a For Each Subsystem, you can operate on each set of coordinates by partitioning the input signal into two row vectors. To accomplish such partitioning, use the default settings of 1 for both the partition dimension and the partition width. If you also use the default concatenation dimension of 1, each new set of coordinates stacks in the d_1 direction, making your display a 2-by-3 array.

Alternatively, if you specify a concatenation dimension of 2, then you get a single row vector because each set of results stacks in the d_2 direction.


0 2 2
1 3 6
Coordinate 1
[2x3]
2
In1
Out1
For Each Subsystem
[1x6]
-20
-8
-3
-19
-7
1
[1x6]
Display
[1x6]
1
Out1

This example shows how to partition an input signal. To learn how the For Each block and subsystem handle a model with states, see the For Each Subsystem documentation.

See Also

For Each Subsystem

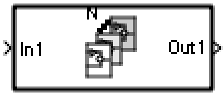
Purpose

Repeatedly perform algorithm on each element or subarray of input signal and concatenate results

Library

Ports & Subsystems

Description



The For Each Subsystem block is useful in modeling scenarios where you need to repeat the same algorithm for individual elements (or subarrays) 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 is that, for certain models, it improves the code reuse in Simulink Coder generated code. Consider a model containing two reusable Atomic Subsystems 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 Subsystems 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 in the model. For each of these cases, the input signal elements have different values.

Limitations

The For Each Subsystem block has these limitations, which you can work around.

Limitation	Workaround
You cannot log any signals in the subsystem.	Pull the signal outside the subsystem using an output port for logging.
You cannot log the states of the blocks in the model in array format. Also, you cannot log the states of the blocks in the subsystem, even if you are using structure format.	Save and restore the simulation state (SimState).
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 the following features:

- You cannot include the following 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
  - 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 the following kinds 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
- Setting the initial state of blocks inside the model if the format of the data is in either of the following formats:
  - Array
  - Structure that includes data for a block inside of the For Each 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, perform the following 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 the following kinds 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.
  - HDL code generation
  - PLC code generation

S-Function Support

The For Each Subsystem block supports both C-MEX S-functions and Level-2 MATLAB S-functions, provided that the S-function supports multiple execution instances using one of the following techniques:

- A C-MEX S-function must declare ssSupportsMultipleExecInstances(S, true) in the mdlSetWorkWidths method.
- A Level-2 MATLAB S-function must declare 'block.SupportsMultipleExecInstances = true' in the Setup method.

If you use the above specifications:

- Do not cache run-time data, such as DWork and Block I/O, using global or persistent variables or within the userdata of the S-function.

- Every S-function execution method from mdlStart up to mdlTerminate is called once for each element processed by the S-function, when it is in a For Each Subsystem. 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.

Data Type Support

The For Each Subsystem block accepts real or complex signals of the following data types:

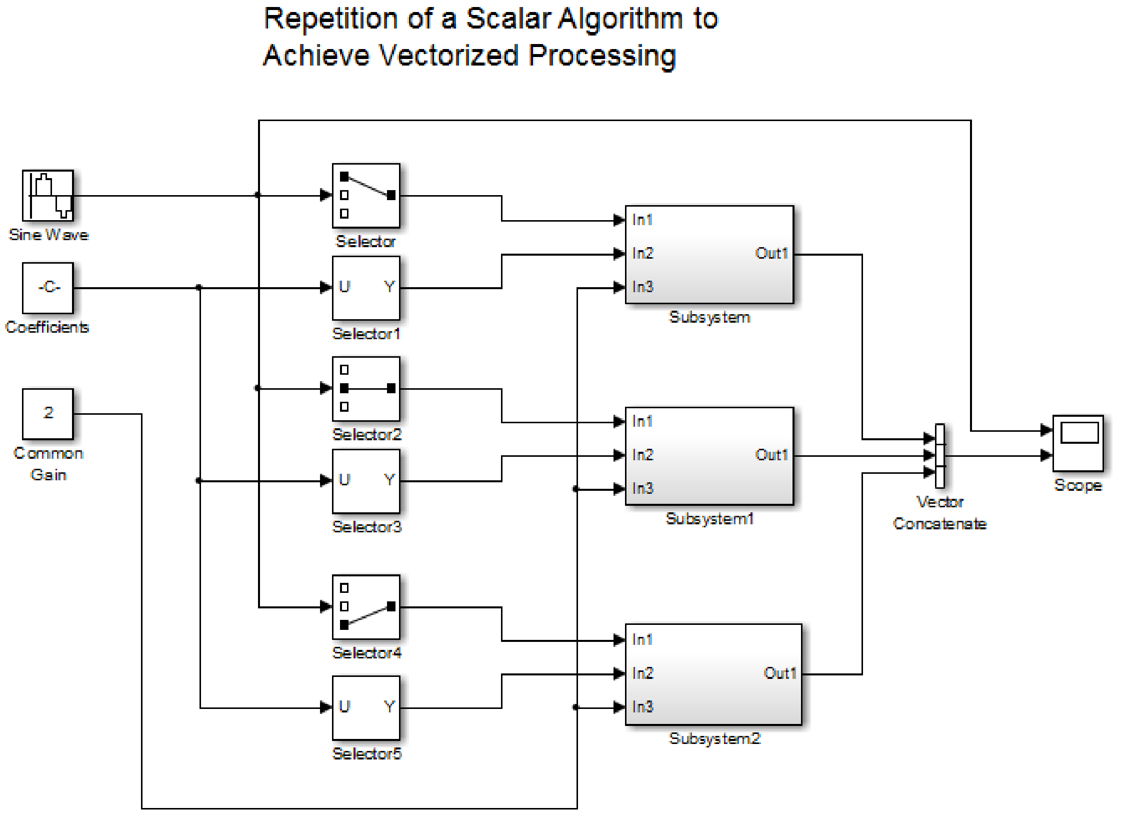
- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

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

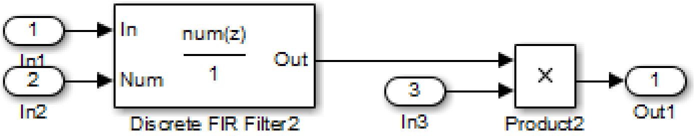
Examples of Working with For Each Subsystems

Example of Simplifying a Model

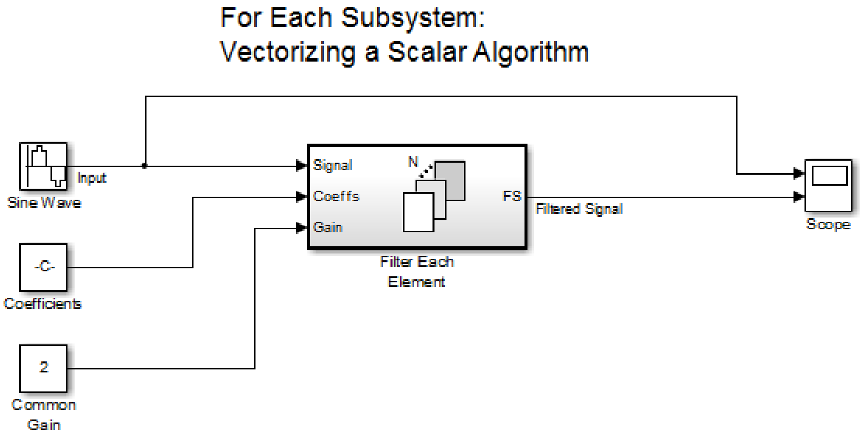
The following model uses an FIR filter to independently process each element of the input sine wave signal, in an identical manner. The model uses Selector blocks and three identical subsystems: Subsystem, Subsystem1, and Subsystem2. A Vector Concatenate block concatenates the resulting vectors. This repetitive process is graphically complex and difficult to maintain. Also, adding another element to the signal requires a significant rework of the model.


Repetition of a Scalar Algorithm to
Achieve Vectorized Processing
Sine Wave
-C-
Coefficients
2
Common Gain
Selector
U Y
Selector1
Selector2
Selector3
Selector4
Selector5
In1
In2
In3
Out1
Subsystem
Subsystem1
Subsystem2
Vector Concatenate
Scope

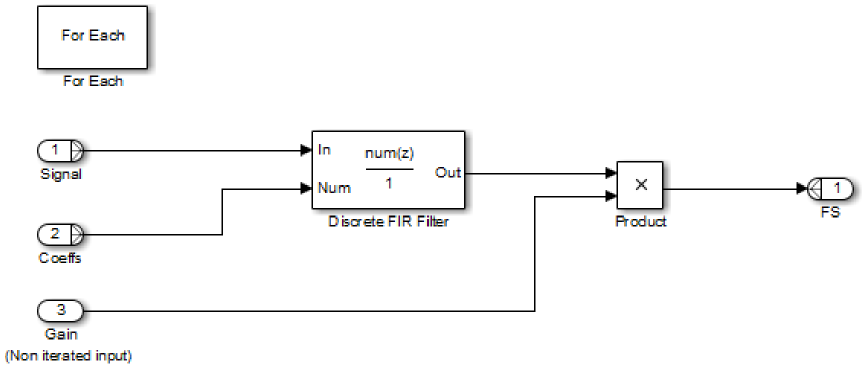
Each subsystem contains an identical FIR filter that accepts a different set of input coefficients. For example, Subsystem2 contains the following blocks:


1
In1
2
In2
In
Num
num(z)
1
Out
Discrete FIR Filter2
3
In3
×
Product2
1
Out1

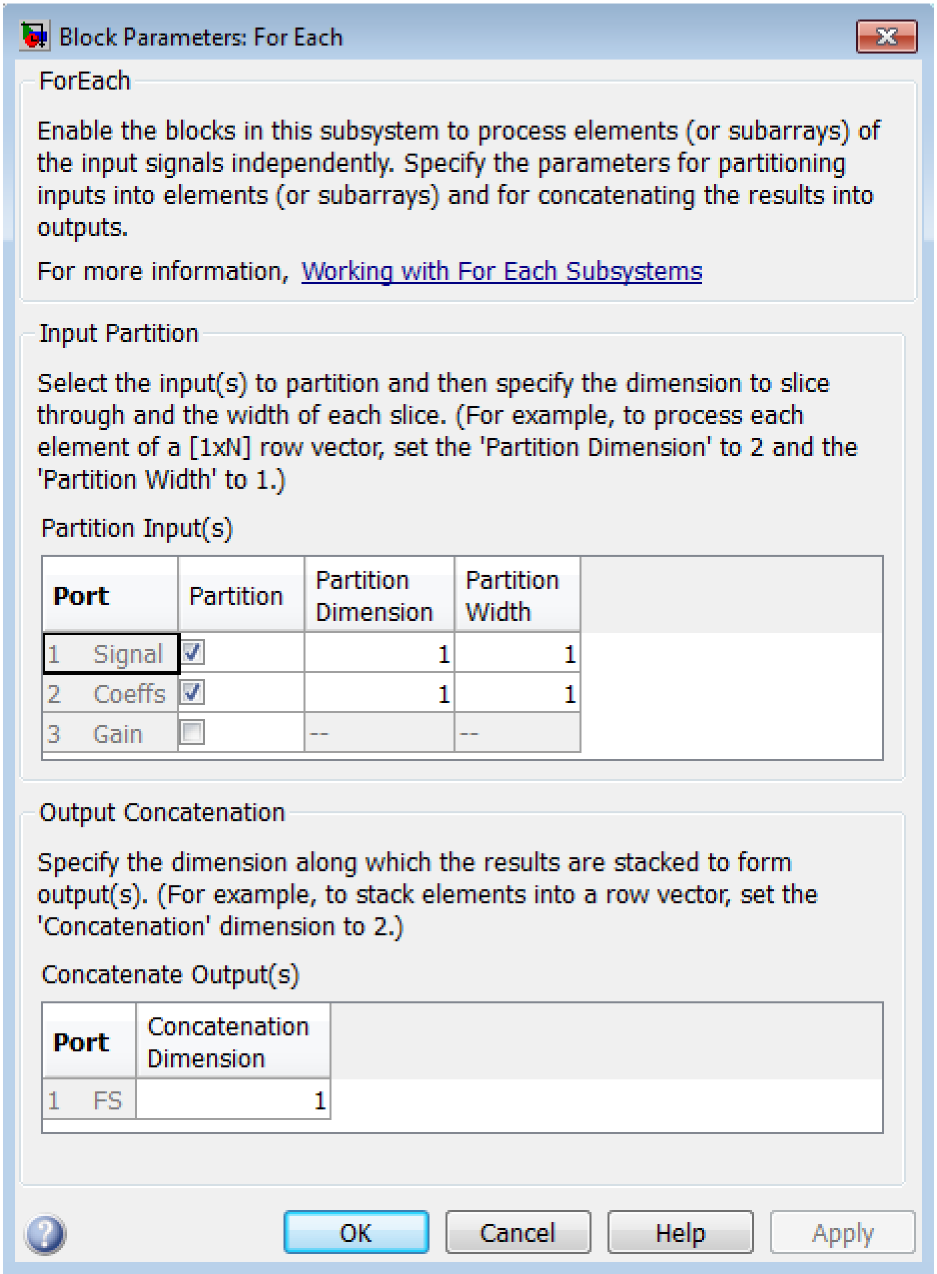
You can simplify this model by replacing the repetitive operations with a single For Each Subsystem block.


For Each Subsystem:
Vectorizing a Scalar Algorithm
Sine Wave
Input
Coefficients
-C-
2
Common Gain
Signal
Coeffs
Gain
N
FS
Filter Each Element
Filtered Signal
Scope

This subsystem contains a For Each block and a model representing the identical algorithm of the three subsystems that it replaces. The For Each block specifies how to partition the input signal into individual elements and how to concatenate the processed signals to form output signals. Also, every block that has state maintains a separate set of states for each input element processed during a given execution step.


For Each
For Each
1
Signal
2
Coeffs
3
Gain
(Non iterated input)
In
Num
num(z)
1
Out
Discrete FIR Filter
×
Product
1
FS

For this example, both the input signal and the input coefficients are selected for partitioning. The partition dimension and the partition width are both set to 1 for both inputs.



Block Parameters: For Each

ForEach

Enable the blocks in this subsystem to process elements (or subarrays) of the input signals independently. Specify the parameters for partitioning inputs into elements (or subarrays) and for concatenating the results into outputs.

For more information, Working with For Each Subsystems

Input Partition

Select the input(s) to partition and then specify the dimension to slice through and the width of each slice. (For example, to process each element of a [1xN] row vector, set the 'Partition Dimension' to 2 and the 'Partition Width' to 1.)

Partition Input(s)

Port	Partition	Partition Dimension	Partition Width
1 Signal	☑	1	1
2 Coeffs	☑	1	1
3 Gain	☐	--	--

Output Concatenation

Specify the dimension along which the results are stacked to form output(s). (For example, to stack elements into a row vector, set the 'Concatenation' dimension to 2.)

Concatenate Output(s)

Port	Concatenation Dimension
1 FS	1

Therefore, the For Each block slices the input signals into horizontal partitions of width 1. The input signal of coefficients transforms from a single array,

$$\begin{array}{l} \text{Partition} \\ \text{dimension} \\ \text{set to 1} \end{array} \updownarrow \left[\begin{array}{ccccc} 0.0284 & 0.2370 & 0.4692 & 0.2370 & 0.0284 \\ \hline -0.0651 & 0.0000 & 0.8698 & 0.0000 & -0.0651 \\ \hline 0.0284 & -0.2370 & 0.4692 & -0.2370 & 0.0284 \end{array}\right]$$

into three signals,

$$\begin{bmatrix} 0.0284 & 0.2370 & 0.4692 & 0.2370 & 0.0284 \end{bmatrix}$$

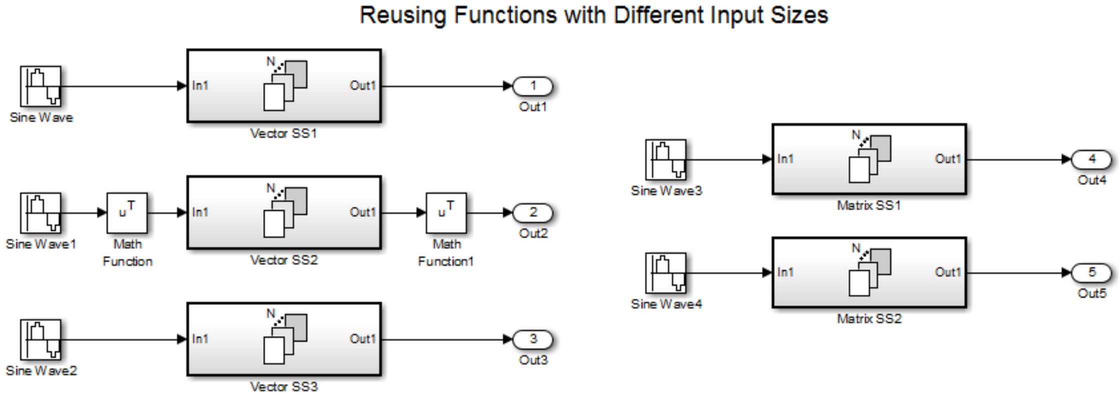
$$\begin{bmatrix} -0.0651 & 0.0000 & 0.8698 & 0.0000 & -0.0651 \end{bmatrix}$$

$$\begin{bmatrix} 0.0284 & -0.2370 & 0.4692 & -0.2370 & 0.0284 \end{bmatrix}$$

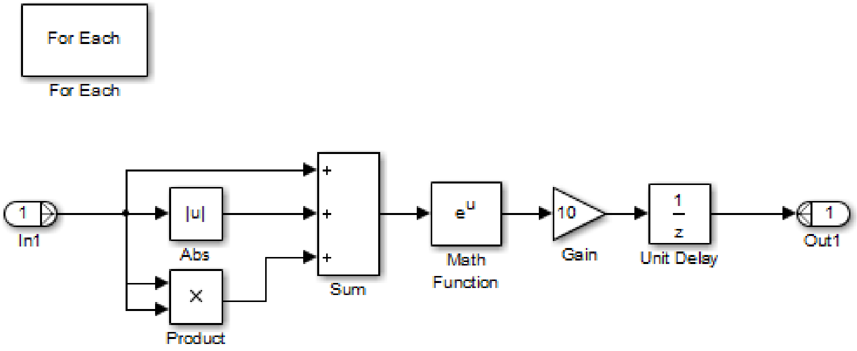
To match the output signal dimension to the input signal dimension, set the concatenation dimension equal to the partition dimension. (The output signal dimension and the input signal dimension do not have to match, but matching them makes it easier to compare the input and output.)

Example of Improved Code Reuse

The purpose of this example is to show how code reuse can be improved when you have two or more identical For Each Subsystems. Consider the following model, rtwdemo_foreachreuse.


Reusing Functions with Different Input Sizes
Sine Wave
In1
Out1
Vector SS1
1
Out1
Sine Wave1
uT
Math Function
Vector SS2
Math Function1
2
Out2
Sine Wave2
Vector SS3
3
Out3
N
Sine Wave3
Matrix SS1
4
Out4
Sine Wave4
Matrix SS2
5
Out5

The intent is for the three subsystems — Vector SS1, Vector SS2, and Vector SS3 — to apply the same processing to each scalar element of the vector signal at their respective input. Because these three subsystems perform the same processing, it is desirable for them to produce a single shared Output (and Update) function for all three subsystems in the code generated for this model. For example, the Vector SS3 subsystem contains the following blocks:


For Each
For Each
1
In1
|u|
Abs
×
Product
+
+
+
Sum
e^u
Math Function
10
Gain
1/z
Unit Delay
1
Out1

To generate a single shared function for the three subsystems, the configuration of the partitioning they perform on their input signals must be identical. For Vector SS1 and Vector SS3, this configuration is straightforward because you can set the partition dimension and width to 1. However, in order for Vector SS2 to also partition its input signal along dimension 1, you must insert a Math Function block to transpose the 1-by-8 row vector into an 8-by-1 column vector. You can subsequently convert the output of the subsystem back to a 1-by-8 row vector using a second Math Function block set to the `transpose` operator.

If you use the **Build** button on the **Code Generation** pane of the Configuration Parameters dialog to generate code, the resulting code uses a single Output function. This function is shared by all three For Each Subsystem instances.

```
/*
 * Output and update for iterator system:
 *    '<Root>/Vector SS1'
 *    '<Root>/Vector SS2'
 *    '<Root>/Vector SS3'
 */
void VectorProcessing(int32_T NumIters, const real_T rtu_In1[],
                      real_T rty_Out1[],
                      rtDW_VectorProcessing *localDW)
```

The functions has an input parameter `NumIters` that indicates the number of independent scalars which each For Each Subsystem is to process. This function is called three times with the parameter `NumIters` set to 10, 8, and 7 respectively.

The remaining two subsystems in this model show how reusable code can also be generated for matrix signals that are processed using the For Each Subsystem. Once again, a **Build** provides code reuse of a single function.

Characteristics

Sample Time	Inherited from the driving block
Scalar Expansion	No

Dimensionalized	Yes
Multidimensionalized	Yes

See Also

For Each

For Iterator

Purpose

Repeatedly execute contents of subsystem at current time step until iteration variable exceeds specified iteration limit

Library

Ports & Subsystems

Description

For N
Iterator

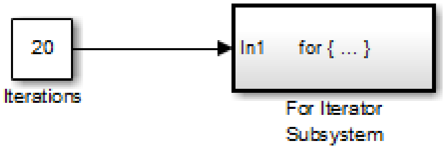
The For Iterator block, when placed in a subsystem, repeatedly executes the contents of the subsystem at the current time step until an iteration variable exceeds a specified iteration limit. You can use this block to implement the block diagram equivalent of a `for` loop in the C programming language.

The output of a For Iterator subsystem can not be a function-call signal. Simulink software will display an error message if the simulation is run or the diagram updated.

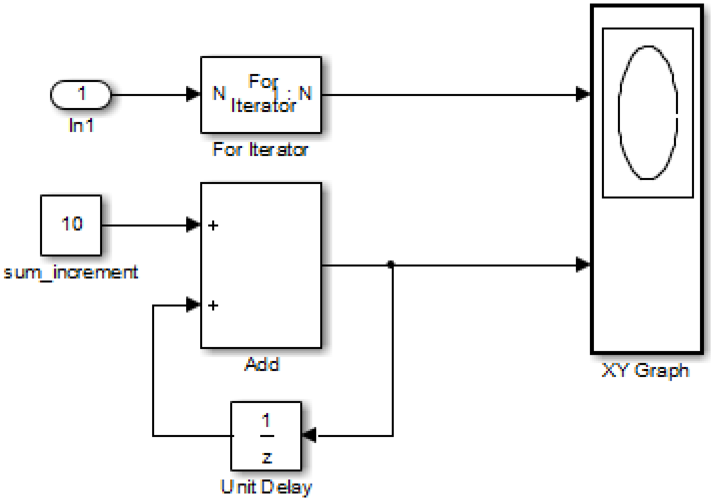
The block's parameter dialog allows you to specify the maximum value of the iteration variable or an external source for the maximum value and an optional external source for the next value of the iteration variable. If you do not specify an external source for the next value of the iteration variable, the next value is determined by incrementing the current value:

$$i_{n+1} = i_n + 1$$

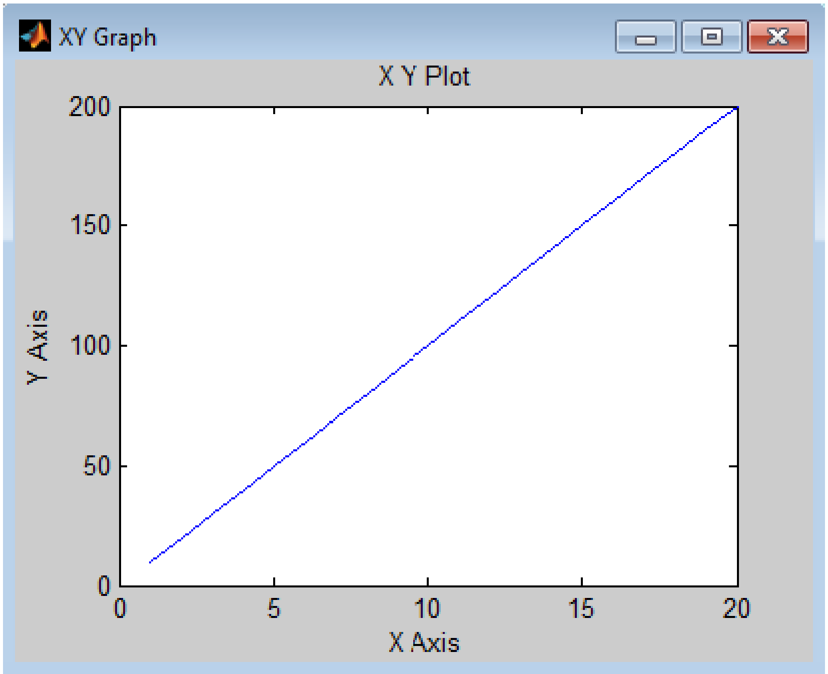
Suppose that you have the following model:



Over 20 iterations, the For Iterator block increments a value by 10 at each time step:


1
In1
N
For Iterator
1 : N
For Iterator
10
sum_increment
+
+
Add
XY Graph
1
z
Unit Delay

The following figure shows the result.



The For Iterator subsystem in this example is equivalent to the following C code.

```
sum = 0;
iterations = 20;
sum_increment = 10;
for (i = 0; i < iterations; i++) {
 sum = sum + sum_increment;
}
```

Note Placing a For Iterator block in a subsystem makes it an atomic subsystem if it is not already an atomic subsystem.

Data Type Support

The following rules apply to the data type of the number of iterations (N) input port:

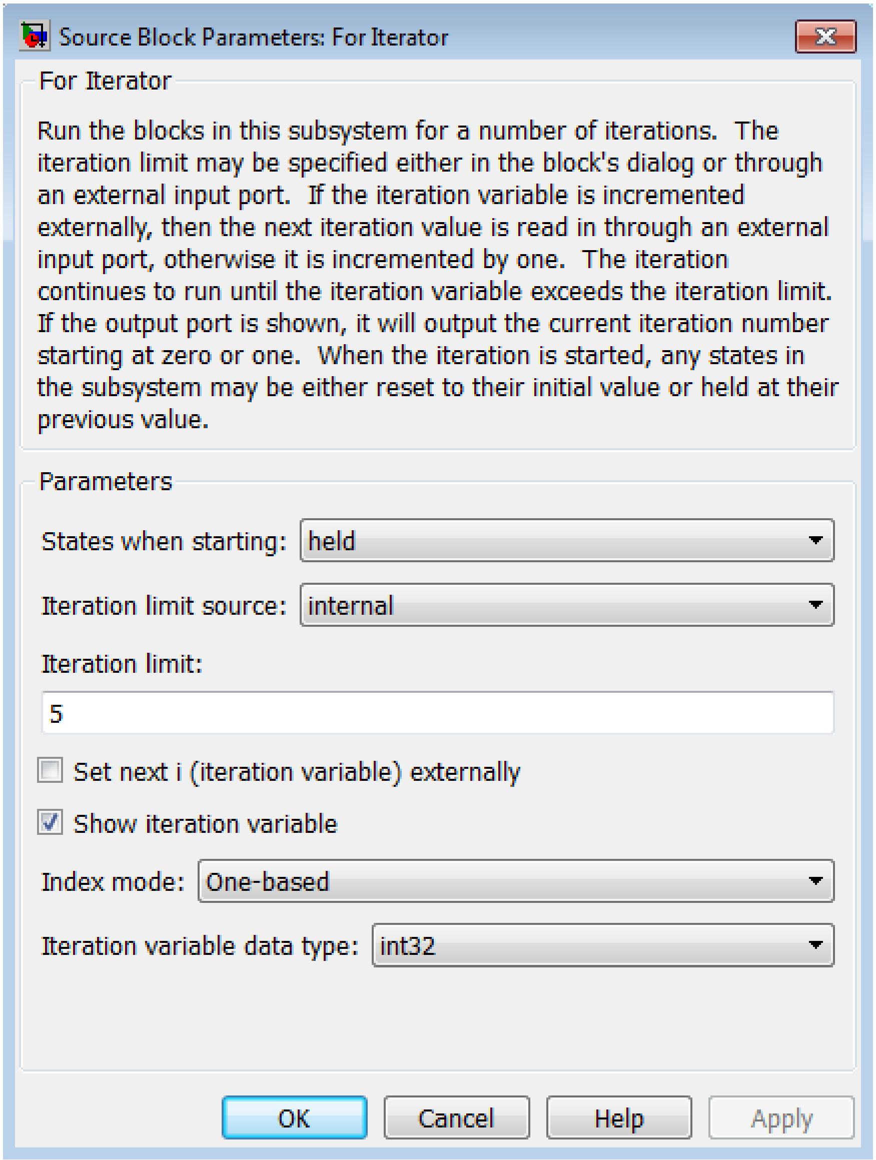
- The input port accepts data of mixed numeric types.
- If the input port value is noninteger, it is first truncated to an integer.
- Internally, the input value is cast to an integer of the type specified for the iteration variable output port.
- If no output port is specified, the input port value is cast to type `int32`.
- If the input port value exceeds the maximum value of the output port's type, it is truncated to that maximum value.

Data output for the iterator value can be selected as `double`, `int32`, `int16`, or `int8` in the block parameters dialog box.

The following rules apply to the iteration variable input port:

- It can appear only if the iteration variable output port is enabled.
- The data type of the iteration variable input port is the same as the data type of the iteration variable output port.

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



For Iterator

Run the blocks in this subsystem for a number of iterations. The iteration limit may be specified either in the block's dialog or through an external input port. If the iteration variable is incremented externally, then the next iteration value is read in through an external input port, otherwise it is incremented by one. The iteration continues to run until the iteration variable exceeds the iteration limit. If the output port is shown, it will output the current iteration number starting at zero or one. When the iteration is started, any states in the subsystem may be either reset to their initial value or held at their previous value.

Parameters

States when starting: held

Iteration limit source: internal

Iteration limit:

5

☐ Set next i (iteration variable) externally

☑ Show iteration variable

Index mode: One-based

Iteration variable data type: int32

Parameters

and Dialog Box

States when starting

Set this field to `reset` if you want the states of the For subsystem to be reinitialized before the first iteration at each time step. Otherwise, set this field to `held` (the default) to make sure that these subsystem states retain their values from the last iteration at the previous time step.

Iteration limit source

If you set this field to `internal`, the value of the **Iteration limit** field determines the number of iterations. If you set this field to `external`, the signal at the For Iterator block's `N` port determines the number of iterations. The iteration limit source must reside outside the For Iterator subsystem.

Iteration limit

Set the number of iterations by specifying a number or a named constant. This field appears only if you selected `internal` for the **Iteration limit source** field. This parameter supports storage classes. You can define the named constant in the base workspace of the Model Explorer as a `Simulink.Parameter` object of the built-in storage class `Define (custom)` type. For more information, see "Apply Custom Storage Classes to Parameters" in the Embedded Coder documentation.

Set next i (iteration variable) externally

This option can be selected only if you select the **Show iteration variable** option. If you select this option, the For Iterator block displays an additional input for connecting an external iteration variable source. The value of the input at the current iteration is used as the value of the iteration variable at the next iteration.

Show iteration variable

If you select this check box, the For Iterator block outputs its iteration value.

Index mode

If you set this field to `Zero-based`, the iteration number starts at zero. If you set this field to `One-based`, the iteration number starts at one.

Iteration variable data type

Set the type for the iteration value output from the iteration number port to `double`, `int32`, `int16`, or `int8`.

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from driving blocks
Scalar Expansion	No
Dimensionalized	No
Zero Crossing	No

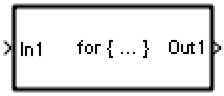
Purpose

Represent subsystem that executes repeatedly during simulation time step

Library

Ports & Subsystems

Description



The For Iterator Subsystem block is a Subsystem block that is preconfigured to serve as a starting point for creating a subsystem that executes repeatedly during a simulation time step.

For more information, see the For Iterator block in the online Simulink block reference and "Control Flow Logic" in the Simulink documentation.

When using simplified initialization mode, you cannot place any block needing elapsed time within an Iterator Subsystem. In simplified initialization mode, Iterator subsystems do not maintain elapsed time, so Simulink will report an error if any such block (such as the Discrete-Time Integrator block) is placed within the subsystem. For more information on simplified initialization modes, see "Underspecified initialization detection".

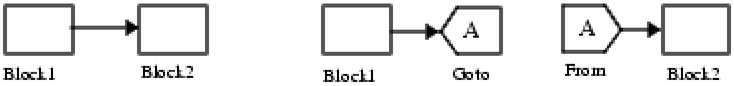
Purpose Accept input from Goto block

Library Signal Routing

Description The From block accepts a signal from a corresponding Goto block, then passes it as output. The data type of the output is the same as that of the input from the Goto block. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them. To associate a Goto block with a From block, enter the Goto block's tag in the **Goto Tag** parameter.

A From block can receive its signal from only one Goto block, although a Goto block can pass its signal to more than one From block.

This figure shows that using a Goto block and a From block is equivalent to connecting the blocks to which those blocks are connected. In the model at the left, Block1 passes a signal to Block2. That model is equivalent to the model at the right, which connects Block1 to the Goto block, passes that signal to the From block, then on to Block2.



The visibility of a Goto block tag determines the From blocks that can receive its signal. For more information, see Goto and Goto Tag Visibility. The block indicates the visibility of the Goto block tag:

- A local tag name is enclosed in brackets ([]).
- A scoped tag name is enclosed in braces ({}).
- A global tag name appears without additional characters.

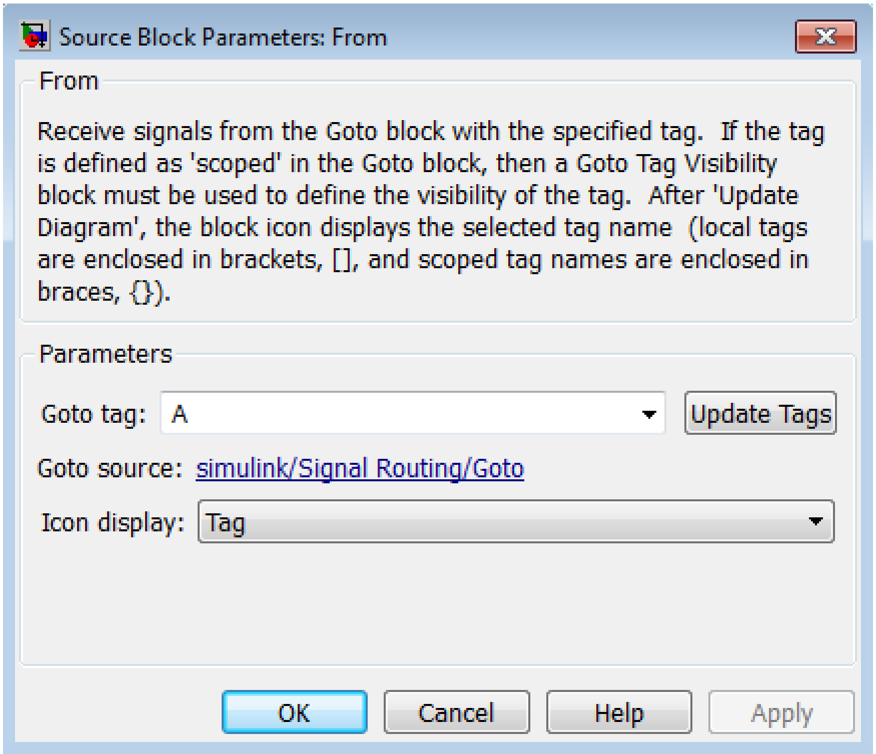
The From block supports signal label propagation.

Data Type Support

The From block outputs real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

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

Parameters and Dialog Box



Receive signals from the Goto block with the specified tag. If the tag is defined as 'scoped' in the Goto block, then a Goto Tag Visibility block must be used to define the visibility of the tag. After 'Update Diagram', the block icon displays the selected tag name (local tags are enclosed in brackets, [], and scoped tag names are enclosed in braces, {}).

Goto tag: A

Goto source: simulink/Signal Routing/Goto

Icon display: Tag

Goto Tag

The tag of the Goto block that forwards its signal to this From block. To change the tag, select a new tag from this control’s drop-down list. The drop-down list displays the Goto tags that the From block can currently see. An item labeled `<More Tags...>` appears at the end of the list the first time you display the list in a Simulink session. Selecting this item causes the block to update the tags list to include the tags of Goto blocks residing in library subsystems referenced by the model containing this From block. Simulink software displays a progress bar while building the list of library tags. Simulink software saves the updated tags list for the duration of the Simulink session or until the next time you select the adjacent **Update Tags** button. You need to update the tags list again in the current session only if the libraries referenced by the model have changed since the last time you updated the list.

Update Tags

Updates the list of tags visible to this From block, including tags residing in libraries referenced by the model containing this From block.

Goto Source

Path of the Goto block connected to this From block. Clicking the path displays and highlights the Goto block.

Icon Display

Specifies the text to display on the From block's icon. The options are the block's tag, the name of the signal that the block represents, or both the tag and the signal name.

Examples

The following models show how to use the From block:

- `sldemo_auto_climatecontrol`
- `sldemo_hardstop`

Characteristics

Sample Time	Inherited from the block driving the Goto block
Dimensionalized	Yes
Multidimensionalized	Yes

Virtual	Yes For more information, see "Virtual Blocks" in the Simulink documentation.
Zero-Crossing Detection	No

See Also

Goto

From File

Purpose

Read data from MAT-file

Library

Sources

Description

The From File block reads data from a MAT-file 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.

untitled.mat

The From File block icon shows the name of the MAT-file that supplies the data.

You can have multiple From File blocks that read from the same MAT-file.

During the current simulation, the From File block cannot read data from a MAT-file that is being written to by a To File block.

Version 7.3 MAT-Files

For a *Version 7.3* MAT-file, the From File block incrementally reads data from the MAT-file during simulation. The **Sample time** parameter specifies the sample time that the From File block uses to read data from a MAT-file. For details, see “Parameters and Dialog Box”.

The time stamps in the file must be monotonically nondecreasing.

For each simulation time hit for which the MAT-file contains no matching time stamp, Simulink software interpolates or extrapolates to obtain the needed data using the method that you select. For details, see “Simulation Time Hits That Have No Corresponding MAT-File Time Stamps.”

When the From File block reads data from a MAT-file, that data must be stored in one of two formats:

- MATLAB `timeseries` object
- Array

Use array format only for vector, double, noncomplex signals. To load a bus signal, use a MATLAB structure that matches the bus hierarchy, where each leaf of the structure is a MATLAB `timeseries` object.

The MATLAB `timeseries` format supports the following simulation and code generation modes:

- Normal
- Accelerator
- Rapid Accelerator
- Rapid simulation target (RSim)
- Model Reference Accelerator

MATLAB `timeseries` format data can have:

- Any dimensionality and complexity
- Any built-in data type, including `Boolean`
- A fixed-point data type with a word length of up to 32 bits
- An enumerated data type

The array format for stored data is a matrix containing two or more rows. The matrix in the MAT-file must have the following form:

$$\begin{bmatrix} t_1 & t_2 & \dots & t_{final} \\ u1_1 & u1_2 & \dots & u1_{final} \\ \dots & & & \\ un_1 & un_2 & \dots & un_{final} \end{bmatrix}$$

The first element of each column contains a time stamp. The remainder of the column contains data for the corresponding output values. Each element must be a double and cannot include a NaN.

For data stored using the array format, the width of the From File output depends on the number of rows in the matrix. Given a matrix containing m rows, the block outputs a vector of length m–1.

Version 7.0 and Earlier Version MAT-Files

For *Version 7.0* or an earlier version of a MAT-file, the From File block reads only array-format data. The complete, uncompressed data from a MAT-file loads into memory at the start of the simulation.

If you have a Version 7.0 or earlier version MAT-file that you want to use with the From File block, consider converting the file to a Version 7.3 MAT-file. Use a Version 7.3 MAT-file if you want the From File block to incrementally read the data during simulation or you want to use MATLAB `timeseries` data. For example, to convert a Version 7.0 file named `my_data_file.mat` that contains the variable `var`, at the MATLAB command line, enter:

```
load('my_data_file.mat')
save('my_data_file.mat', 'var', '-v7.3')
```

Simulation Time Hits That Have No Corresponding MAT-File Time Stamps

If the simulation time hit does not have a corresponding MAT-file time stamp, then the From File block output depends on:

- The location of the simulation time hit relative to the time stamps in the MAT-file
- The interpolation or extrapolation methods that you select
- The data type of the MAT-file data

For details about interpolation and extrapolation options, see the descriptions of the following parameters in “Parameters and Dialog Box”:

- **Data extrapolation before first data point**
- **Data interpolation within time range**

- **Data extrapolation after last data point**

Sometimes the MAT-file includes duplicate time stamps (two or more data values that have the same time stamp). In such cases, the From File block action depends on the location of the simulation time hit, relative to the duplicate time stamps in the MAT-file.

For example, suppose the MAT-file contains the following data, with three data points having 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 following table describes the From File block output.

Simulation Time, Relative to Duplicate Time Stamp Values in MAT-File	From File 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, that 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.)

Using Data Saved by a To File Block

The From File block reads data written by a To File block without any modifications to the data or other special provisions.

Using Bus Data

The From File block supports loading nonvirtual bus signals.

The data must be in a MATLAB structure that matches the bus hierarchy. Each leaf of the structure must be a MATLAB `timeseries` object.

The data can underspecify the bus signal, but cannot overspecify the bus signal. The structure cannot have any elements that do not have corresponding signals in the bus.

The structure does not require data for every element in the bus hierarchy, but the structure must have data for at least one of the signals in the bus. For signals that do not specify data, the From File block outputs the ground values. If any data read by the From File block has variable dimensions, then you must specify data for all complex bus signals.

Code Generation Requirements

For a From File block, simulating in Accelerator, Rapid Accelerator, or model reference Accelerator mode or building an RSim target behaves the same way, and has the same requirements, as simulating in Normal mode.

For a From File block, generating code that involves building ERT or GRT targets, or using SIL or PIL simulation modes, requires that:

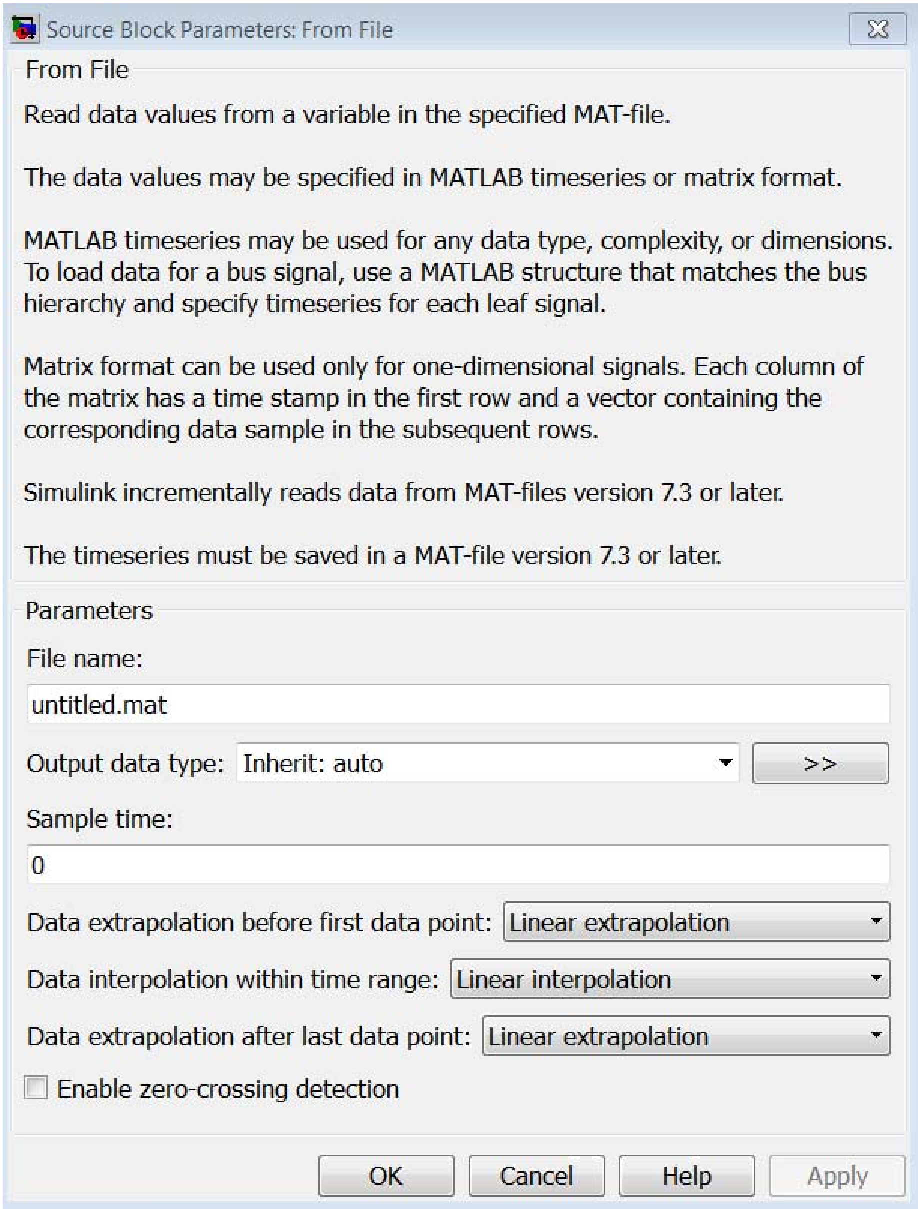
- The MAT-file contains a non-empty finite real matrix with at least two rows.
  - The matrix must have a data type of `double`.
  - The matrix does not have any NaN elements.
- The From File block parameters dialog box has the following settings:
  - Set the **Data extrapolation before first data point** and **Data extrapolation after last data point** parameters to `Linear extrapolation`.

- Set tthe **Data interpolation within time range** parameter to `Linear interpolation`
- Clear the **Enable zero-crossing detection** parameter.

Data Type Support

The From File block can read real or complex signal data of any data type that Simulink supports, except that fixed-point data cannot have a word length that exceeds 32 bits.

The From File block supports reading nonvirtual bus signals.



Source Block Parameters: From File

From File

Read data values from a variable in the specified MAT-file.

The data values may be specified in MATLAB timeseries or matrix format.

MATLAB timeseries may be used for any data type, complexity, or dimensions. To load data for a bus signal, use a MATLAB structure that matches the bus hierarchy and specify timeseries for each leaf signal.

Matrix format can be used only for one-dimensional signals. Each column of the matrix has a time stamp in the first row and a vector containing the corresponding data sample in the subsequent rows.

Simulink incrementally reads data from MAT-files version 7.3 or later.

The timeseries must be saved in a MAT-file version 7.3 or later.

Parameters

File name: untitled.mat

Output data type: Inherit: auto

Sample time: 0

Data extrapolation before first data point: Linear extrapolation

Data interpolation within time range: Linear interpolation

Data extrapolation after last data point: Linear extrapolation

☐ Enable zero-crossing detection

Parameters and Dialog Box

File name

The path or file name of the MAT-file that contains the data used as input. On UNIX® systems, the path name can start with a tilde

(~) character signifying your home folder. The default file name is `untitled.mat`. If you specify a file name without path information, Simulink reads the file in the MATLAB working folder. (To determine the working folder, type `pwd` at the MATLAB command line.) If Simulink cannot find the specified file name in the working folder, it displays an error message.

Output data type

The data type for the data that the From File block loads. For non-bus 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 data type that you specified. For more information, see "Specify Block Output Data Types".

If you set **Output data type** to be a bus object, the bus object must be available when you compile the model. For each signal in a bus, the From File block verifies that the data type, dimension, and complexity of the data matches the data type, dimension, and complexity that the bus object defines for the signal.

- `Inherit: auto` — Default
- `double`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `fixdt(1,16,0)` — Data type is fixed-point (1,16,0).
- `fixdt(1,16,2^0,0)` — Data type is fixed-point (1,16,2^0,0).

- `Enum: <class_name>` — Data type is enumerated.
- `Bus: <bus_object>` — Data type is a bus object. For details, see the "Using Bus Data" section.
- `<data type expression>` — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see "Specify Block Output Data Types".

- `Inherit` — Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right. (Default)
- `Built in` — Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:
  - `double` — Default
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`
  - `uint32`
  - `boolean`
  - `Fixed point` — Fixed-point data types

  - `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
  - `Bus` — Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see "Manage Bus Objects with the Bus Editor".
  - `Expression` — Expression that evaluates to a data type. Selecting `Expression` enables a second menu/text box to the right, where you enter the expression. Do not specify a bus object as the expression.

Sample time

The sample period and offset of the data read from the file.

The From File block reads data from a MAT-file using a sample time that:

- 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 is `0`, which specifies a continuous sample time, The MAT-file is read at the base (fastest) rate of the model. For details, see "Specify Sample Time".

Data extrapolation before first data point

Extrapolation method that Simulink uses for a simulation time hit that is before the first time stamp in the 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`, upcasts the data to `double`, performs linear extrapolation (as described above for `double` data), and then 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, so 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`

Method	Description
	- `Boolean` — uses `false` - Enumerated data types — uses default value

Data interpolation within time range

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the MAT-file. Choose one of the following interpolation methods.

Method	Description
`Linear interpolation`	(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 a built-in data type other than `double` or `Boolean`, upcasts the data to `double`, performs linear interpolation (as described above for `double` data), and then downcasts the interpolated value to the original data type You cannot use the `Linear interpolation` option with enumerated (`enum`) data. All signals in a bus use the same interpolation setting, so if any signal in a bus uses `enum` data, then you cannot use the `Linear interpolation` option.
`Zero order hold`	Uses the data from the first of the two samples

Data extrapolation after last data point

The extrapolation method that Simulink uses for a simulation time hit that is after the last time stamp in the 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 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), and then downcasts the extrapolated 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, so if any signal in a bus uses `enum` data, then you cannot use the `Linear extrapolation` option.
`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:

Method	Description
	• 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

Enable zero-crossing detection

Select to enable zero-crossing detection.

For details, see “Zero-Crossing Detection”, as well as the discussion below.

Simulink uses a technique known as zero-crossing detection to accurately locate a discontinuity, without resorting to excessively small time steps. This section uses “zero-crossing” to represent discontinuities.

For the From File block, zero-crossing detection can only occur at sample points in the file. To determine whether zero crossing occurs, the From File block examines only the time stamps within the sample points.

If the input array contains duplicate time stamps (more than one entry for the same time stamp), Simulink detects a zero crossing at that time stamp. For example, suppose 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**

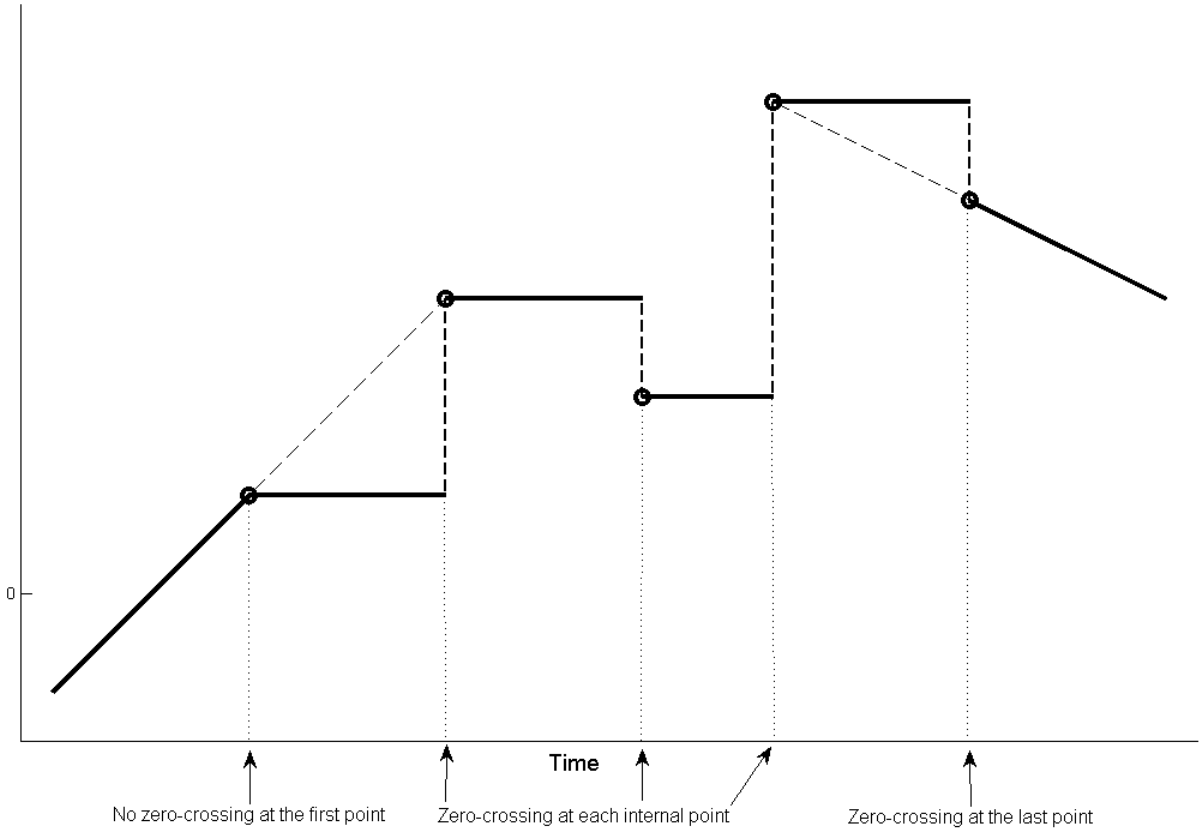
- **Data interpolation within time range**
- **Data extrapolation after last data point**

The From File 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, and for the last time stamp.

Time Stamp	When Zero-Crossing Detection Occurs
First	**Data extrapolation before first data point** is set to `Ground value`.
Between first and last	**Data interpolation within time range** is set to `Zero-order hold`.
Last	One or both of these settings occur: • **Data extrapolation after last data point** is set to `Ground value`. • **Data interpolation within time range** is set to `Zero-order hold`.

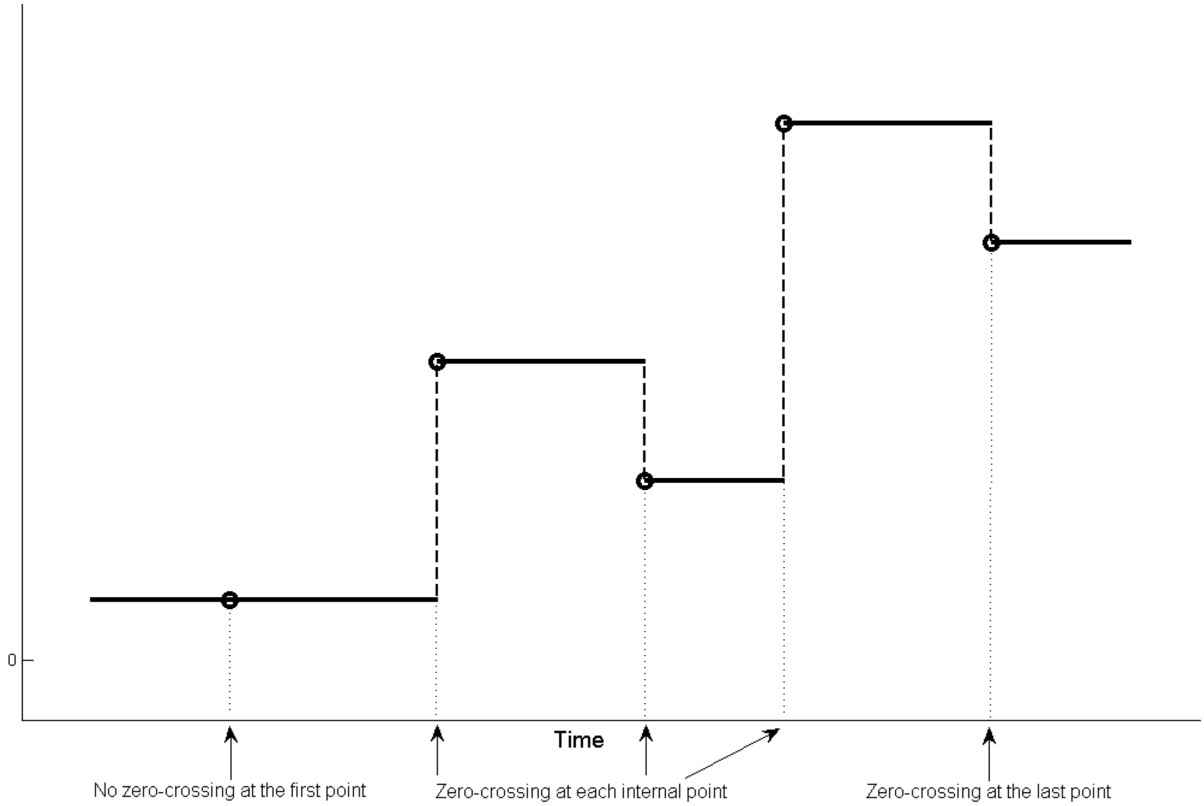
The following figure illustrates zero-crossing detection for data accessed by a From File block that has the following settings:

- **Data extrapolation before first data point** — `Linear extrapolation`
- **Data interpolation within time range** (for internal points) — `Zero order hold`
- **Data extrapolation after last data point** — `Linear extrapolation`



The following 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`



Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Zero Crossing	Yes, if enabled

See Also “Import Data”, From Workspace, To File, To Workspace

From Workspace

Purpose

Read data from workspace

Library

Sources

Description

The From Workspace block reads data from a workspace and outputs the data as a signal.

The From Workspace icon displays the expression specified in the **Data** parameter. For details about how Simulink software evaluates this expression, see “Symbol Resolution”.

Specifying the Workspace Data

In the **Data** parameter of the block, enter a MATLAB expression that specifies the workspace data. The expression must evaluate to one of the following:

- A MATLAB `timeseries` object
- A structure of MATLAB `timeseries` objects
- An array or structure containing an array of simulation times and corresponding signal values

The format of a MATLAB `timeseries` object, matrix, or structure loaded (imported) by a From Workspace block is the same as that used to load root-level input port data from the workspace.

- To load any non-bus data from the workspace, use a MATLAB `timeseries` object or a structure with time. For guidelines on choosing time vectors for discrete systems, see “Techniques for Importing Signal Data”.
- To load bus data, use a structure of MATLAB `timeseries` objects. For details, see “Import Structures of timeseries Objects for Buses”.

Using Data Saved by a To File Block

You can use the From Workspace block to load MATLAB `timeseries` object data that was saved by a To File block, without making any changes to the data.

To use the From Workspace block to load `Array` format data, before the From Workspace block loads the data that a To File block saved, transpose the array data. The data saved by the To File block contains consecutive time stamps at the beginnings of columns, followed by the corresponding data. The transposed data contains consecutive time stamps at the beginning of rows, followed by the corresponding data. To provide the required format, use MATLAB `load` and `transpose` commands with the MAT-file (see “Reshaping a Matrix”). To avoid having to transpose the data again later, resave the transposed data for future use.

Using Data Saved by a To Workspace Block

To use the From Workspace block to load data exported by a To Workspace block in a previous simulation for use in a later simulation, save the To Workspace block data in either `Timeseries` or `Structure with Time` format. For details, see “Techniques for Importing Signal Data”.

Loading Variable-Size Signals

You can use a To Workspace block (with `Structure` or `Structure With Time` format) or a root Outport block to log variable-size signals. You can then use the To Workspace variable with the From Workspace block.

Alternatively, you can create a MATLAB structure to contain variable-size signal data. For each `values` field in the structure, include a `valueDimensions` field that specifies the run-time dimensions for the signal. For details, see Simulink Models Using Variable-Size Signals.

Interpolating Missing Data Values

If you select the **Interpolate data** option, the block uses linear Lagrangian interpolation to compute data values for time steps that occur between time steps for which the workspace supplies data.

For variable-size signals, clear **Interpolate data**.

Specifying Output After Final Data

Combine the settings of the **Form output after final data value by** and **Interpolate data** parameters to determine the block output after the last time step for which workspace data is available. For details, see the **Form output after final data value by** parameter.

Detecting Zero Crossings

The **Enable zero-crossing detection** parameter applies only if the sample time is continuous (`0`).

If you select the **Enable zero-crossing detection** parameter, and if the input array contains more than one entry for the same time step, Simulink detects a zero crossing at that time step.

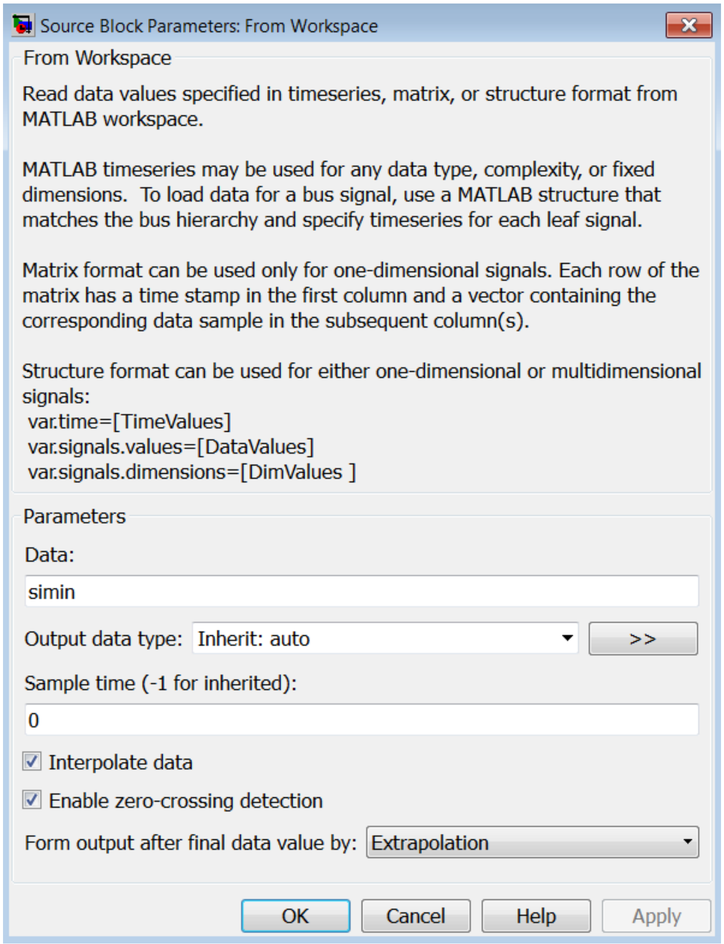
For bus signals, Simulink detects zero crossings across all leaf bus elements.

Data Type Support

The From Workspace block accepts data from the workspace and outputs real or complex signals of any type supported by Simulink, including fixed-point and enumerated data types.

The From Workspace block also accepts a bus object as a data type. To load bus data, use a structure of MATLAB `timeseries` objects. For details, see "Import Structures of timeseries Objects for Buses".

Real signals of type `double` can be in any 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`.



Source Block Parameters: From Workspace

From Workspace

Read data values specified in timeseries, matrix, or structure format from MATLAB workspace.

MATLAB timeseries may be used for any data type, complexity, or fixed dimensions. To load data for a bus signal, use a MATLAB structure that matches the bus hierarchy and specify timeseries for each leaf signal.

Matrix format can be used only for one-dimensional signals. Each row of the matrix has a time stamp in the first column and a vector containing the corresponding data sample in the subsequent column(s).

Structure format can be used for either one-dimensional or multidimensional signals:

```
 var.time=[TimeValues]
 var.signals.values=[DataValues]
 var.signals.dimensions=[DimValues ]
```

Parameters

Data: simin

Output data type: Inherit: auto

Sample time (-1 for inherited): 0

☑ Interpolate data

☑ Enable zero-crossing detection

Form output after final data value by: Extrapolation

Parameters and Dialog Box

Data

A MATLAB expression that evaluates to one of the following:

- A MATLAB timeseries object
- A structure of MATLAB timeseries objects

To load bus data, use a structure of MATLAB timeseries objects. For details, see “Import Structures of timeseries Objects for Buses”.

- A two-dimensional matrix:
  - The first element of each matrix row is a time stamp.
  - The rest of each row is a scalar or vector of signal values.

  The leftmost element of each row is the time stamp of the value(s) in the rest of the row.
- A structure, with or without time, which contains:
  - An array or matrix named `signals.values`, which contains scalars or vectors of signal values.
  - An array named `signals.dimensions`, which contains the dimensions of the signals.
  - Optionally, a vector named `time`, which contains time stamps.

  The `n`’th `time` element is the time stamp of the `n`’th `signals.values` element. For details, see “Techniques for Importing Signal Data”.

For example, suppose that the workspace contains a column vector of times named `T` and a column vector of corresponding signal values named `U`. Entering the expression `[T U]` for this parameter yields the required input array. If the required array or structure already exists in the workspace, enter the name of the structure or matrix in this field.

Output data type

The required data type for the data for the workspace variable that the From Workspace block loads. For non-bus types, you can use `Inherit: auto` to skip any data type verification. For more information, see “Specify Block Output Data Types”.

To load bus data, use a structure of MATLAB timeseries objects. For details, see “Import Structures of timeseries Objects for Buses”.

- `Inherit:  auto` — Default.

- `double`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `fixdt(1,16,0)` — Data type is fixed-point (1,16,0).
- `fixdt(1,16,2^0,0)` — Data type is fixed-point (1,16,2^0,0).
- `Enum: <class_name>` — Data type is enumerated, for example, `Enum: Basic Colors`.
- `Bus: <bus_object>` — Data type is a bus object.
- `<data type expression>` — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see "Specify Block Output Data Types".

- `Inherit` — Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right. (Default)
- `Built in` — Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` — Default
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `Fixed point` — Fixed-point data types
- `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
- `Bus` — Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** (to the right of the **Bus object** field) to open the Simulink Bus Editor. For details, see "Manage Bus Objects with the Bus Editor".
- `Expression` — Expression that evaluates to a data type. Selecting `Expression` enables a second menu/text box to the right, where you enter the expression. Do not specify a bus object as the expression.

Sample time

Sample rate of data from the workspace. For details, see "Specify Sample Time".

Interpolate data

Select this option to have the block linearly interpolate at time steps for which no corresponding workspace data exists. Otherwise, the current output equals the output at the most recent time for which data exists.

If you select this option, the block linearly interpolates a missing data point from the two known data points between which it falls. For example, suppose the block reads the following time series from the workspace:

```
time:      1   2   3    4
signal:   253 254  ?   256
```

The block would output:

```
time:      1   2   3    4
signal:   253 254 255  256
```

If you clear the **Interpolate data** option, the block uses the most recent data value supplied from the workspace to provide any missing data values. For example, the result of the incomplete set of signal values shown above would be:

```
time:      1   2   3    4
signal:   253 254 254  256
```

Integer data — If the input data type is an integer type and an interpolated data point exceeds the data type's range, the block sets the missing data point to be the maximum value that the data type can represent. Similarly, if the interpolated or extrapolated value is less than the minimum value that the data type can represent, the block sets the missing data point to the minimum value that the data type can represent. For example, suppose that the data type is `uint8` and the value interpolated for a missing data point is 256.

```
time:      1   2   3   4
signal:   253 254 255  ?
```

In this case, the block sets the value of the missing point to 255, the largest value that can be represented by the `uint8` data type:

```
time:      1   2   3    4
signal:   253 254 255  255
```

Boolean data — If the input data is boolean, the block uses the value of the nearest workspace data point as the value of missing data point when determining missing data points that fall between the first and last known points. For example, suppose the workspace supplies values at time steps 1 and 4 but not at 2 and 3:

```
time:     1 2 3 4
signal:   1 ? ? 0
```

In this case, the block would use the value of data point 1 as the value of data point 2 and the value of data point 4 as the value of data point 3:

```
time:     1 2 3 4
signal:   1 1 0 0
```

The block uses the value of the last known data point as the value of time steps that occur after the last known data point.

Combine the settings of the **Interpolate data** and **Form output after final data value by** parameters to determine the block output after the last time step for which workspace data is available. For details, see the **Form output after final data value by** parameter.

Enable zero-crossing detection

If you select the **Enable zero-crossing detection** parameter, then when the input array contains more than one entry for the same time step, Simulink detects a zero crossing at that time step. For example, suppose 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”.

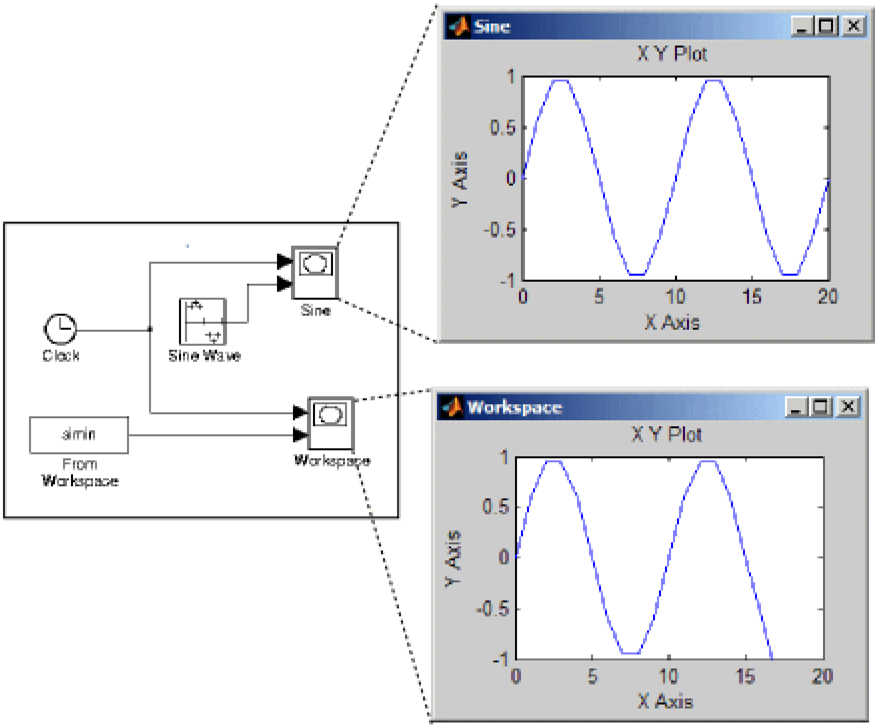
For bus signals, Simulink detects zero crossings across all leaf bus elements.

Form output after final data value by

Combine the settings of the **Form output after final data value by** and **Interpolate data** parameters to determine the block output after the last time step for which workspace data is available. The following table describes 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, if **Form output after final data value by** is `Extrapolation` and **Interpolate data** is selected, the block uses the last two known data points to extrapolate data points that occur after the last known point. Consider the following model.


Sine
X Y Plot
1
0.5
0
-0.5
-1
0
5
10
15
20
Y Axis
X Axis
Clock
Sine Wave
Sine
simin
From Workspace
Workspace
Workspace
X Y Plot
1
0.5
0
-0.5
-1
0
5
10
15
20
Y Axis
X Axis

In this model, the From Workspace block reads data from the workspace consisting of the output of the Simulink Sine block sampled at one-second intervals. The workspace contains the first 16 samples of the output. The top and bottom X-Y plots display the output of the Sine Wave and From Workspace blocks, respectively, from 0 to 20 seconds. The straight line in the output of the From Workspace block reflects the block's linear extrapolation of missing data points at the end of the simulation.

Examples

The From Workspace block allows you to read 1-D and 2-D signals into Simulink.

Reading 1-D Signals in Array Format

Create two signals `x` and `y` with a time vector `t`, and then import the values into Simulink with an array.

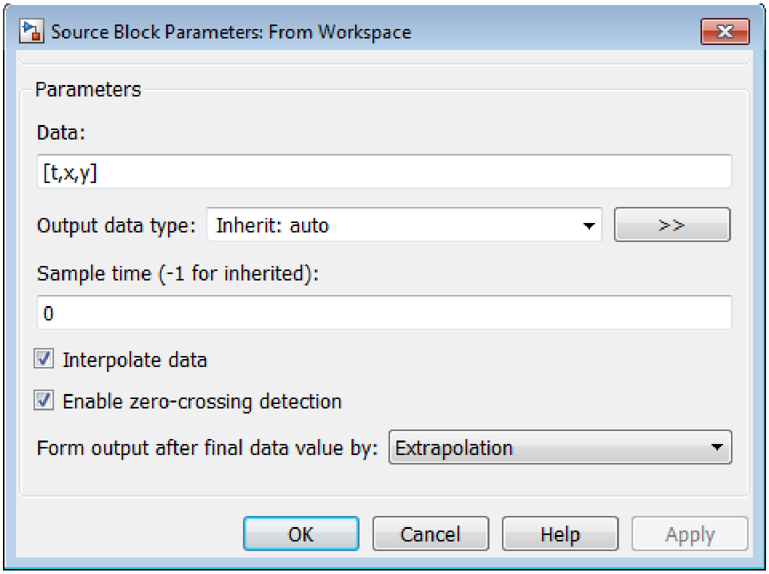
1. In the MATLAB Command Window, enter

   ```
   t = [0:0.2:10]';
   x = sin(t);
   y = 10*sin(t);
   ```

   The time vector must be a column vector.

2. Add a **From Workspace** block to your model.

3. Double-click the block to open the block parameters dialog. In the **Data** field, enter the array [t,x,y],



Source Block Parameters: From Workspace
Parameters

Data: [t,x,y]

Output data type: Inherit: auto

>>

Sample time (-1 for inherited): 0

☑ Interpolate data

☑ Enable zero-crossing detection

Form output after final data value by: Extrapolation

OK
Cancel
Help
Apply

Reading 1-D Signals in Structure Format

Create two signals `x` and `y` with a time vector `t`, and then import the values into Simulink with a structure.

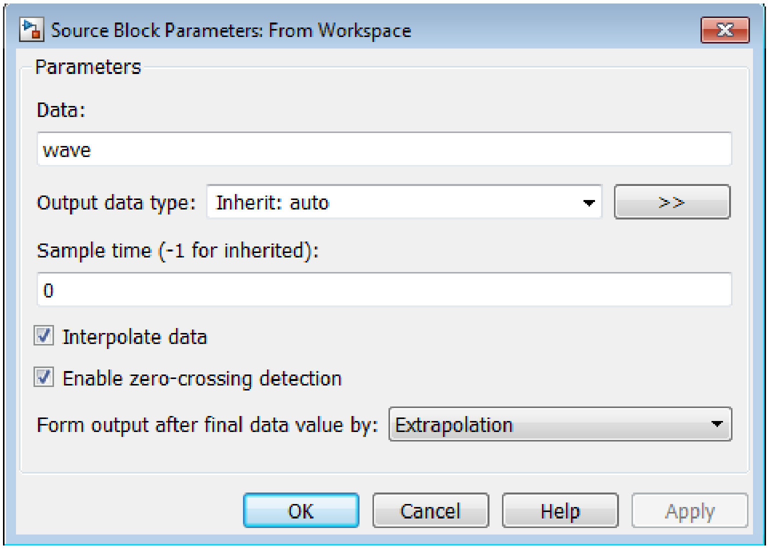
1. In the MATLAB Command Window, enter

   ```
   t = [0:0.2:10]';
   x = sin(t);
   y = 10*sin(t);
   wave.time = t;
   wave.signals.values = [x,y];
   wave.signals.dimensions =2;
   ```

   The time vector must be a column vector. The `signals.dimensions` field for a 1-D signal is a scalar corresponding to the number of signals or number of columns in the `signals.values` field. Each column in the `signals.values` field corresponds to a signal

2. Add a **From Workspace** block to your model.

3. Double-click the block to open the block parameters dialog. In the **Data** field, enter the structure name.



Data: wave

Output data type: Inherit: auto

Sample time (-1 for inherited): 0

☑ Interpolate data

☑ Enable zero-crossing detection

Form output after final data value by: Extrapolation

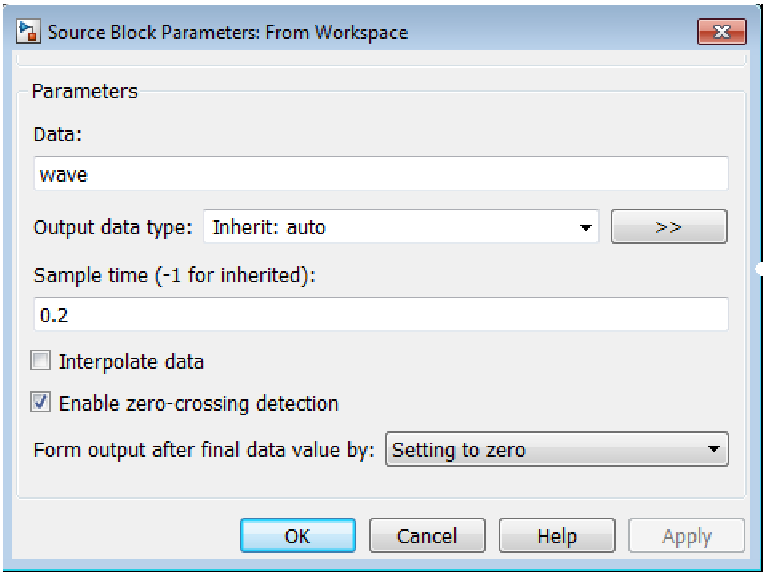
Using Sample Time from Model

If you do not have a time vector, you can define the sample time in your model.

1. In the MATLAB Command Window, enter

   ```
   wave.time = [];
   ```

2. Double-click the **From Workspace** block to open the block parameters dialog. In the **Sample time** field, enter a time interval. For example, enter `0.2`. Clear the **Interpolate data** check box. **From the Form output after final data value by**, select either `Setting to zero`, `Holding final value`, or `Cyclic repetition`. Do not select `Extrapolation`.



Data: wave

Output data type: Inherit: auto

Sample time (-1 for inherited): 0.2

☐ Interpolate data

☑ Enable zero-crossing detection

Form output after final data value by: Setting to zero

Reading 2-D Signals in Structure Format

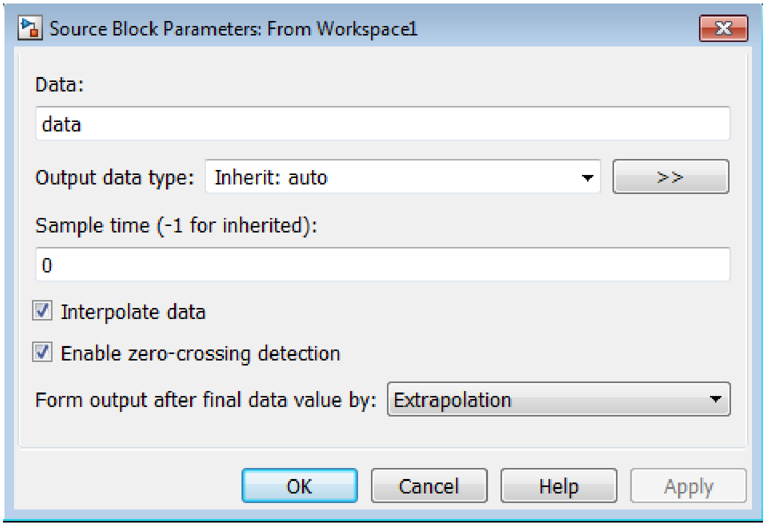
To load 2-D signals from the MATLAB workspace into Simulink, you must have the signals in a structure format. This example creates a 10–by–10 matrix (2-D signal) using the `magic` function, and then creates a 3-D matrix by adding a time vector.

1. In the MATLAB Command Window, enter

   ```
   t1 = [0:0.2:10]';
   m = magic(10);
   M = repmat(m,[1 1 length(t1)]);
   data.time=t1;
   data.signals.values = M;
   data.signals.dimensions=[10 10];
   ```

   The time vector must be a column vector. The `signals.values` `field` is a 3-D matrix where the third dimension is the signal values at a specific time. The `signals.dimensions` field is a two element vector where the first element is the number of rows and the second element is the number of columns in the `signals.values` field.

2 Double-click the **From Workspace** block to open the block parameters dialog. In the **Data** field, enter the name of the structure.



Data:

data

Output data type: Inherit: auto

Sample time (-1 for inherited):

0

☑ Interpolate data

☑ Enable zero-crossing detection

Form output after final data value by: Extrapolation

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled.

See Also

“Import Data”, “Import Data for an Input Test Case”, From File, To File, To Workspace

Function-Call Feedback Latch

Purpose

Break feedback loop involving data signals between function-call blocks

Library

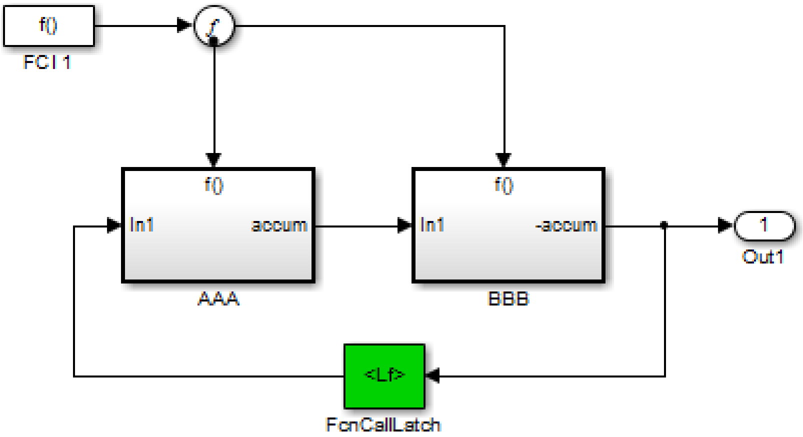
Ports & Subsystems

Description

Use the Function-Call Feedback Latch block to break a feedback loop of data signals between one or more function-call blocks. Specifically, break a loop formed in one of the following ways.

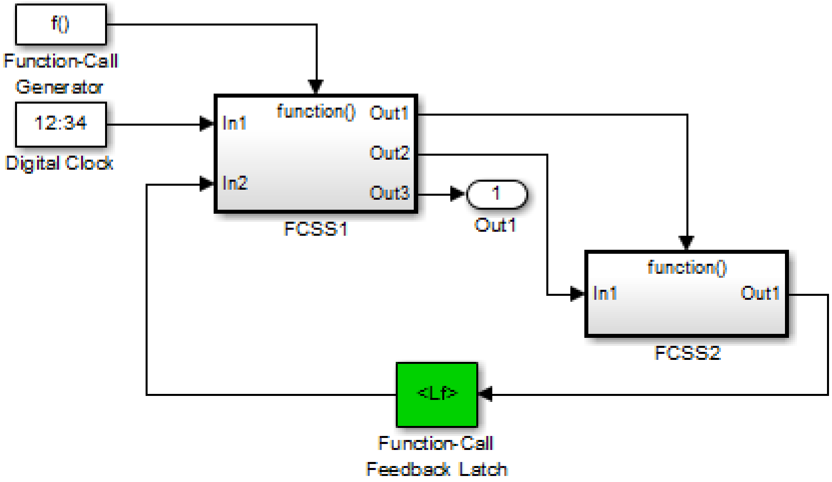
- **When function-call blocks connect to branches of the same function-call signal**

  Place the Function-Call Feedback Latch block on the feedback signal between the branched blocks. As a result, the latch block delays the signal at the input of the destination function-call block, and the destination function-call block executes prior to the source function-call block of the latch block.



- **When the loop involves parent and child function-call blocks, where the child initiator is inside the parent**

  Place the Function-Call Feedback Latch block on the feedback signal between the child and the parent. This arrangement prevents the signal value, read by the parent (FCSS1), from changing during execution of the child. In other words, the parent reads the value from the previous execution of the child (FCSS2).


f()
Function-Call Generator
12:34
Digital Clock
In1
In2
function()
Out1
Out2
Out3
FCSS1
1
Out1
function()
In1
Out1
FCSS2
<Lf>
Function-Call Feedback Latch

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. However, an advantage of the latch block over 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 the 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.

Data Type Support

The Function-Call Feedback Latch block accepts real or complex signals of the following data types:

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

In addition, the latch block accepts bus signals provided that they do not contain any variable-sized signals.

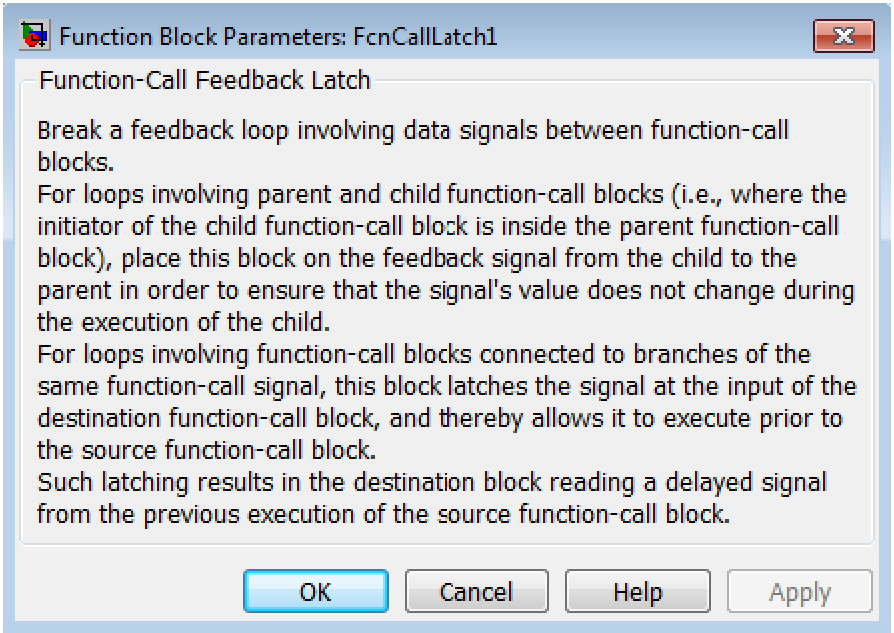
This block does not accept:

- Function-call signals
- Action signals
- Variable-sized signals

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

Parameters and Dialog Box

The Function-Call Feedback Latch block dialog box appears as follows:



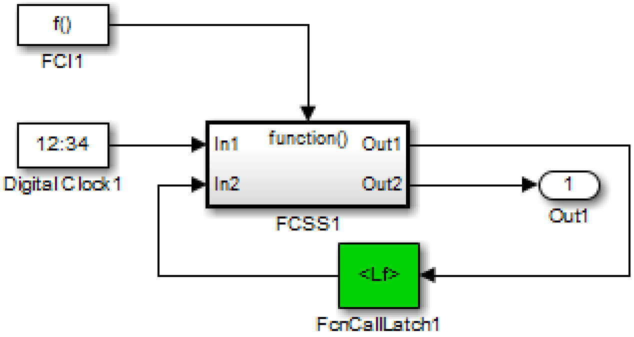
Function Block Parameters: FcnCallLatch1

Function-Call Feedback Latch

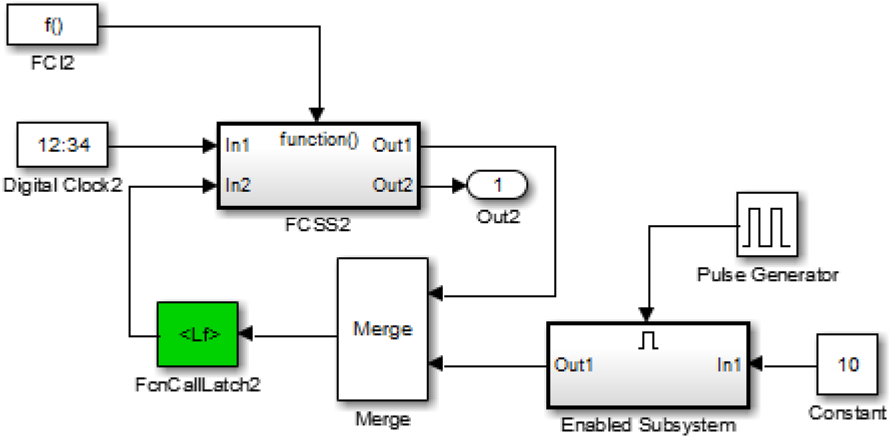
Break a feedback loop involving data signals between function-call blocks.
For loops involving parent and child function-call blocks (i.e., where the initiator of the child function-call block is inside the parent function-call block), place this block on the feedback signal from the child to the parent in order to ensure that the signal's value does not change during the execution of the child.
For loops involving function-call blocks connected to branches of the same function-call signal, this block latches the signal at the input of the destination function-call block, and thereby allows it to execute prior to the source function-call block.
Such latching results in the destination block reading a delayed signal from the previous execution of the source function-call block.

Examples

In the following model, a single function-call subsystem output serves as its own input.


f()
FCI1
12:34
Digital Clock1
In1
In2
function()
Out1
Out2
FCSS1
1
Out1
<Lf>
FcnCallLatch1

A more complex case occurs when a merged signal serves as the input to a function-call subsystem. Latching is necessary if one of the signals entering the Merge block forms a feedback loop with the function-call subsystem. In this example, one of the output signals from FCSS2 combines with the output of an Enabled Subsystem block and then feeds back into an inport of FCSS2.


f()
FCI2
12:34
Digital Clock2
In1
In2
function()
Out1
Out2
FCSS2
1
Out2
Pulse Generator
Merge
Merge
<Lf>
FcnCallLatch2
Out1
In1
Enabled Subsystem
10
Constant

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from the driving block
Scalar Expansion	Yes
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Function-Call Subsystem block

Function-Call Generator

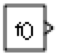
Purpose

Execute function-call subsystem specified number of times at specified rate

Library

Ports & Subsystems

Description

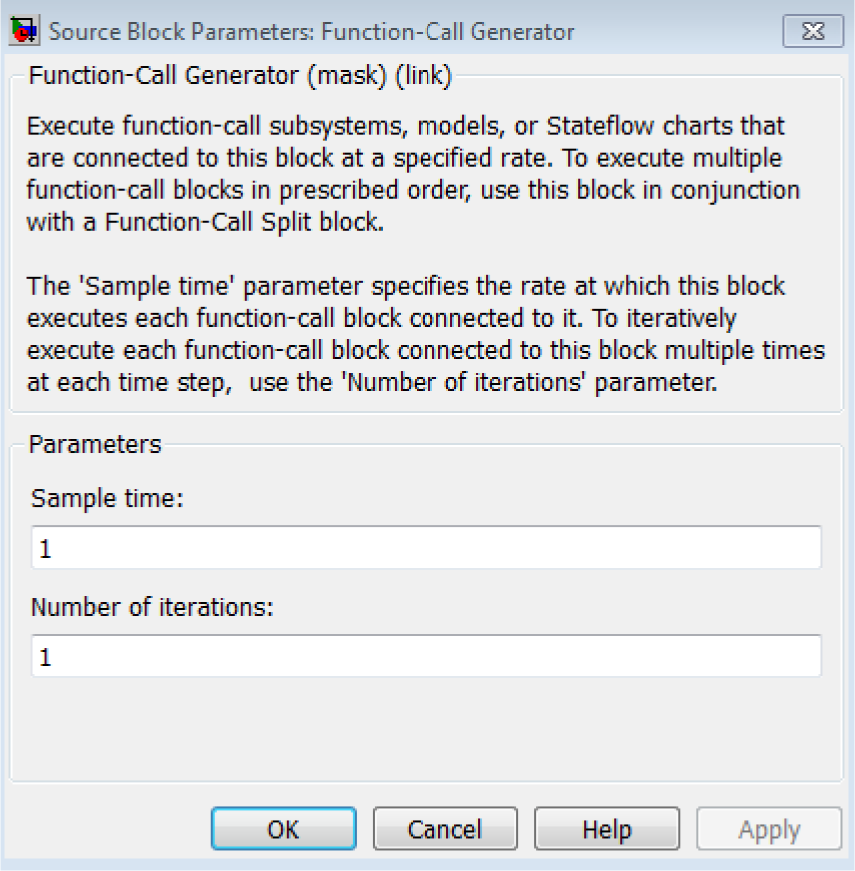


The Function-Call Generator block executes a function-call subsystem (for example, a Stateflow chart acting as a function-call subsystem) at the rate that you specify with the **Sample time** parameter. To iteratively execute each function-call block connected to this block multiple times at each time step, use the 'Number of Iterations' parameter.

To execute multiple function-call subsystems or models in a prescribed order, use the Function-Call Generator block in conjunction with a Function-Call Split block. For an example, see the Function-Call Split block documentation.

Data Type Support

The Function-Call Generator block outputs a signal of type `fcn_call`.



Function-Call Generator (mask) (link)

Execute function-call subsystems, models, or Stateflow charts that are connected to this block at a specified rate. To execute multiple function-call blocks in prescribed order, use this block in conjunction with a Function-Call Split block.

The 'Sample time' parameter specifies the rate at which this block executes each function-call block connected to it. To iteratively execute each function-call block connected to this block multiple times at each time step, use the 'Number of iterations' parameter.

Parameters

Sample time: 1

Number of iterations: 1

Parameters and Dialog Box

Sample time

Specify the time interval between samples. See “Specify Sample Time” in the online documentation for more information.

Number of iterations

Number of times to execute the block per time step. 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.

Characteristics

Direct Feedthrough	No
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Zero-Crossing Detection	No

Purpose

Provide junction for splitting function-call signal

Library

Ports & Subsystems

Description

The Function-Call Split block allows a function-call signal to be branched and connected to several function-call subsystems and models.

Several Function-Call Split blocks may be cascaded one after another to create multiple branches for a function-call signal. An advantage of the Function-Call Split block is that in some cases, using this block eliminates the need for the initiator block to create multiple function-call signals for invoking a set of function-call subsystems or models.

Function-call subsystems or models connected to the output port of the Function-Call Split block marked with a dot execute before the subsystems or models connected to the other output port. If any data dependencies between these subsystems (or between models) do not support the specified execution order, then an error occurs. This implies that an error will always occur when you create a data-dependency loop involving function-call subsystems or models connected to branches of the same function-call signal. To eliminate this error, consider turning on the **Latch input for feedback signals of function-call subsystem outputs** on one or more Inport blocks of the function-call subsystems (or models) involved in the loop. Setting this option to `on` has the effect of delaying the corresponding input signal, thereby eliminating the data-dependency loop.

In order 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 turn on the model option **Format > Block Displays > Sorted Order**, then the execution order of function-call subsystems connected to branches of a given function-call signal displays on the blocks . Each subsystem has an execution order of the form S:B#{C}. Here, # 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.

Limitations

The Function-Call Split block has these limitations:

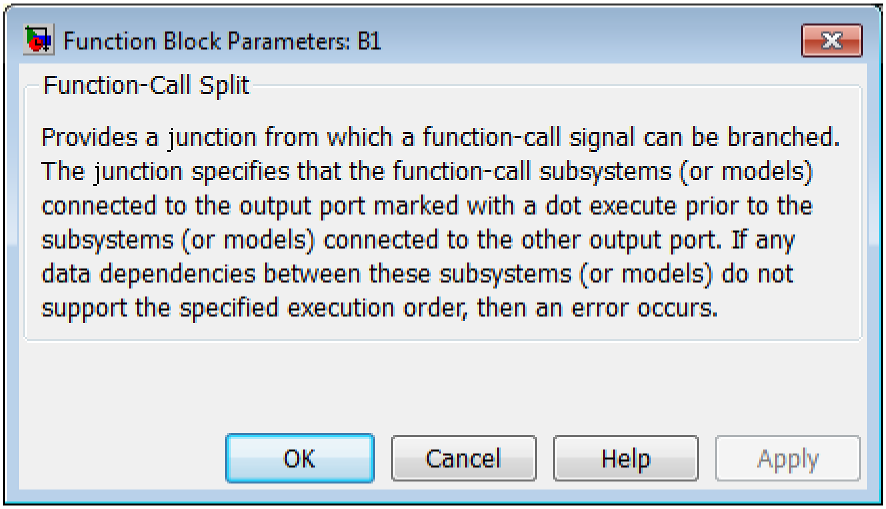
Limitations
All function-call subsystems and models connected to a given function-call signal must reside within the same nonvirtual layer of the model hierarchy.
Branched function-call subsystems (or models) and their children cannot be connected directly back to the initiator.
Function-call subsystems and models connected to branches of a function-call signal cannot have multiple (muxed) initiators.

Data Type Support

The Function-Call Split block accepts periodic and asynchronous function-call signals only.

Parameters and Dialog Box

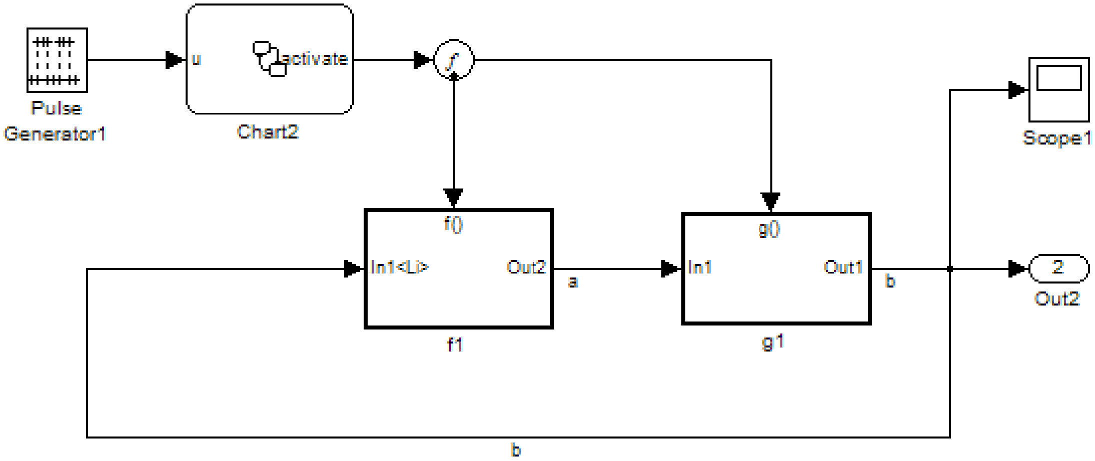
The Function-Call Split block dialog box appears as follows:



Provides a junction from which a function-call signal can be branched. The junction specifies that the function-call subsystems (or models) connected to the output port marked with a dot execute prior to the subsystems (or models) connected to the other output port. If any data dependencies between these subsystems (or models) do not support the specified execution order, then an error occurs.

Examples

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 and thus breaks the loop of data dependencies between subsystems f1 and g1. The model thus achieves the desired 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.



Characteristics

Sample Time	Inherited from the block driving the function-call split
Scalar Expansion	No
Dimensionalized	No
Multidimensionalized	No
Virtual	Yes

See Also

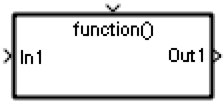
Function-Call Subsystem | Function-Call Generator

Function-Call Subsystem

Purpose Represent subsystem that can be invoked as function by another block

Library Ports & Subsystems

Description



The Function-Call Subsystem block is a Subsystem block that is preconfigured to serve as a starting point for creating a function-call subsystem. For more information, see “Function-Call Subsystems” in the “Creating a Model” chapter of the Simulink documentation.

Purpose

Multiply input by constant

Library

Math Operations

Description

The Gain block multiplies the input by a constant value (gain). The input and the gain can each be a scalar, vector, or matrix.

You specify the value of the gain in the **Gain** parameter. The **Multiplication** parameter lets you specify element-wise or matrix multiplication. For matrix multiplication, this parameter also lets you indicate the order of the multiplicands.

The gain is converted from doubles to the data specified in the block mask offline using round-to-nearest and saturation. The input and gain are then multiplied, and the result is converted to the output data type using the specified rounding and overflow modes.

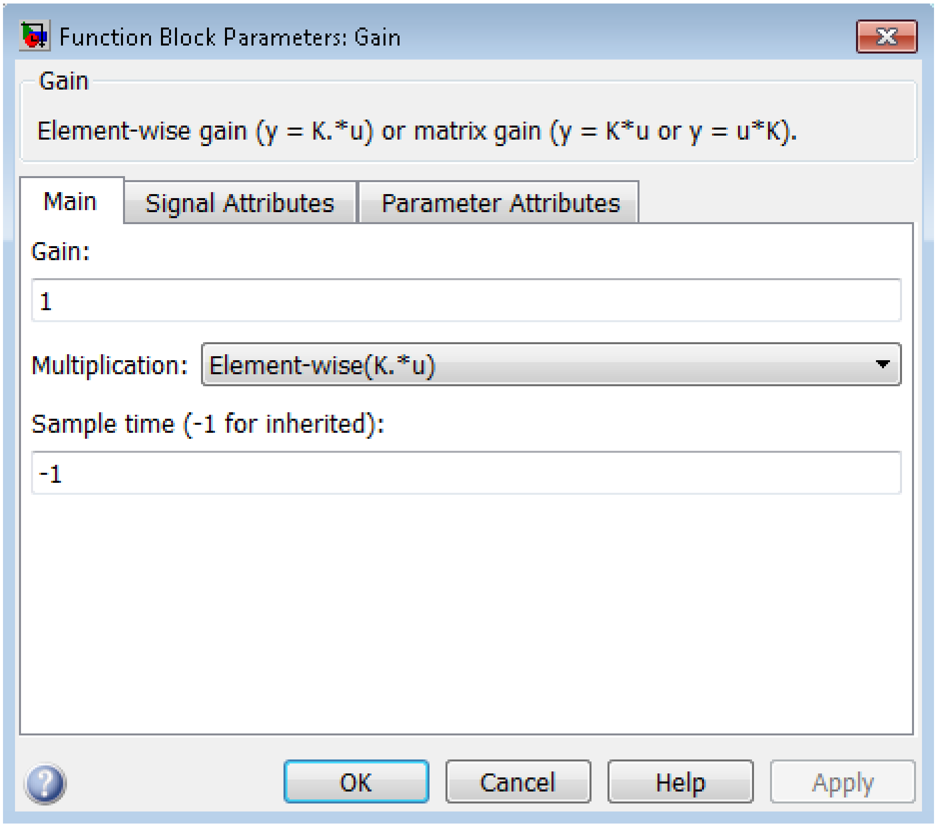
Data Type Support

The Gain block accepts a real or complex scalar, vector, or matrix of any numeric data type that Simulink supports. The Gain block supports fixed-point data types. If the input of the Gain block is real and the gain is complex, the output is complex.

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

Parameters and Dialog Box

The **Main** pane of the Gain block dialog box appears as follows:



Function Block Parameters: Gain

Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

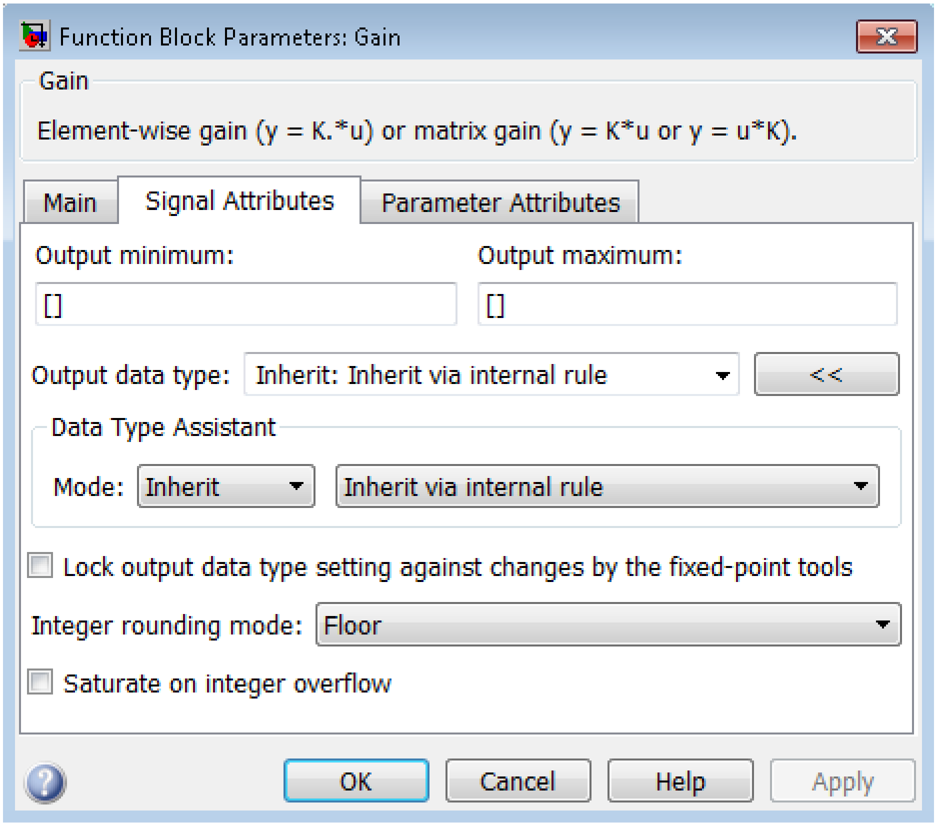
Gain: 1

Multiplication: Element-wise(K.*u)

Sample time (-1 for inherited): -1

Main
Signal Attributes
Parameter Attributes
OK
Cancel
Help
Apply

The **Signal Attributes** pane of the Gain block dialog box appears as follows:



Function Block Parameters: Gain

Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Main
Signal Attributes
Parameter Attributes

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

<<

Data Type Assistant

Mode: Inherit — Inherit via internal rule

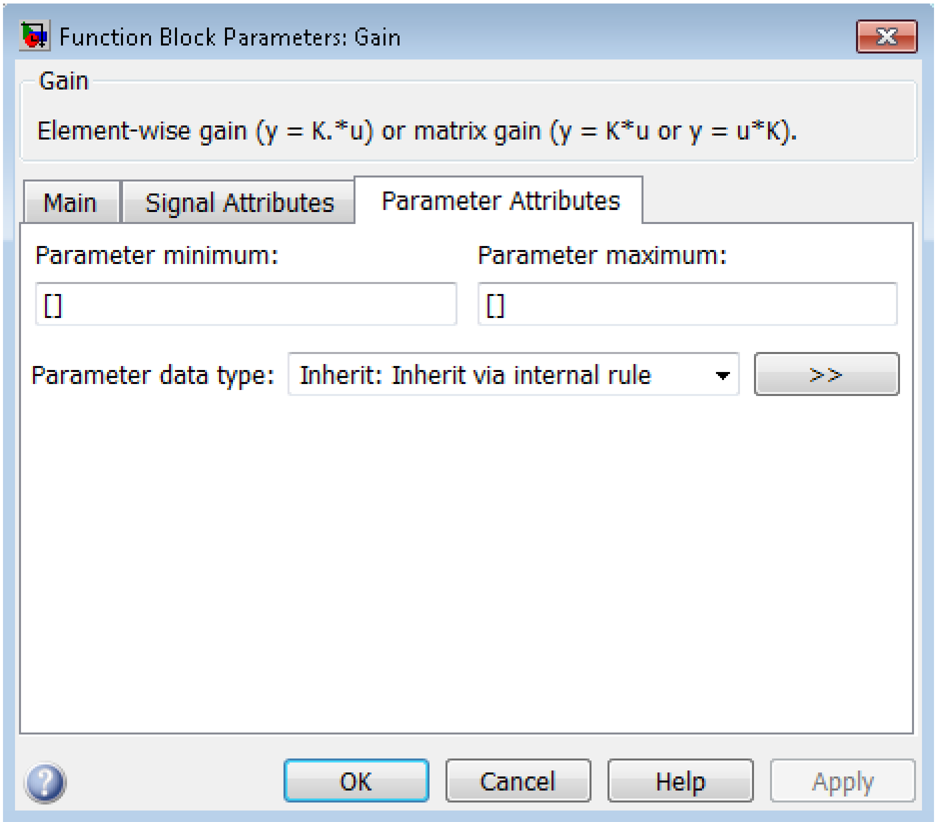
☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate on integer overflow

OK
Cancel
Help
Apply

The **Parameter Attributes** pane of the Gain block dialog box appears as follows:


Function Block Parameters: Gain
Gain
Main
Signal Attributes
Parameter Attributes
>>
OK
Cancel
Help
Apply

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Parameter minimum: []

Parameter maximum: []

Parameter data type: Inherit: Inherit via internal rule

Gain

Specify the value by which to multiply the input.

Settings

Default: 1
Minimum: value of **Parameter minimum** parameter
Maximum: value of **Parameter maximum** parameter

The gain can be a scalar, vector, or matrix.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Multiplication

Specify the multiplication mode.

Settings

Default: `Element-wise(K.*u)`

`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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Specify the time interval between samples.

Settings

Default: -1

To inherit the sample time, set this parameter to `-1`.

See "Specify Sample Time" in the online documentation for more information.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the minimum to perform:

- Parameter range checking (see "Check Parameter Values") for some blocks
- Simulation range checking (see "Signal Ranges" and "Enabling Simulation Range Checking")
- Automatic scaling of fixed-point data types

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type `int8` by a gain of `int16` and `ASIC/FPGA` is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified (assume 32-bit Generic)`, i.e., a generic 32-bit microprocessor, is specified as the target hardware, the output data type is `int32`. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error message in the Simulation Diagnostics Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`
Use data type of the driving block.

`Inherit: Same as input`
Use data type of input signal.

`double`
Output data type is `double`.

`single`
Output data type is `single`.

`int8`
Output data type is `int8`.

`uint8`
Output data type is `uint8`.

`int16`
Output data type is `int16`.

`uint16`
Output data type is `uint16`.

`int32`
Output data type is `int32`.

`uint32`
Output data type is `uint32`.

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

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

`<data type expression>`
Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See "Specify Block Output Data Types" for more information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`
: Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

  - `Inherit via internal rule` (default)
  - `Inherit via back propagation`
  - `Same as input`

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

  - `double` (default)
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`
  - `uint32`

`Fixed point`
: Fixed-point data types.

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

Dependency

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`

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

`Off`

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

Tip

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

Dependency

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

Signedness

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

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Word length

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

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Scaling

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

Settings

Default: `Binary point`

`Binary point`
  Specify binary point location.

`Slope and bias`
  Enter slope and bias.

Dependencies

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

Selecting `Binary point` enables:

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

Selecting `Slope and bias` enables:

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

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

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

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

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

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
: Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`
: Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`
: Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`
: Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`
: Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`
: Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`
: Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see “Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Parameter minimum

Specify the minimum value of the gain.

Settings

Default: `[]`

The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")
- Automatic scaling of fixed-point data types

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Parameter maximum

Specify the maximum value of the gain.

Settings

Default: `[]`

The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")
- Automatic scaling of fixed-point data types

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Parameter data type

Specify the data type of the **Gain** parameter.

Settings

Default: `Inherit:  Inherit via internal rule`

`Inherit:  Inherit via internal rule`
: Use an internal rule to inherit the data type.

`Inherit:  Same as input`
: Use data type of sole input signal.

`Inherit:  Inherit from 'Gain'`
: Use data type of the **Gain** value. For example:

If you set Gain to...	The parameter data type inherits...
`2`	`double`
`single(2)`	`single`
`int8(2)`	`int8`

`double`
: Data type is `double`.

`single`
: Data type is `single`.

`int8`
: Data type is `int8`.

`uint8`
: Data type is `uint8`.

`int16`
: Data type is `int16`.

`uint16`
: Data type is `uint16`.

`int32`
: Data type is `int32`.

`uint32`
Data type is `uint32`.

`fixdt(1,16)`
Data type is `fixdt(1,16)`.

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

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

`<data type expression>`
Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule` (default)
- `Same as input`
- `Inherit from 'Gain'`

`Built in`

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

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

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

Dependency

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant" in the Simulink documentation.

Scaling

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

Settings

Default: `Best precision`

`Binary point`
  Specify binary point location.

`Slope and bias`
  Enter slope and bias.

`Best precision`
  Specify best-precision values.

Dependencies

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

Selecting `Binary point` enables:

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

Selecting `Slope and bias` enables:

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

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

Examples

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

- `sldemo_bounce`
- `sldemo_tonegen_fixpt`
- `sldemo_hardstop`
- `sldemo_enginewc`

Characteristics

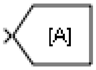
Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of input and **Gain** parameter for `Element-wise(K.*u)` multiplication
Dimensionalized	Yes
Multidimensionalized	Yes, only if the **Multiplication** parameter specifies `Element-wise(K.*u)`
Zero-Crossing Detection	No

Goto

Purpose Pass block input to From blocks

Library Signal Routing

Description The Goto block passes its input to its corresponding From blocks. The input can be a real- or complex-valued signal or vector of any data type. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them.



A Goto block can pass its input signal to more than one From block, although a From block can receive a signal from only one Goto block. The input to that Goto block is passed to the From blocks associated with it as though the blocks were physically connected. Goto blocks and From blocks are matched by the use of Goto tags.

The **Tag Visibility** parameter determines whether the location of From blocks that access the signal is limited:

- `local`, the default, means that From and Goto blocks using the same tag must be in the same subsystem. A local tag name is enclosed in brackets ([]).
- `scoped` means that From and Goto blocks using the same tag must be in the same subsystem or at any level in the model hierarchy below the Goto Tag Visibility block that does not entail crossing a nonvirtual subsystem boundary, i.e., the boundary of an atomic, conditionally executed, or function-call subsystem or a model reference. A scoped tag name is enclosed in braces ({}).
- `global` means that From and Goto blocks using the same tag can be anywhere in the model except in locations that span nonvirtual subsystem boundaries.

The rule that From-Goto block connections cannot cross nonvirtual subsystem boundaries has the following exception. A Goto block connected to a state port in one conditionally executed subsystem is visible to a From block inside another conditionally executed subsystem. For more information about conditionally executed subsystems, see "Conditional Execution Behavior".

Note A `scoped` Goto block in a masked system is visible only in that subsystem and in the nonvirtual subsystems it contains. Simulink generates an error if you run or update a diagram that has a Goto Tag Visibility block at a higher level in the block diagram than the corresponding `scoped` Goto block in the masked subsystem.

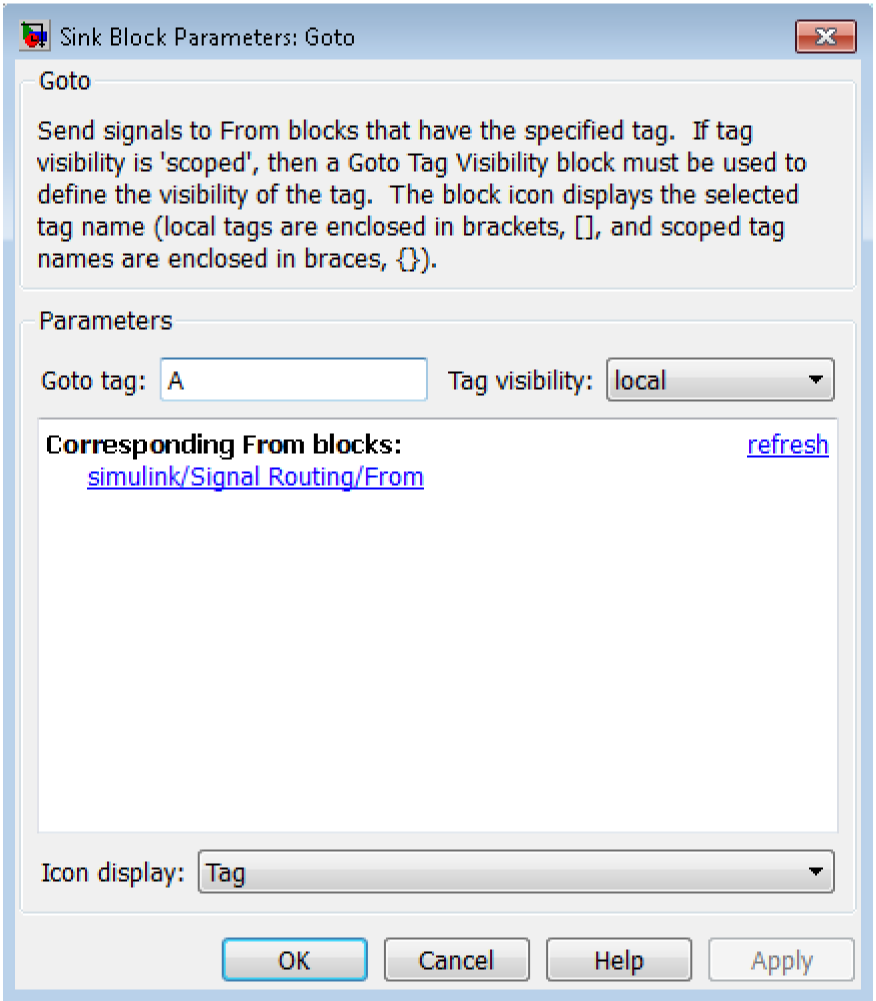
Use local tags when the Goto and From blocks using the same tag name reside in the same subsystem. You must use global or scoped tags when the Goto and From blocks using the same tag name reside in different subsystems. When you define a tag as global, all uses of that tag access the same signal. A tag defined as scoped can be used in more than one place in the model.

The Goto block supports signal label propagation.

Data Type Support

The Goto block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

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


Sink Block Parameters: Goto
Goto

Send signals to From blocks that have the specified tag. If tag visibility is 'scoped', then a Goto Tag Visibility block must be used to define the visibility of the tag. The block icon displays the selected tag name (local tags are enclosed in brackets, [], and scoped tag names are enclosed in braces, {}).

Parameters

Goto tag: A

Tag visibility: local

Corresponding From blocks:

simulink/Signal Routing/From

Icon display: Tag

OK
Cancel
Help
Apply

Parameters and Dialog Box

Goto Tag

The Goto block identifier. This parameter identifies the Goto block whose scope is defined in this block.

Tag Visibility

The scope of the Goto block tag: `local`, `scoped`, or `global`. The default is `local`.

Corresponding From blocks

List of the From blocks connected to this Goto block. Double-clicking any entry in this list displays and highlights the corresponding From block.

Icon Display

Specifies the text to display on the block's icon. The options are the block's tag, the name of the signal that the block represents, or both the tag and the signal name.

Examples

The following models show how to use the Goto block:

- `sldemo_auto_climatecontrol`
- `sldemo_hardstop`

Characteristics

Sample Time	Inherited from driving block
Dimensionalized	Yes
Multidimensionalized	Yes
Virtual	Yes For more information, see "Virtual Blocks" in the Simulink documentation.
Zero-Crossing Detection	No

See Also

From

Goto Tag Visibility

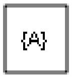
Purpose

Define scope of Goto block tag

Library

Signal Routing

Description

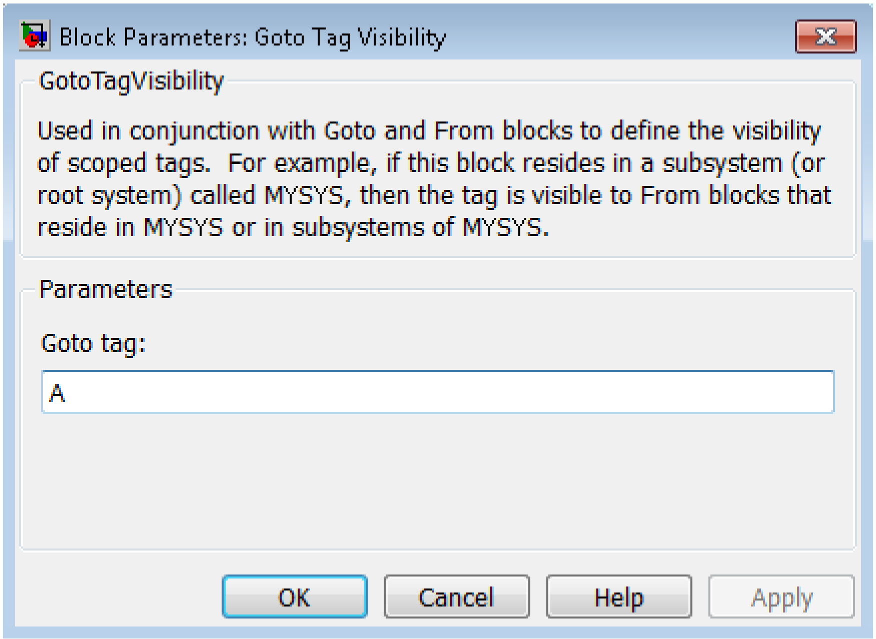


The Goto Tag Visibility block defines the accessibility of Goto block tags that have `scoped` visibility. The tag specified as the **Goto tag** parameter is accessible by From blocks in the same subsystem that contains the Goto Tag Visibility block and in subsystems below it in the model hierarchy.

A Goto Tag Visibility block is required for Goto blocks whose **Tag Visibility** parameter value is `scoped`. No Goto Tag Visibility block is needed if the tag visibility is either `local` or `global`. The block shows the tag name enclosed in braces ({}).

Data Type Support

Not applicable.



Block Parameters: Goto Tag Visibility

GotoTagVisibility

Used in conjunction with Goto and From blocks to define the visibility of scoped tags. For example, if this block resides in a subsystem (or root system) called MYSYS, then the tag is visible to From blocks that reside in MYSYS or in subsystems of MYSYS.

Parameters

Goto tag:

A

Parameters and Dialog Box

Goto tag

The Goto block tag whose visibility is defined by the location of this block.

Characteristics

Sample Time	N/A
Dimensionalized	N/A
Virtual	Yes For more information, see “Virtual Blocks” in the Simulink documentation.

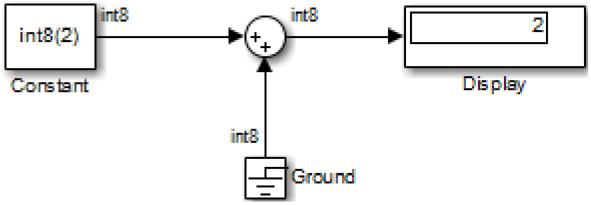
Ground

Purpose Ground unconnected input port

Library Sources

Description The Ground block connects to blocks whose input ports do not connect to other blocks. If you run a simulation with blocks having unconnected input ports, Simulink issues warnings. Using a Ground block to ground those unconnected blocks can prevent these warnings.

The Ground block outputs a signal of the same data type as the port to which it connects. For example, consider the following model:



In this example, the output of the Constant block determines the data type (`int8`) of the port to which the Ground block is connected. That port determines the output data type of the Ground block.

The Ground block outputs a signal with zero value. When the output data type cannot represent zero exactly, the Ground block outputs a nonzero value that is the closest possible value to zero. This behavior applies only to fixed-point data types with nonzero bias. The following expressions are examples of fixed-point data types that cannot represent zero:

- `fixdt(0, 8, 1, 1)` — an unsigned 8-bit type with slope of 1 and bias of 1
- `fixdt(1, 8, 6, 3)` — a signed 8-bit type with slope of 6 and bias of 3

If the output is an enumerated data type, the Ground block outputs the default value of the enumeration. This behavior applies whether or not:

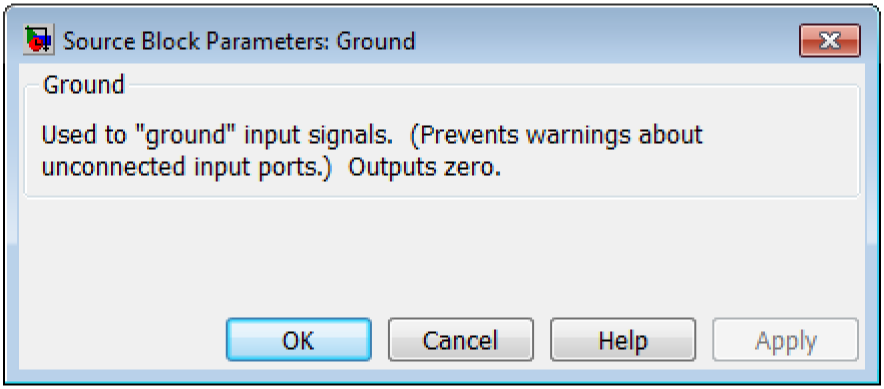
- The enumeration can represent zero.
- The default value of the enumeration is zero.

If the enumerated type does not have a default value, the Ground block outputs the first enumeration value in the type definition.

Data Type Support

The Ground block supports all data types that Simulink supports, including fixed-point and enumerated data types. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



Ground

Used to "ground" input signals. (Prevents warnings about unconnected input ports.) Outputs zero.

Examples

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

- `sldemo_doublebounce`

Characteristics

Sample Time	Constant (`inf`)
Dimensionalized	Yes

Multidimensionalized	Yes
Virtual	Yes For more information, see “Virtual Blocks” in the Simulink documentation.

Purpose

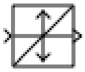
Detect crossing point

Library

Discontinuities

Description

The Hit Crossing block detects when the input reaches the **Hit crossing offset** parameter value in the direction specified by the **Hit crossing direction** property.



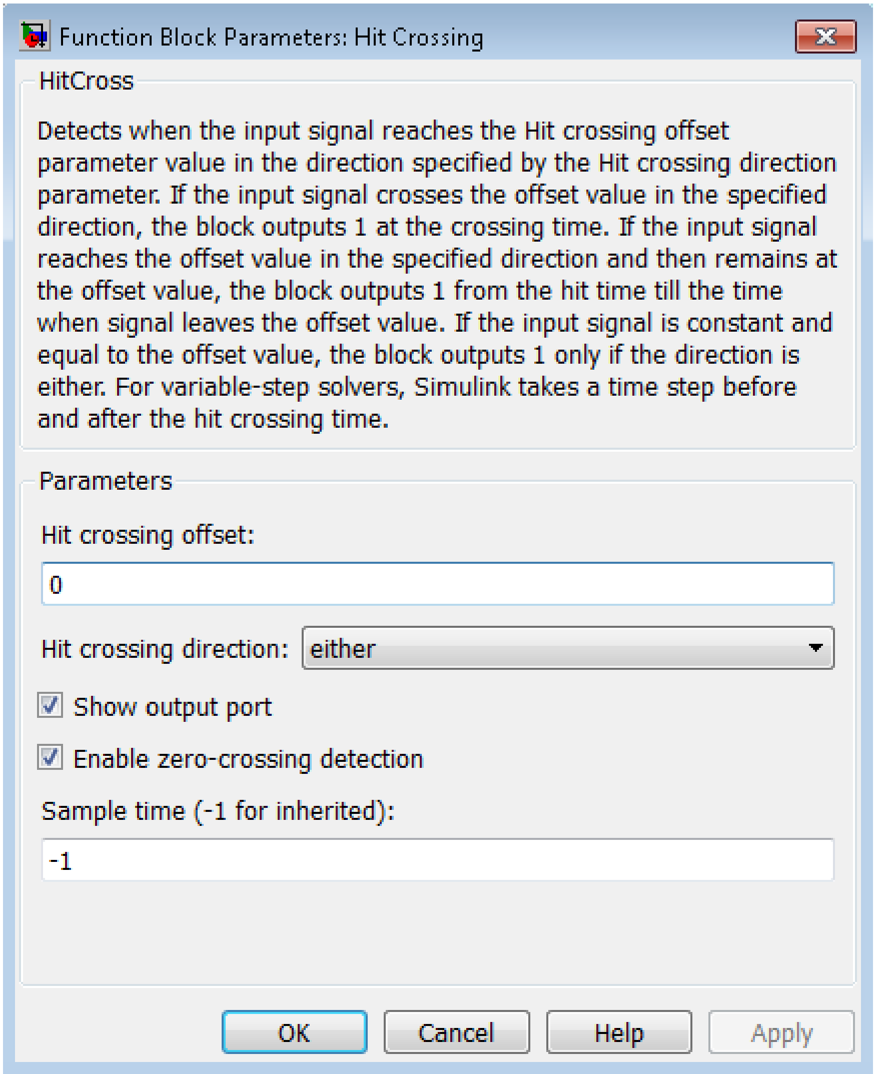
The block accepts one input of type `double`. If you select the **Show output port** check box, the block output indicates when the crossing occurs. If the input signal is exactly the value of the offset value after the hit crossing is detected, the block continues to output a value of 1. If the input signals at two adjacent points bracket the offset value (but neither value is exactly equal to the offset), the block outputs a value of 1 at the second time step. If the **Show output port** check box is *not* selected, the block ensures that the simulation finds the crossing point but does not generate output. If the input signal is constant and equal to the offset value, the block outputs 1 only if the **Hit crossing direction** property is set to `either`.

When the block's **Hit crossing direction** property is set to `either`, the block serves as an "Almost Equal" block, useful in working around limitations in finite mathematics and computer precision. Used for these reasons, this block might be more convenient than adding logic to your model to detect this condition.

When the block's **Hit crossing direction** property is set to `either` and the model uses a fixed-step solver, the block has the following behavior. If the output signal is 1, the block sets the output signal to 0 at the next time step, unless the input signal equals the offset value.

Data Type Support

The Hit Crossing block outputs a signal of type `Boolean` if Boolean logic signals are enabled (see “Implement logic signals as Boolean data (vs. double)”). Otherwise, the block outputs a signal of type `double`.



HitCross

Detects when the input signal reaches the Hit crossing offset parameter value in the direction specified by the Hit crossing direction parameter. If the input signal crosses the offset value in the specified direction, the block outputs 1 at the crossing time. If the input signal reaches the offset value in the specified direction and then remains at the offset value, the block outputs 1 from the hit time till the time when signal leaves the offset value. If the input signal is constant and equal to the offset value, the block outputs 1 only if the direction is either. For variable-step solvers, Simulink takes a time step before and after the hit crossing time.

Parameters

Hit crossing offset: 0

Hit crossing direction: either

☑ Show output port

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

Parameters

and Dialog Box

Hit crossing offset
The value whose crossing is to be detected.

Hit crossing direction
The direction from which the input signal approaches the hit crossing offset for a crossing to be detected.

Show output port
If selected, draw an output port.

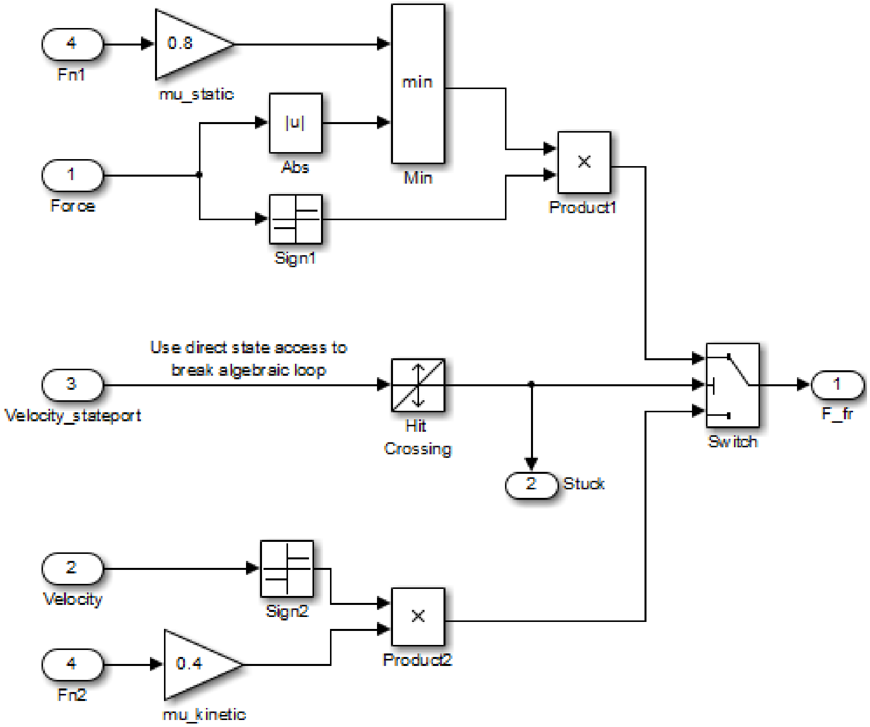
Enable zero-crossing detection
Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Sample time (-1 for inherited)
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

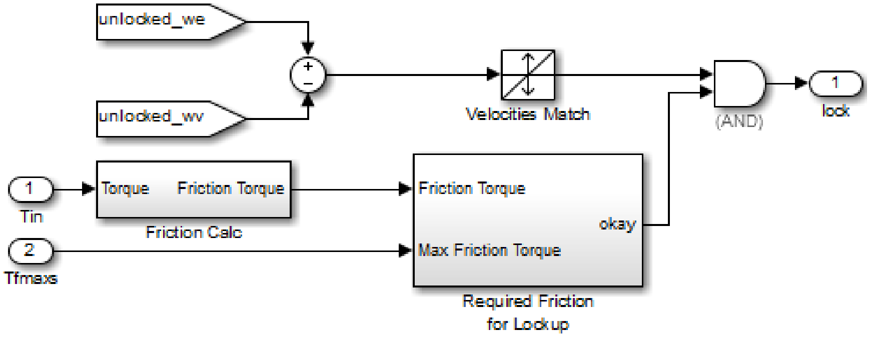
Examples

The `sldemo_hardstop` and `sldemo_clutch` models show how you can use the Hit Crossing block.

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


4
Fn1
0.8
mu_static
min
Min
|u|
Abs
1
Force
Product1
Sign1
Use direct state access to
break algebraic loop
3
Velocity_stateport
Hit
Crossing
Switch
1
F_fr
2
Stuck
2
Velocity
Sign2
Product2
4
Fn2
0.4
mu_kinetic

In the `sldemo_clutch` model, the Hit Crossing block is in the Friction Mode Logic/Lockup Detection subsystem.


unlocked_we
unlocked_wv
+
–
Velocities Match
(AND)
1
lock
1
Tin
Torque
Friction Torque
Friction Calc
2
Tfmaxs
Friction Torque
okay
Max Friction Torque
Required Friction
for Lockup

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled.

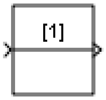
Purpose

Set initial value of signal

Library

Signal Attributes

Description



The IC block sets the initial condition of the signal at its input port, for example, the value of the signal at the simulation start time (t_{start}). The block does this by outputting the specified initial condition when you start the simulation, regardless of the actual value of the input signal. Thereafter, the block outputs the actual value of the input signal.

Note If an IC block inherits or specifies a nonzero sample time offset (t_{offset}), the IC block outputs its initial value at time t,

$$t = n * t_{period} + t_{offset}$$

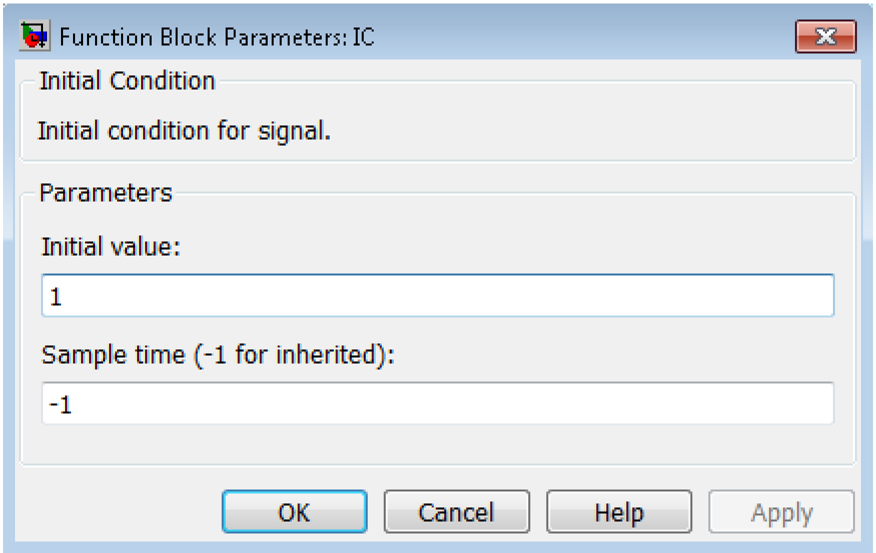
where n is the smallest integer such that $t \qquad t_{start}$.

That is, the IC block outputs its initial value the first time blocks with sample time $[t_{period},\ t_{offset}]$ execute, which can be after t_{start}.

The IC block is useful for providing an initial guess for the algebraic state variables in a loop. For more information, see “Algebraic Loops”.

Data Type Support

The IC block accepts and outputs signals of any Simulink built-in and fixed-point data type. The **Initial value** parameter accepts any built-in data type that Simulink supports. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.


Function Block Parameters: IC
Initial Condition
Parameters
OK
Cancel
Help
Apply

Initial condition for signal.

Initial value:

1

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Initial value

Specify the initial value for the input signal.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Examples

The following examples show how to use the IC block:

- `sldemo_bounce`
- `sldemo_hardstop`
- `sldemo_enginewc`

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of **Initial value** parameter only
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

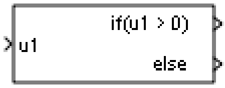
Purpose

Model `if-else` control flow

Library

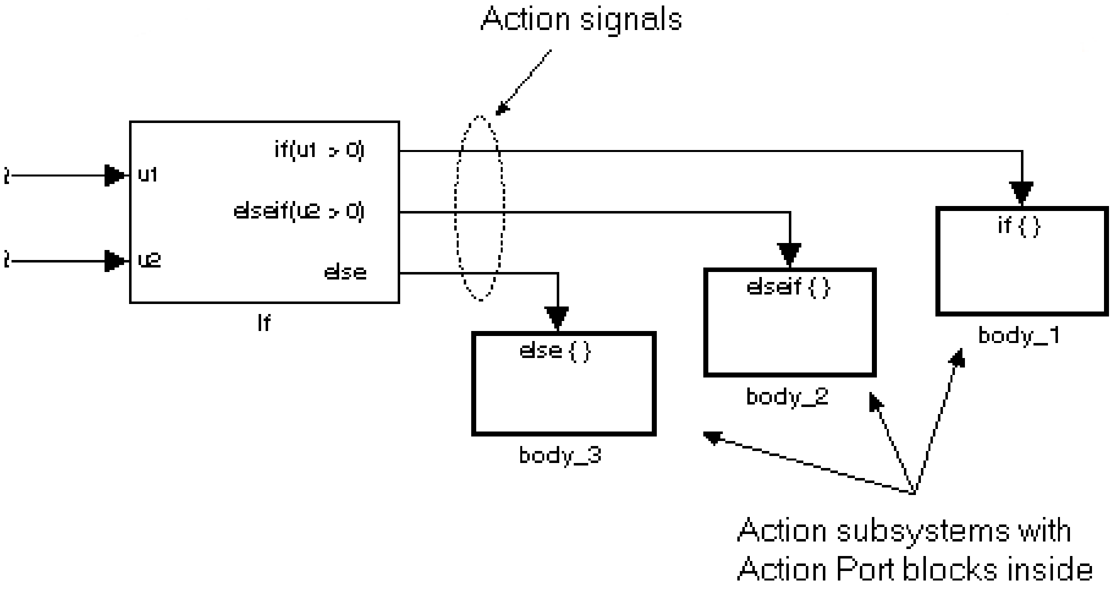
Ports & Subsystems

Description


u1
if(u1 > 0)
else

The If block, along with the If Action Subsystem block containing an Action Port, implements standard C-like `if-else` logic.

The following shows a completed `if-else` control flow statement.


Action signals
u1
u2
if(u1 > 0)
elseif(u2 > 0)
else
If
if { }
body_1
elseif { }
body_2
else { }
body_3
Action subsystems with
Action Port blocks inside

In this example, the inputs to the If block determine the values of conditions represented as output ports. Each output port is attached to an If Action Subsystem, named `body_1`, `body_2`, and `body_3`. The conditions are evaluated top down starting with the `if` condition. If a condition is true, its If Action Subsystem is executed and the If block does not evaluate any remaining conditions.

The preceding `if-else` control flow statement can be represented by the following pseudocode.

```
if (u1 > 0) {
 body_1;
}
else if (u2 > 0){
 body_2;
}
else {
 body_3;
}
```

You construct a Simulink `if-else` control flow statement like the preceding example as follows:

1. Place an If block in the current system.
2. Open the dialog of the If block and enter as follows:
   - Enter the **Number of inputs** field with the required number of inputs necessary to define conditions for the `if-else` control flow statement.

     Elements of vector inputs can be accessed for conditions using (row, column) arguments. For example, you can specify the fifth element of the vector u2 in the condition u2(5) > 0 in an **If expression** or **Elseif expressions** field.
   - Enter the expression for the if condition of the `if-else` control flow statement in the **If expression** field.

This creates an if output port for the If block with a label of the form `if(condition)`. This is the only required If Action signal output for an If block.

- Enter expressions for any elseif conditions of the `if-else` control flow statement in the **Elseif expressions** field.

  Use a comma to separate one condition from another. Entering these conditions creates an output port for the If block for each condition, with a label of the form `elseif(condition)`. elseif ports are optional and not required for operation of the If block.

- Check the **Show else condition** check box to create an else output port.

  The else port is optional and not required for the operation of the If block.

3. Create If Action subsystems to connect to each of the if, else, and elseif ports.

   These consist of a subsystem with an Action Port block. When you place an Action Port block inside each subsystem, an input port named Action is added to the subsystem.

4. Connect each if, else, and elseif port of the If block to the Action port of an If Action subsystem.

   When you make the connection, the icon for the If Action block is renamed to the type of the condition that it attaches to.

   Note During simulation of an `if-else` control flow statement, the Action signal lines from the If block to the If Action subsystems turn from solid to dashed.

5. In each If Action Subsystem, enter the Simulink blocks appropriate to the body to be executed for the condition it handles.

Note All blocks in an If Action Subsystem must run at the same rate as the driving If block. You can achieve this by setting each block's sample time parameter to be either inherited (`-1`) or the same value as the If block's sample time.

Limitations

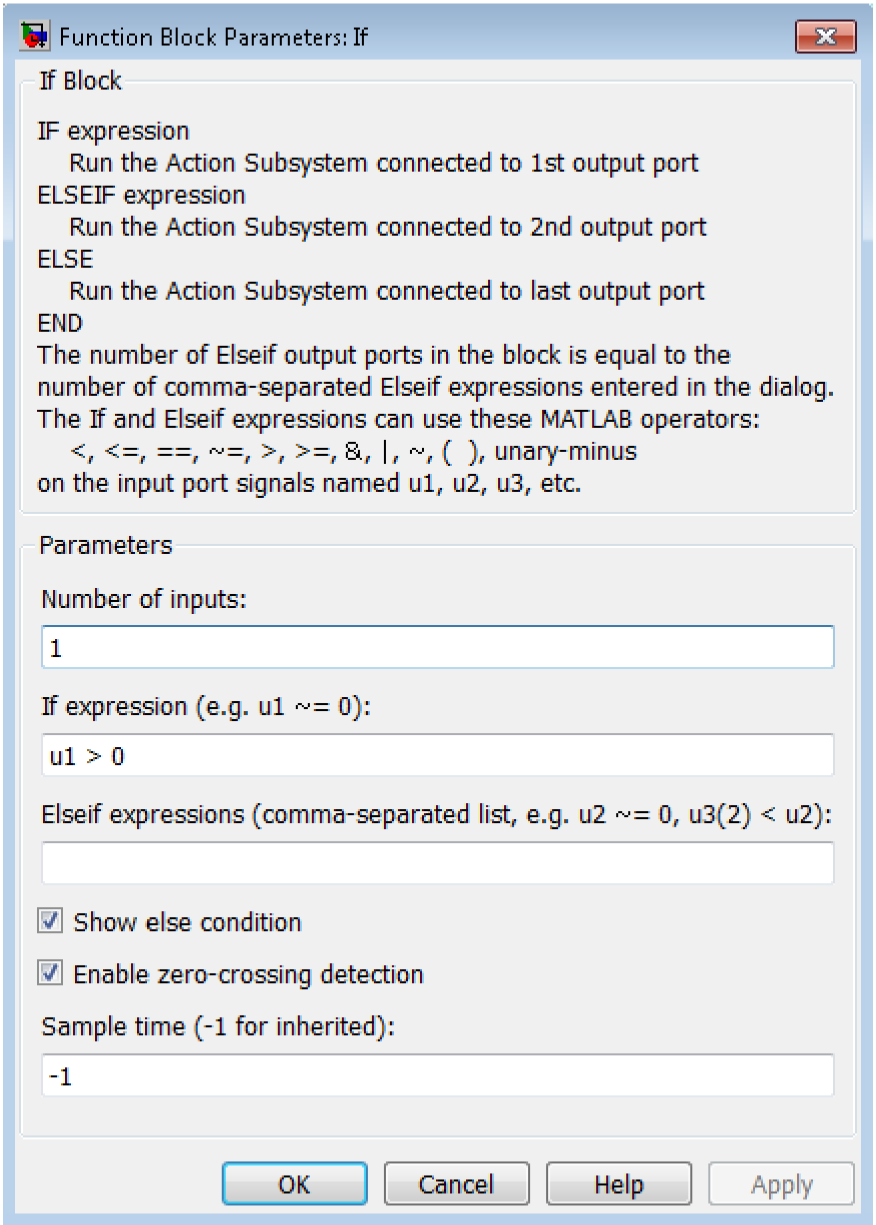
The If block has the following limitations:

- You cannot tune an if or elseif expression during simulation in Normal or Accelerator mode (see "How Acceleration Modes Work"), or in generated code. The If block doesn't support tunable parameters if **Inline Parameters** is turned on. To implement tunable if/else expressions, tune the expression outside the If block. For example, use the Relational operator block to evaluate the expression outside or add the tunable parameter as an input to the If block.
- The If block does not support custom storage classes. See "Custom Storage Classes" in the Embedded Coder documentation.

Data Type Support

Inputs `u1,u2,...,un` can be scalars or vectors of any built-in Simulink data type and must all be of the same data type. The inputs cannot be of any user-defined type, such as an enumerated type. Outputs from the `if`, `else`, and `elseif` ports are Action signals to If Action subsystems that you create by using Action Port blocks and subsystems.

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



Function Block Parameters: If

If Block

IF expression
Run the Action Subsystem connected to 1st output port
ELSEIF expression
Run the Action Subsystem connected to 2nd output port
ELSE
Run the Action Subsystem connected to last output port
END
The number of Elseif output ports in the block is equal to the number of comma-separated Elseif expressions entered in the dialog.
The If and Elseif expressions can use these MATLAB operators:
<, <=, ==, ~=, >, >=, &, |, ~, ( ), unary-minus
on the input port signals named u1, u2, u3, etc.

Parameters

Number of inputs: 1

If expression (e.g. u1 ~= 0): u1 > 0

Elseif expressions (comma-separated list, e.g. u2 ~= 0, u3(2) < u2):

☑ Show else condition

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

Parameters and Dialog Box

Number of inputs

Specify the number of inputs to the If block. These appear as data input ports labeled with a `'u'` character followed by a number, `1,2,...,n`, where `n` equals the number of inputs that you specify.

If expression

Specify the condition for the `if` output port. This condition appears on the If block adjacent to the `if` output port. The `if` expression can use any of the following operators: `<.  <=, ==, ~=, >, >=, &, |, ~, (), unary-minus`. The If Action subsystem attached to the `if` port executes if its condition is true. The expression must not contain data type expressions, for example, `int8(6)`, and must not reference workspace variables whose data type is other than `double` or `single`.

Note There are limitations to the tunability of if expressions. See "Limitations" on page 1-762 for more information.

Elseif expressions

Specify a string list of `elseif` conditions delimited by commas. These conditions appear below the `if` port and above the `else` port when you select the **Show else condition** check box. `elseif` expressions can use any of the following operators: `<, <=, ==, ~=, >, >=, &, |, ~, (), unary-minus`. The If Action subsystem attached to an `elseif` port executes if its condition is true and all of the `if` and `elseif` conditions are false. The expression must not contain data type expressions, for example, `int8(6)`, and must not reference workspace variables whose data type is other than `double` or `single`.

Note There are limitations to the tunability of elseif expressions. See "Limitations" on page 1-762 for more information.

Show else condition

If you select this check box, an `else` port is created. The If Action subsystem attached to the `else` port executes if the `if` port and all the `elseif` ports are false.

Enable zero-crossing detection

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

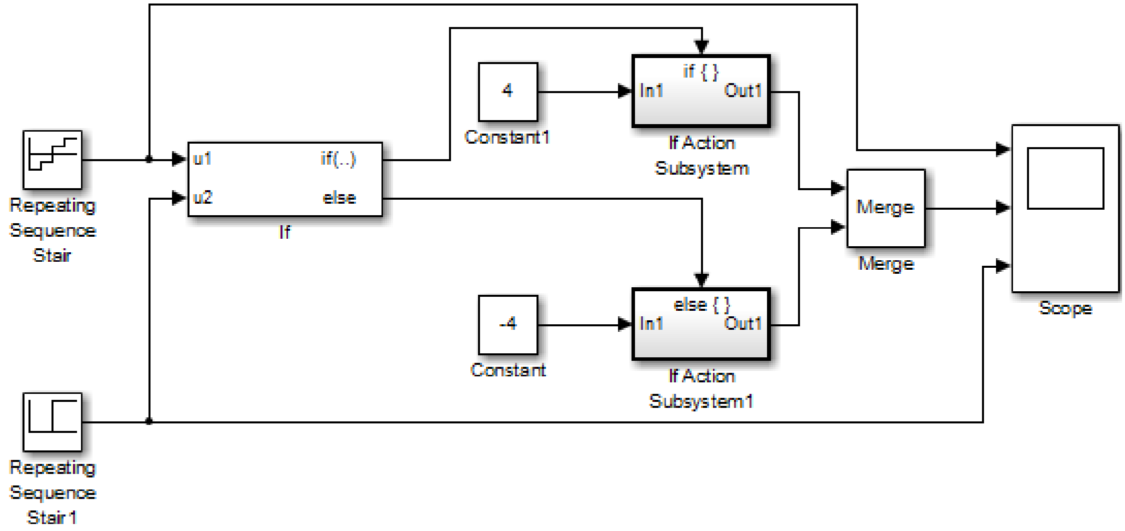
Sample time

Specify the sample time of the input signal. See “Specify Sample Time” in the online documentation for more information.

Examples

The If block does not directly support fixed-point data types. However, you can use the Compare To Constant block to work around this limitation.

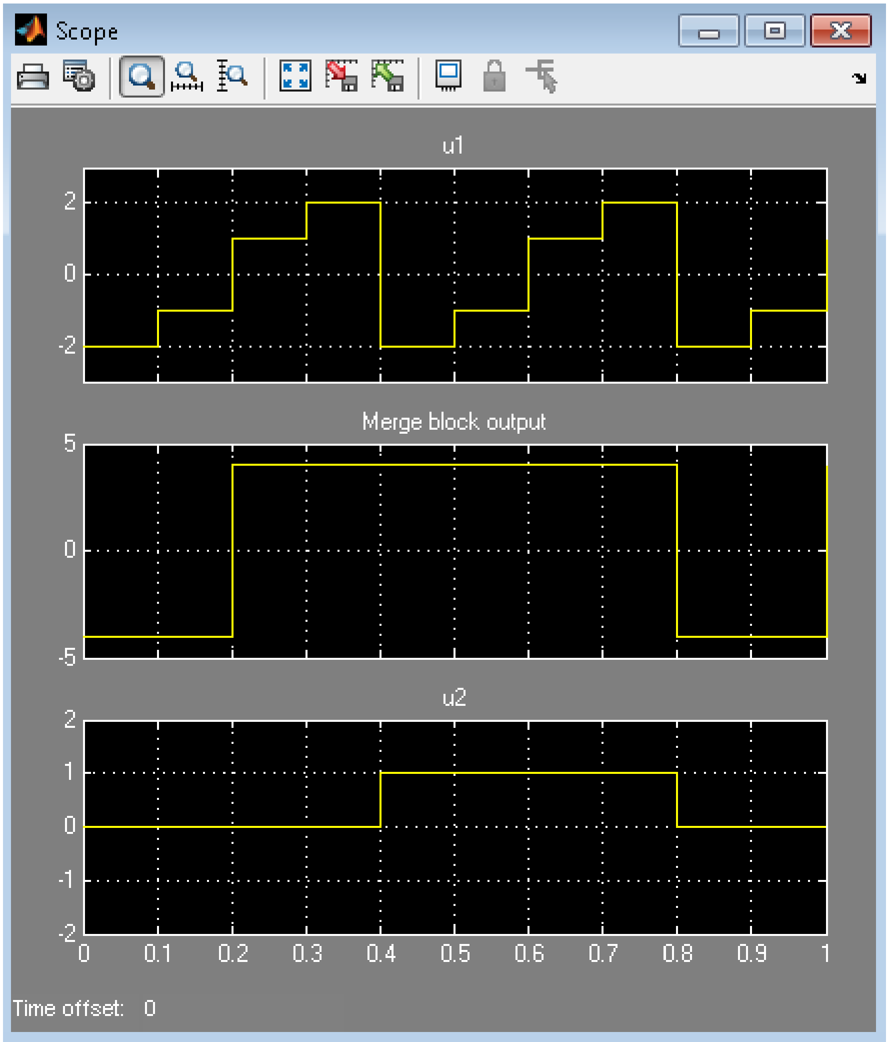
For example, consider the following floating-point model:


Repeating Sequence Stair
u1
u2
if(..)
else
If
4
Constant1
if { }
In1
Out1
If Action Subsystem
Merge
Merge
Scope
-4
Constant
else { }
In1
Out1
If Action Subsystem1
Repeating Sequence Stair1

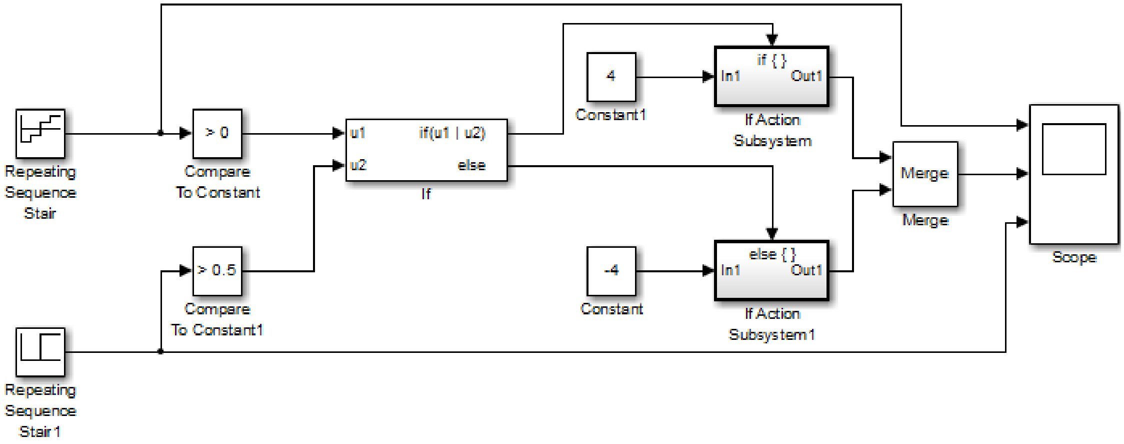
In this model, the If Action subsystems use their default configurations. The block and simulation parameters for the model are set to their default values except as follows:

Block or Dialog Box	**Parameter**	**Setting**
Configuration Parameters — **Solver** pane	**Start time**	`0.0`
	Stop time	`1.0`
	Type	`Fixed-step`
	Solver	`discrete (no continuous states)`
	Fixed-step size	`0.1`
Repeating Sequence Stair	**Vector of output values**	`[-2 -1 1 2].'`
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, if input `u1` is greater than 0 or input `u2` is greater than 0.5, the output is 4. Otherwise, the output is –4. The Scope block shows the output, `u1`, and `u2`:


Scope
u1
2
0
-2
Merge block output
5
0
-5
u2
2
1
0
-1
-2
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Time offset: 0

You can implement this block diagram as a model with fixed-point data types:



> 0
Compare To Constant
Repeating Sequence Stair
u1
if(u1 | u2)
u2
else
If
> 0.5
Compare To Constant1
Repeating Sequence Stair1
4
Constant1
if { }
In1
Out1
If Action Subsystem
-4
Constant
else { }
In1
Out1
If Action Subsystem1
Merge
Merge
Scope

The Repeating Sequence Stair blocks now output fixed-point data types.

The Compare To Constant blocks implement two parts of the **If expression** that is used in the If block in the floating-point version of the model, `(u1 > 0)` and `(u2 > 0.5)`. The `OR` operation, `(u1|u2)`, can still be implemented inside the If block. For a fixed-point model, the expression must be partially implemented outside of the If block as it is here.

The block and simulation parameters for the fixed-point model are the same as for the floating-point model with the following exceptions and additions:

Block	Parameter	Setting
Compare To Constant	**Operator**	`>`
	Constant value	`0`

Block	Parameter	Setting
	Output data type mode	`Boolean`
	Enable zero-crossing detection	off
Compare To Constant1	**Operator**	`>`
	Constant value	`0.5`
	Output data type mode	`Boolean`
	Enable zero-crossing detection	off
If	**Number of inputs**	`2`
	If expression	`u1|u2`

Characteristics

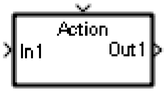
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled

If Action Subsystem

Purpose Represent subsystem whose execution is triggered by If block

Library Ports & Subsystems

Description



The If Action Subsystem block is a Subsystem block that is preconfigured to serve as a starting point for creating a subsystem whose execution is triggered by an If block.

Note All blocks in an If Action Subsystem must run at the same rate as the driving If block. You can achieve this by setting each block's sample time parameter to be either inherited (`-1`) or the same value as the If block's sample time.

For more information, see "Action Subsystem, If Action Subsystem, Switch Case Action Subsystem", If block and Modeling with Control Flow Blocks in the "Creating a Model" chapter of the Simulink documentation.

Purpose

Increase real world value of signal by one

Library

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.

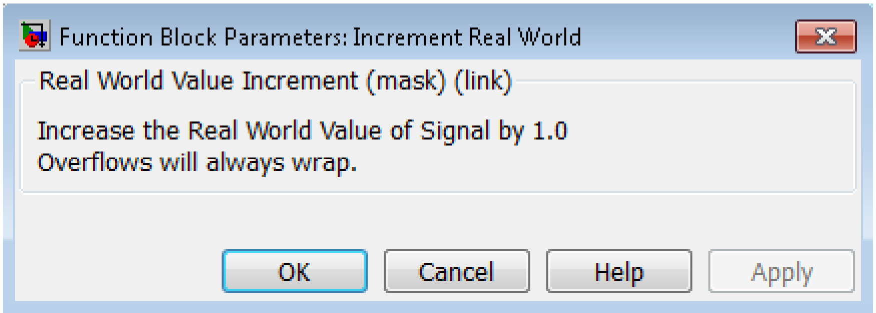
Data Type Support

The Increment Real World block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Real World Value Increment (mask) (link)

Increase the Real World Value of Signal by 1.0
Overflows will always wrap.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	No

Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Decrement Real World, Increment Stored Integer

Purpose

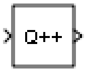
Increase stored integer value of signal by one

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Increment Stored Integer block increases the stored integer value of a signal by one.



Floating-point signals also increase by one, and overflows always wrap.

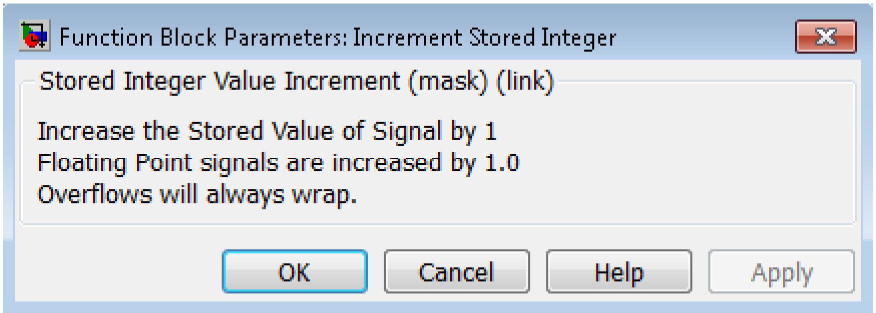
Data Type Support

The Increment Stored Integer block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box


Function Block Parameters: Increment Stored Integer

Stored Integer Value Increment (mask) (link)

Increase the Stored Value of Signal by 1
Floating Point signals are increased by 1.0
Overflows will always wrap.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	No
Multidimensionalized	No
Zero-Crossing Detection	No

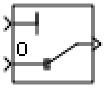
See Also

Decrement Stored Integer, Increment Real World

Purpose Switch output between different inputs based on value of first input

Library Signal Routing

Description The Index Vector block is an implementation of the Multiport Switch block. See Multiport Switch for more information.



Inport

Purpose

Create input port for subsystem or external input

Library

Ports & Subsystems, Sources

Description

Inport blocks are the links from outside a system into the system.

Simulink software assigns Inport block port numbers according to these rules:

- It automatically numbers the Inport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Inport block, the label is the next available number.
- If you delete an Inport block, other port numbers are automatically renumbered to ensure that the Inport blocks are in sequence and that no numbers are omitted.
- If you copy an Inport block into a system, its port number is *not* renumbered unless its current number conflicts with an Inport block already in the system. If the copied Inport block port number is not in sequence, renumber the block. Otherwise, you get an error message when you run the simulation or update the block diagram.

You can specify the dimensions of the input to the Inport block using the **Port dimensions** parameter. Entering a value of `-1` lets Simulink determine the port dimension.

The **Sample time** parameter is the rate at which the signal is coming into the system. A value of `-1` causes the block to inherit its sample time from the block driving it. You might need to set this parameter for:

- Inport blocks in a top-level system.
- Models with blocks where Simulink cannot determine the sample time, but these blocks drive Inport blocks.

For more information, see “Specify Sample Time”.

Inport Blocks in a Top-Level System

Inport blocks in a top-level system have two uses:

- To supply external inputs from the workspace, use the Configuration Parameters dialog box (see "Techniques for Importing Signal Data") or the `ut` argument of the `sim` command (see `sim`) to specify the inputs. If no external outputs are supplied, then the default output is the ground value.
- To provide a means for perturbation of the model by the `linmod` and `trim` analysis functions, use Inport blocks to inject inputs into the system.

Inport Blocks in a Subsystem

Inport blocks in a subsystem represent inputs to the subsystem. A signal arriving at an input port on a Subsystem block flows out of the associated Inport block in that subsystem. The Inport block associated with an input port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the input port on the Subsystem block. For example, the Inport block whose **Port number** parameter is `1` gets its signal from the block connected to the topmost port on the Subsystem block.

If you renumber the **Port number** of an Inport block, the block becomes connected to a different input port, although the block continues to receive its signal from the same block outside the subsystem.

The Inport block name appears in the Subsystem icon as a port label. To suppress display of the label, select the Inport block and choose **Format > Hide Name**.

Inport blocks inside a subsystem support signal label propagation, but root-level Inport blocks do not.

You can use a subsystem inport to supply fixed-point data in a structure or any other format.

Creating Duplicate Inports

You can create any number of duplicates of an Inport block. The duplicates are graphical representations of the original intended to simplify block diagrams by eliminating unnecessary lines. The duplicate has the same port number, properties, and output as the original. Changing properties of a duplicate changes properties of the original and vice versa.

To create a duplicate of an Inport block:

1. In the block diagram, select the block that you want to duplicate.
2. In the Model Editor menu bar, select **Edit > Copy**.
3. In the block diagram, place your cursor where you want to place the duplicate.
4. Select **Edit > Paste Duplicate Inport**.

Connecting Buses to Root Level Inports

If you want a root level Inport of a model to produce a bus signal, you must set the **Data type** parameter to the name of a bus object that defines the bus that the Inport produces. For more information, see "Bus Objects".

Data Type Support

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

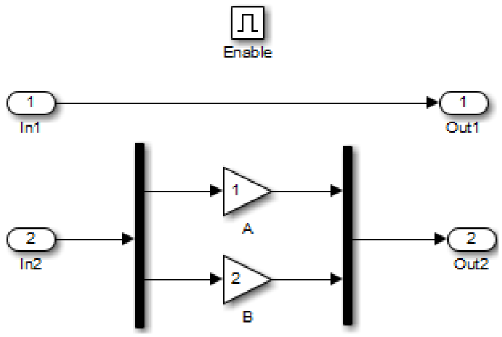
Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite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

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

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

The numeric and data types of the block output are the same as those of its input. You can specify the signal type, data type, and sampling mode of an external input to a root-level Inport block using the **Signal type**, **Data type**, and **Sampling mode** parameters.

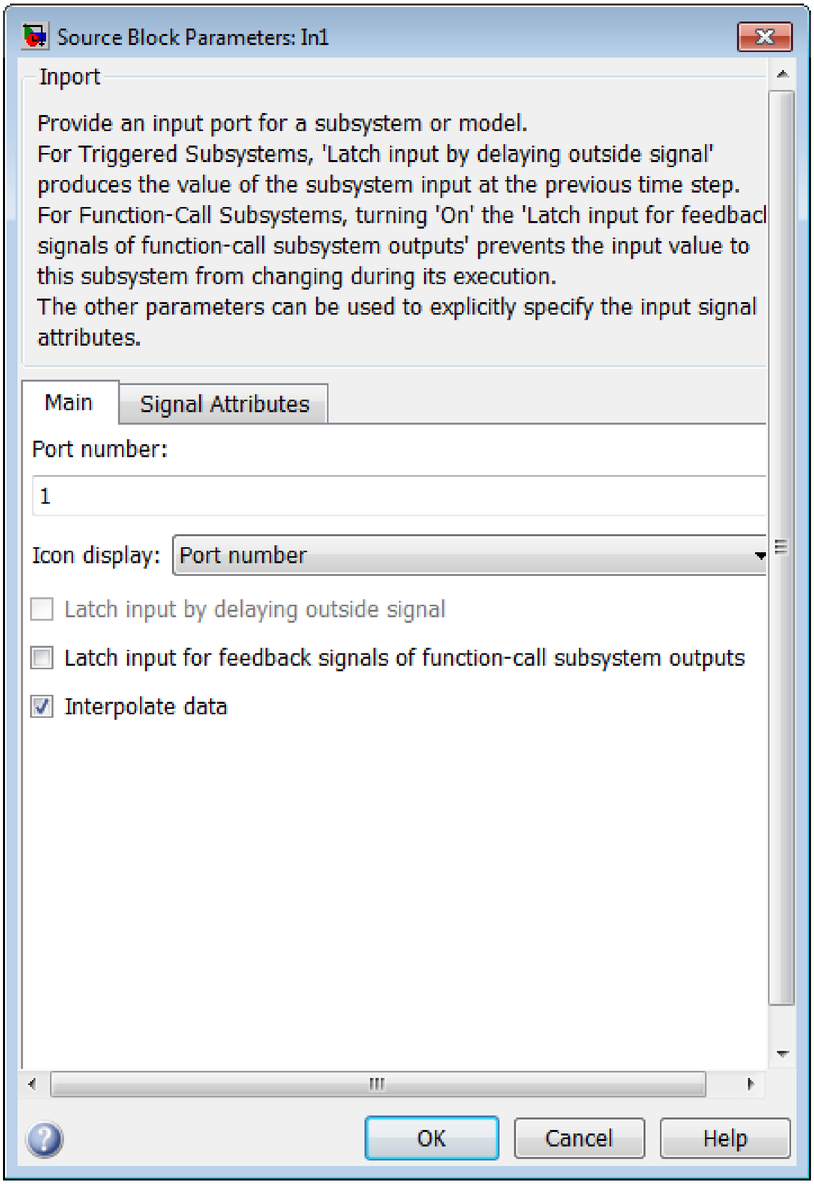
The elements of a signal array connected to a root-level Inport block must be of the same numeric and data types. Signal elements connected to a subsystem input port can be of differing numeric and data types, except in the following circumstance: If the subsystem contains an Enable, Trigger, or Atomic Subsystem block and the input port, or an element of the input port, connects directly to an output port, the input elements must be of the same type. For example, consider the following enabled subsystem:



In this example, the elements of a signal vector connected to `In1` must be of the same type. The elements connected to `In2`, however, can be of differing types.

Parameters and Dialog Box

The **Main** pane of the Inport block dialog box appears as follows:



Source Block Parameters: In1

Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedbacl signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

Port number: 1

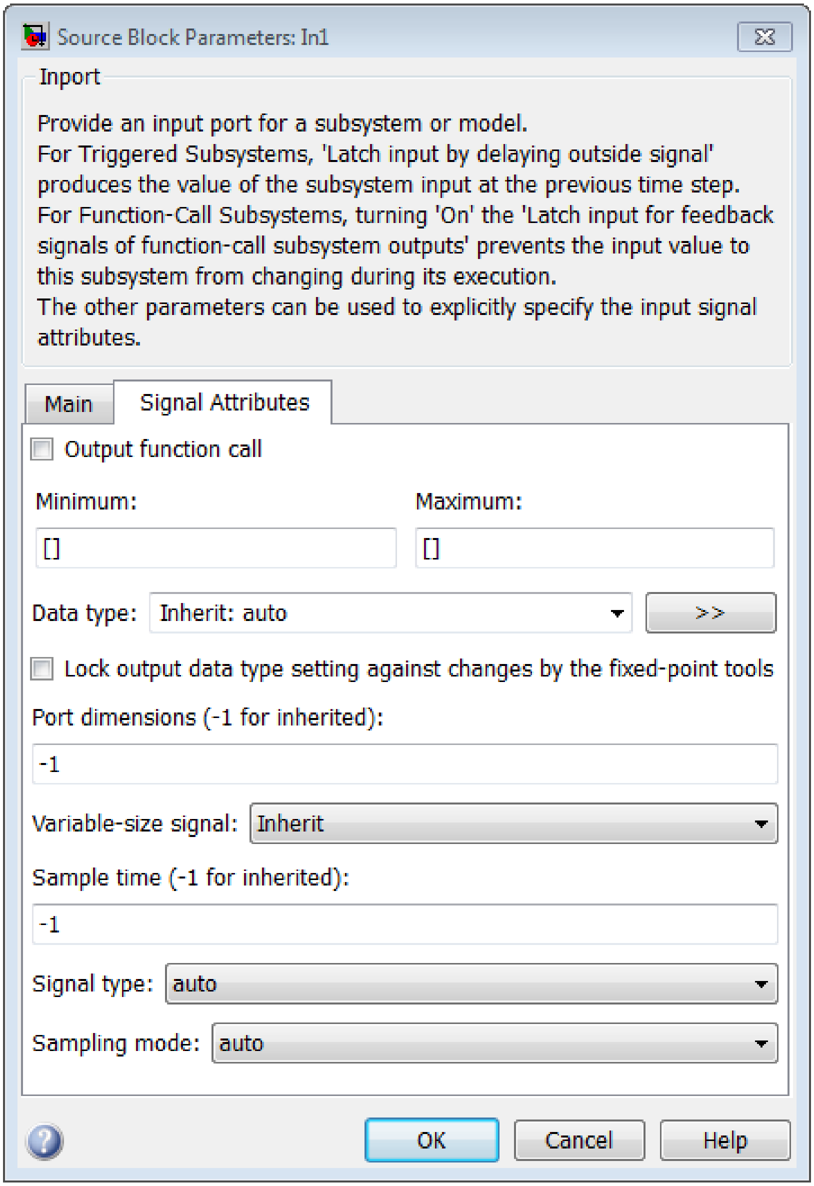
Icon display: Port number

☐ Latch input by delaying outside signal

☐ Latch input for feedback signals of function-call subsystem outputs

☑ Interpolate data

The **Signal Attributes** pane of the Inport block dialog box appears as follows:



Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

☐ Output function call

Minimum: []

Maximum: []

Data type: Inherit: auto

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

Port dimensions (-1 for inherited): -1

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Sampling mode: auto

- “Package” on page 1-1977
- “Icon display” on page 1-1086
- “Latch input by delaying outside signal” on page 1-788
- “Latch input for feedback signals of function-call subsystem outputs” on page 1-789
- “Interpolate data” on page 1-791
- “Output function call” on page 1-792
- “Minimum” on page 1-1920
- “Maximum” on page 1-1921
- “Data type” on page 1-795
- “Show data type assistant” on page 1-1672
- “Mode” on page 1-798
- “Data type override” on page 1-1926
- “Signedness” on page 1-1927
- “Word length” on page 1-1930
- “Scaling” on page 1-1736
- “Fraction length” on page 1-1931
- “Slope” on page 1-1932
- “Bias” on page 1-1932
- “Output as nonvirtual bus” on page 1-806
- “Port dimensions (-1 for inherited)” on page 1-807
- “Variable-size signal” on page 1-808
- “Sample time (-1 for inherited)” on page 1-1974
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1721

- “Signal type” on page 1-811
- “Sampling mode” on page 1-812

Port number

Specify the port number of the block.

Settings

Default: 1

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

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Icon display

Specify the information to be displayed on the icon of this input port.

Settings

Default: `Port number`

`Signal name`
: Display the name of the signal connected to this port (or signals if the input is a bus).

`Port number`
: Display port number of this port.

`Port number and signal name`
: Display both the port number and the names of the signals connected to this port.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Latch input by delaying outside signal

Output the value of the input signal at the previous time step.

Settings

Default: Off

☑ On

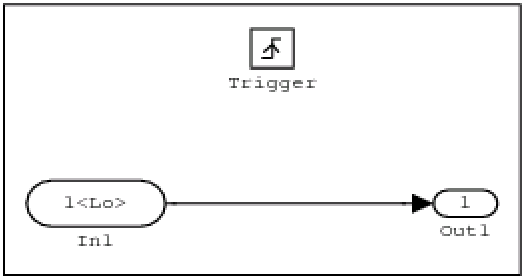
Output the value of the input signal at the previous time step.

☐ Off

Do not output the value of the input signal at the previous time step.

Tips

- This option applies only to triggered subsystems and is enabled only if the Inport block resides in a triggered subsystem.
- Selecting this check box enables Simulink to resolve data dependencies among triggered subsystems that are part of a loop.
- Type `sl_subsys_semantics` at the MATLAB prompt for examples using latched inputs with triggered subsystems.
- The Inport block indicates that this option is selected by displaying `<Lo>`.



Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Latch input for feedback signals of function-call subsystem outputs

Latch the value of the input to this subsystem and prevent this value from changing during the execution of the subsystem. For a single function call that is branched to invoke multiple function-call subsystems, this option allows you to break a loop formed by a signal fed back from one of these function-call subsystems into the other. A second functionality of this option is to prevent any change to the values of a feedback signal from a function-call subsystem that is invoked during the execution of this subsystem.

Settings

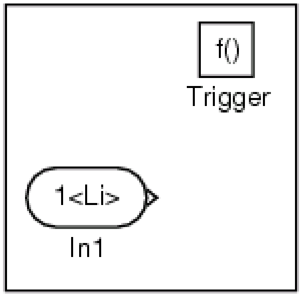
Default: Off

☑ On
Latch the input value.

☐ Off
Do not latch the input value.

Tips

- This parameter applies only to function-call subsystems and is enabled only if the Inport block resides in a function-call subsystem.
- This parameter ensures that the subsystem inputs, including those generated within the subsystem's context, do not change during execution of the subsystem.
- The Inport block indicates that this option is selected by displaying `<Li>`.


f()
Trigger
1<Li>
In1

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Interpolate data

When loading data from the workspace, cause the block to linearly interpolate and extrapolate output at time steps for which no corresponding data exists.

Settings

Default: On

☑ On

When loading data from the workspace, cause the block to linearly interpolate and extrapolate output at time steps for which no corresponding data exists.

☐ Off

When loading data from the workspace, do not cause the block to linearly interpolate or extrapolate output at time steps for which no corresponding data exists. Simulink uses the following interpolation and extrapolation:

- For time steps between the first specified data point and the last specified data point — zero-order hold
- For time steps before the first specified data point and after the last specified data point — ground value

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output function call

Specify that the input signal is outputting a function-call trigger signal.

Settings

Default: Off

☑ On

Input signal is the output of a function-call subsystem.

☐ Off

Input signal is not the output of a function-call subsystem.

Tips

- Select this option if it is necessary for a current model to accept a function-call trigger signal when referenced in the top model.
- This feature is limited to an asynchronous function call.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Minimum

Specify the minimum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

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

Simulink software uses this value to perform:

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Maximum

Specify the maximum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

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

Simulink software uses this value to perform:

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

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Data type

Specify the output data type of the external input.

Settings

Default: `Inherit:  auto`

`Inherit:  auto`
A rule that inherits a data type

`double`
Data type is `double`.

`single`
Data type is `single`.

`int8`
Data type is `int8`.

`uint8`
Data type is `uint8`.

`int16`
Data type is `int16`.

`uint16`
Data type is `uint16`.

`int32`
Data type is `int32`.

`uint32`
Data type is `uint32`.

`boolean`
Data type is `boolean`.

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

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

`Enum: <class name>`
Data type is enumerated, for example, `Enum: Basic Colors`.

`Bus: <object name>`
Data type is a bus object.

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

Do not specify a bus object as the expression.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`
: Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right.

`Built in`
: Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

  - `double` (default)
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`
  - `uint32`
  - `boolean`

`Fixed point`
: Fixed-point data types.

`Enumerated`
: Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

`Bus`
: Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or

change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details about the Bus Editor, see “Manage Bus Objects with the Bus Editor”.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`

`Binary point`
    Specify binary point location.

`Slope and bias`
    Enter slope and bias.

`Best precision`
    Specify best-precision values.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Output as nonvirtual bus

Output a nonvirtual bus.

Settings

Default: Off

☑ On

Output a nonvirtual bus.

☐ Off

Output a virtual bus.

Tips

- Select this option if you want code generated from this model to use a C structure to define the structure of the bus signal output by this block.
- All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. Therefore, if you select this option all signals in the bus must have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus, to allow the signal or bus to be included in a nonvirtual bus.

Dependency

Specify properties via bus object enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Port dimensions (-1 for inherited)

Specify the dimensions of the input signal to the block.

Settings

Default: `-1`

Valid values are:

`-1`	Dimensions are inherited from input signal
`n`	Vector signal of width `n` accepted
`[m n]`	Matrix signal having `m` rows and `n` columns accepted

Dependency

Clearing **Specify properties via bus object** enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Variable-size signal

Specify the type of signals allowed into this port.

Settings

Default: `Inherit`

`Inherit`
: Allow variable-size and fixed-size signals.

`No`
: Do not allow variable-size signals.

`Yes`
: Allow only variable-size signals.

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: string
Value: `'Inherit’| 'No' | 'Yes'`
Default: `'Inherit'`

Sample time (-1 for inherited)

Specify the time interval between samples.

Settings

Default: `-1`

To inherit the sample time, set this parameter to `-1`.

See "Specify Sample Time" in the online documentation for more information.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Signal type

Specify the numeric type of the external input.

Settings

Default: `auto`

`auto`
Accept either `real` or `complex` as the numeric type.

`real`
Specify the numeric type as a real number.

`complex`
Specify the numeric type as a complex number.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sampling mode

Specify whether the output signal is `Sample based` or `Frame based`.

Settings

Default: `auto`

`auto`
: Accept any sampling mode.

`Sample based`
: The output signal is sample-based.

`Frame based`
: The output signal is frame-based.

Dependency

Frame-based processing requires a DSP System Toolbox license.

For more information, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes
Multidimensionalized	Yes

Virtual	Yes, if the block does *not* reside in a conditionally-executed or atomic subsystem and does *not* connect directly to an Outport block For more information, see "Virtual Blocks" in the Simulink documentation.
Zero-Crossing Detection	No

See Also

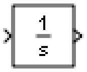
Outport

Asynchronous Task Specification

Purpose Integrate signal

Library Continuous

Description The Integrator block outputs the value of the integral of its input signal with respect to time.



The Integrator Limited block is identical to the Integrator block with the exception that the output of the block is limited based on the upper and lower saturation limits. See “Limiting the Integral” on page 1-816 for details.

Simulink treats the Integrator block as a dynamic system with one state. The block dynamics are given by:

$$\begin{cases} \dot{x}(t) = u(t) \\ y(t) = x(t) \end{cases} \quad x(t_0) = x_0$$

where:

- u is the block input.
- y is the block output.
- x is the block state.
- x_0 is the initial condition of x.

While these equations define an exact relationship in continuous time, Simulink uses numerical approximation methods to evaluate them with finite precision. Simulink can use a number of different numerical integration methods to compute the Integrator block’s output, each with advantages in particular applications. Use the **Solver** pane of the Configuration Parameters dialog box (see “Solver Pane”) to select the technique best suited to your application.

The selected solver computes the output of the Integrator block at the current time step, using the current input value and the value of the state at the previous time step. To support this computational model, the Integrator block saves its output at the current time step for use by the solver to compute its output at the next time step. The block also provides the solver with an initial condition for use in computing the block's initial state at the beginning of a simulation. The default value of the initial condition is 0. Use the block parameter dialog box to specify another value for the initial condition or create an initial value input port on the block.

Use the parameter dialog box to:

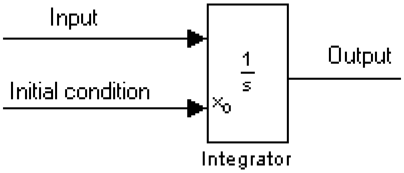
- Define upper and lower limits on the integral
- Create an input that resets the block's output (state) to its initial value, depending on how the input changes
- Create an optional state output so that the value of the block's output can trigger a block reset

Use the Discrete-Time Integrator block to create a purely discrete system.

Defining Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as `internal` and enter the value in the **Initial condition** field.
- To provide the initial conditions from an external source, specify the **Initial condition source** parameter as `external`. An additional input port appears under the block input.



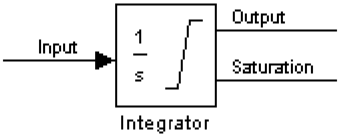
Note If the integrator limits its output (see "Limiting the Integral" on page 1-816), the initial condition must fall inside the integrator's saturation limits. If the initial condition is outside the block saturation limits, the block displays an error message.

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.

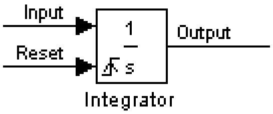
- –1 indicates that the lower limit is being applied.

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`.

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.



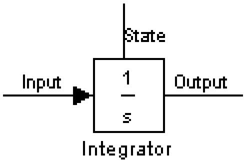
- Select `rising` to reset the state when the reset signal rises from a zero to a positive value or from a negative to a positive value.
- Select `falling` to reset the state when the reset signal falls from a positive value to zero or from a positive to a negative value.
- Select `either` to reset the state when the reset signal changes from a zero to a nonzero value 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.



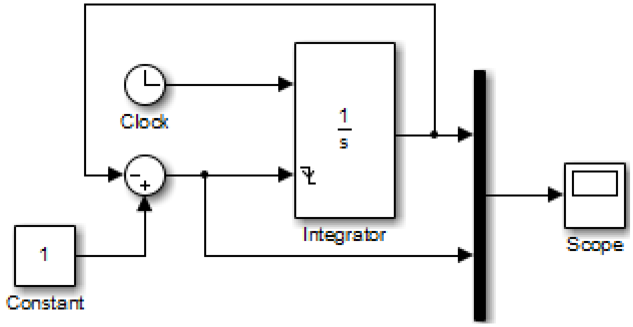
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. 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 1-819)
- Handing off a state from one enabled subsystem to another (see "Handing Off States Between Enabled Subsystems" on page 1-821)

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.

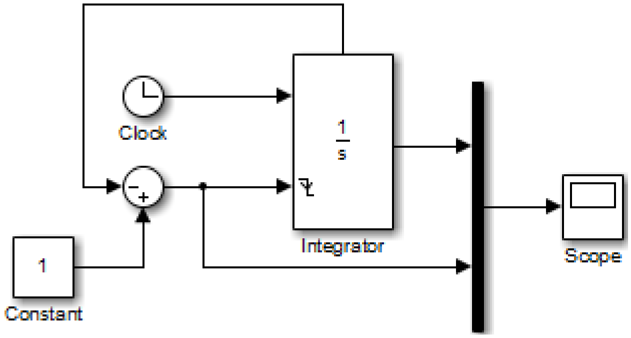
Creating Self-Resetting Integrators

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.

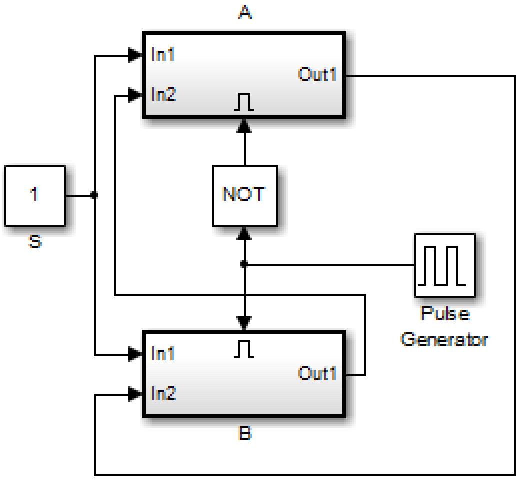
The following model uses the integrator's state port to avoid the algebraic loop.



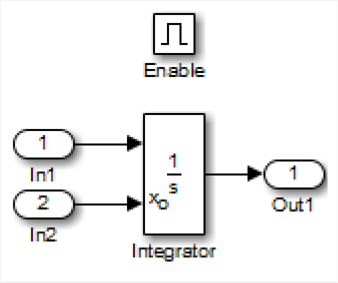
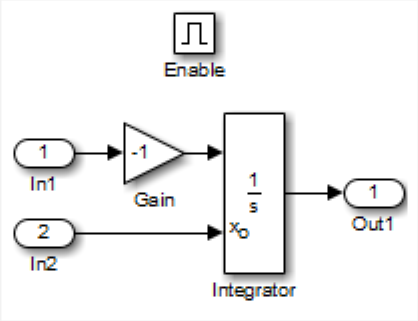
In this version, the value of the reset signal depends on the value of the state port. The value of the state port is available earlier in the current time step than the value of the integrator block's output port. Therefore, Simulink can determine whether the block needs to be reset before computing 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.


A
In1
In2
Out1
1
S
NOT
Pulse Generator
In1
In2
Out1
B

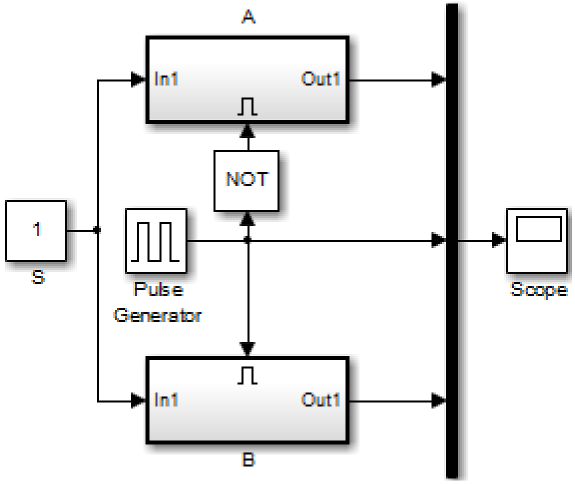
The enabled subsystems, A and B, contain the following blocks:

Subsystem A	Subsystem B
 Enable 1 In1 2 In2 1/s xo Integrator 1 Out1	 Enable 1 In1 -1 Gain 2 In2 1/s xo Integrator 1 Out1

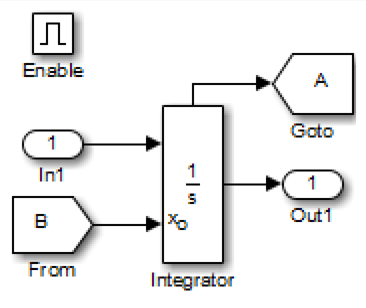
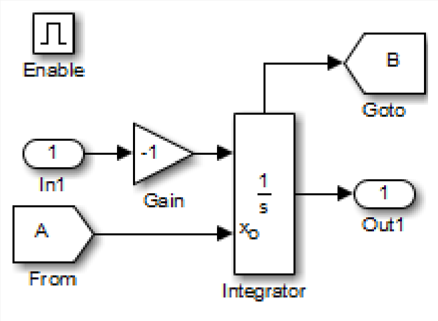
In this model, a constant input signal drives two enabled subsystems that integrate the signal. A pulse generator generates an enabling signal that causes execution to alternate between the two subsystems. The enable port of each subsystem is set to reset, which causes the subsystem to reset its integrator 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 A Goto 1 In1 1/s 1 Out1 B From x0 Integrator	 Enable B Goto 1 In1 -1 Gain 1/s 1 Out1 A From x0 Integrator

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.

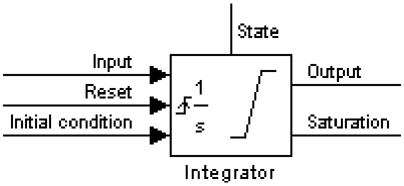
Note Simulink does not permit three or more enabled subsystems to hand off a model state. If Simulink detects that a model is handing off a state among more than two enabled subsystems, it generates an error.

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 “Specifying 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.

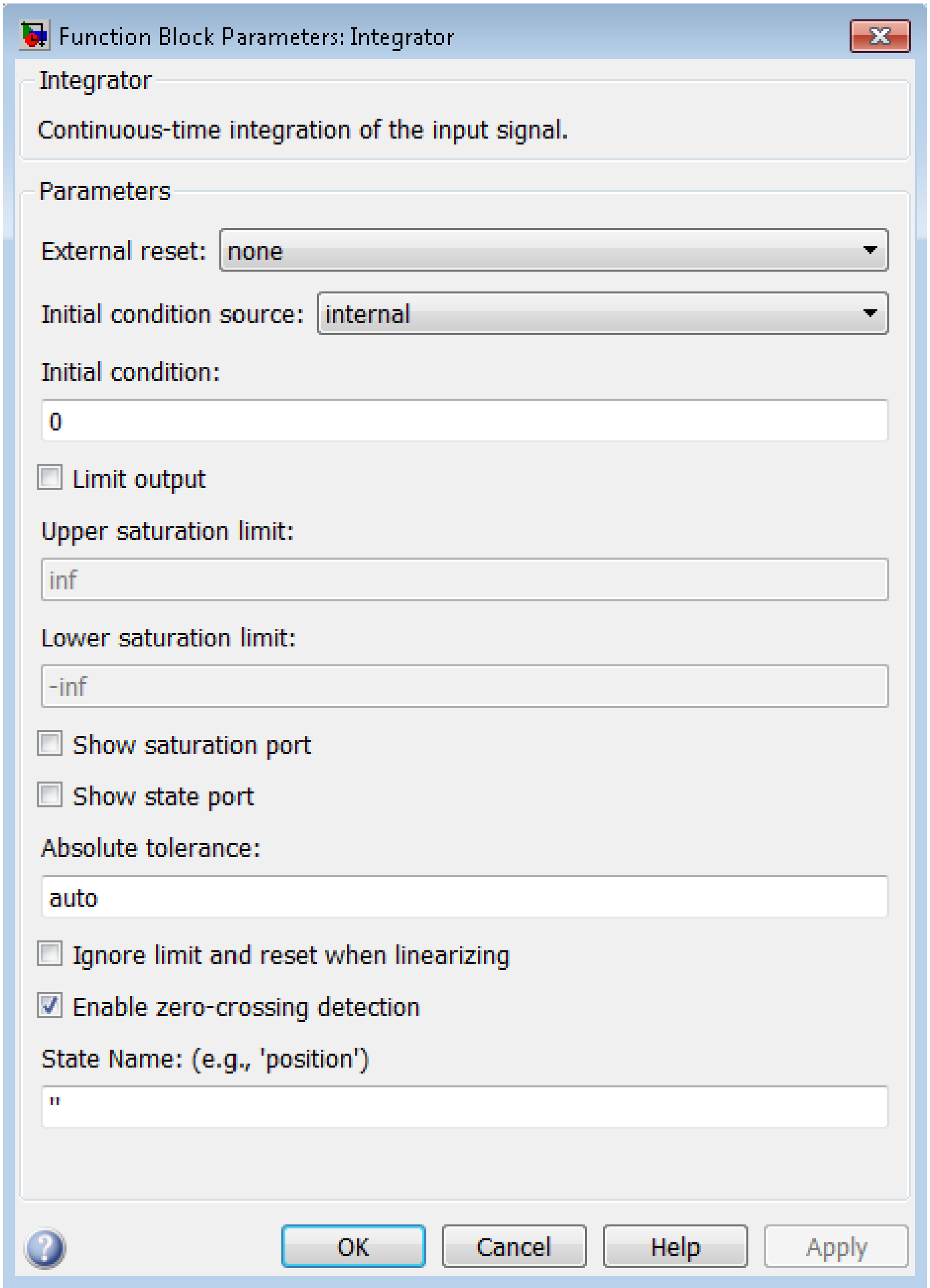
Selecting All Options

When you select all options, the block icon looks like this.


State
Input
Reset
Initial condition
1/s
Output
Saturation
Integrator

Data Type Support

The Integrator block accepts and outputs signals of type `double` on its data ports. The external reset port accepts signals of type `double` or `Boolean`.



Function Block Parameters: Integrator

Integrator

Continuous-time integration of the input signal.

Parameters

External reset: none

Initial condition source: internal

Initial condition: 0

☐ Limit output

Upper saturation limit: inf

Lower saturation limit: -inf

☐ Show saturation port

☐ Show state port

Absolute tolerance: auto

☐ Ignore limit and reset when linearizing

☑ Enable zero-crossing detection

State Name: (e.g., 'position') ''

Parameters and Dialog Box

External reset

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

Settings

Default: `none`

`none`
: Do not reset the state to initial conditions.

`rising`
: Reset the state when the reset signal rises from a zero to a positive value or from a negative to a positive value.

`falling`
: Reset the state when the reset signal falls from a positive value to zero or from a positive to a negative value.

`either`
: Reset the state when the reset signal changes from a zero to a nonzero value or changes sign.

`level`
: Reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.

`level hold`
: Reset the state when the reset signal is nonzero at the current time step.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Initial condition source

Get the initial conditions of the states.

Settings

Default: `internal`

`internal`
: Get the initial conditions of the states from the **Initial condition** parameter.

`external`
: Get the initial conditions of the states from an external block.

Tips

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Initial condition

Specify the states' initial conditions.

Settings

Default: `0`

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Limit output

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

Settings

Default: Off

☑ On
: Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

☐ Off
: Do not limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

Dependencies

This parameter enables **Upper saturation limit**.

This parameter enables **Lower saturation limit**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Upper saturation limit

Specify the upper limit for the integral.

Settings

Default: `inf`

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Dependencies

Limit output enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Lower saturation limit

Specify the lower limit for the integral.

Settings

Default: `-inf`

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Dependencies

Limit output enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Show saturation port

Add a saturation output port to the block.

Settings

Default: Off

☑ On
Add a saturation output port to the block.

☐ Off
Do not add a saturation output port to the block.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Show state port

Add an output port to the block for the block's state.

Settings

Default: Off

☑ On
Add an output port to the block for the block's state.

☐ Off
Do not add an output port to the block for the block's state.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, `1`, a real scalar, or a real 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Ignore limit and reset when linearizing

Cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the block's reset and output limitation options.

Settings

Default: Off

☑ On

Cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the block's reset and output limitation options.

☐ Off

Do not cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the block's reset and output limitation options.

Tip

Use this check box to linearize a model around an operating point that causes the integrator to reset or saturate.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Settings

Default: On

☑ On

Use zero crossings to detect and take a time step at any of the following events: reset, entering or leaving an upper saturation state, entering or leaving a lower saturation state.

☐ Off

Do not use zero crossings to detect and take a time step at any of the following events: reset, entering or leaving an upper saturation state, entering or leaving a lower saturation state.

If you select this check box, **Limit output**, and zero-crossing detection for the model as a whole, the Integrator block uses zero crossings as described.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

State Name (e.g., 'position')

Assign a unique name to each state.

Settings

Default: `'  '`

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 string, cell array, or structure.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Examples

The following example models show how to use the Integrator block:

- `sldemo_hardstop`
- `sldemo_suspn`

- `sldemo_wheelspeed_absbrake`

Characteristics

Direct Feedthrough	Yes, of the reset and external initial condition source ports
Sample Time	Continuous
Scalar Expansion	Yes, of parameters
States	Inherited from driving block or parameter
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	Yes, if enabled and you select the **Limit output** check box, one for detecting reset, one each to detect upper and lower saturation limits, and one when leaving saturation

See Also

Discrete-Time Integrator

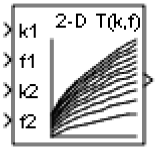
Interpolation Using Prelookup

Purpose Use precalculated index and fraction values to accelerate approximation of N-dimensional function

Library Lookup Tables

Description

How This Block Works with a Prelookup Block



The Interpolation Using Prelookup block works best 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. You feed the resulting index and fraction values into an Interpolation Using Prelookup block to interpolate an *n*-dimensional table. This combination of blocks performs the same operation that a single instance of the n-D Lookup Table block performs. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility that can provide more efficient simulation and code generation. For more information, see “Efficiency of Performance” in the Simulink documentation.

Supported Block Operations

To use the Interpolation Using Prelookup block, you specify a set of table data values directly on the dialog box or feed values into the `T` input port. Typically, these table values correspond to the breakpoint data sets specified in Prelookup blocks. The Interpolation Using Prelookup block generates output by looking up or estimating table values based on index and interval fraction values fed from Prelookup blocks. Labels for the index and interval fraction appear as `k` and `f` on the Interpolation Using Prelookup block icon.

When inputs for index and interval fraction...	The Interpolation Using Prelookup block...
Map to values in breakpoint data sets	Outputs the table value at the intersection of the row, column, and higher dimension breakpoints
Do not map to values in breakpoint data sets, but are within range	Interpolates appropriate table values, using the **Interpolation method** you select
Do not map to values in breakpoint data sets, and are out of range	Extrapolates the output value, using the **Extrapolation method** you select

How The Block Interpolates a Subset of Table Data

You can use the **Number of sub-table selection dimensions** parameter to specify that interpolation occur only on a subset of the table data. To activate this interpolation mode, set this parameter to a positive integer. This value defines the number of dimensions to select, starting from the highest dimension of table data. Therefore, the value must be less than or equal to the **Number of table dimensions**.

Suppose that you have 3-D table data in your Interpolation Using Prelookup block. The following behavior applies.

Number of Selection Dimensions	Action by the Block	Block Appearance
0	Interpolates the entire table and does not activate subtable selection	Does not change
1	Interpolates the first two dimensions and	Displays an input port with the label `sel1` that

Number of Selection Dimensions	Action by the Block	Block Appearance
	selects the third dimension	you use to select and interpolate 2-D tables
2	Interpolates the first dimension and selects the second and third dimensions	Displays two input ports with the labels `sel1` and `sel2` that you use to select and interpolate 1-D tables

Subtable selection uses zero-based indexing. For an example of interpolating a subset of table data, type `sldemo_bpcheck` at the MATLAB command prompt.

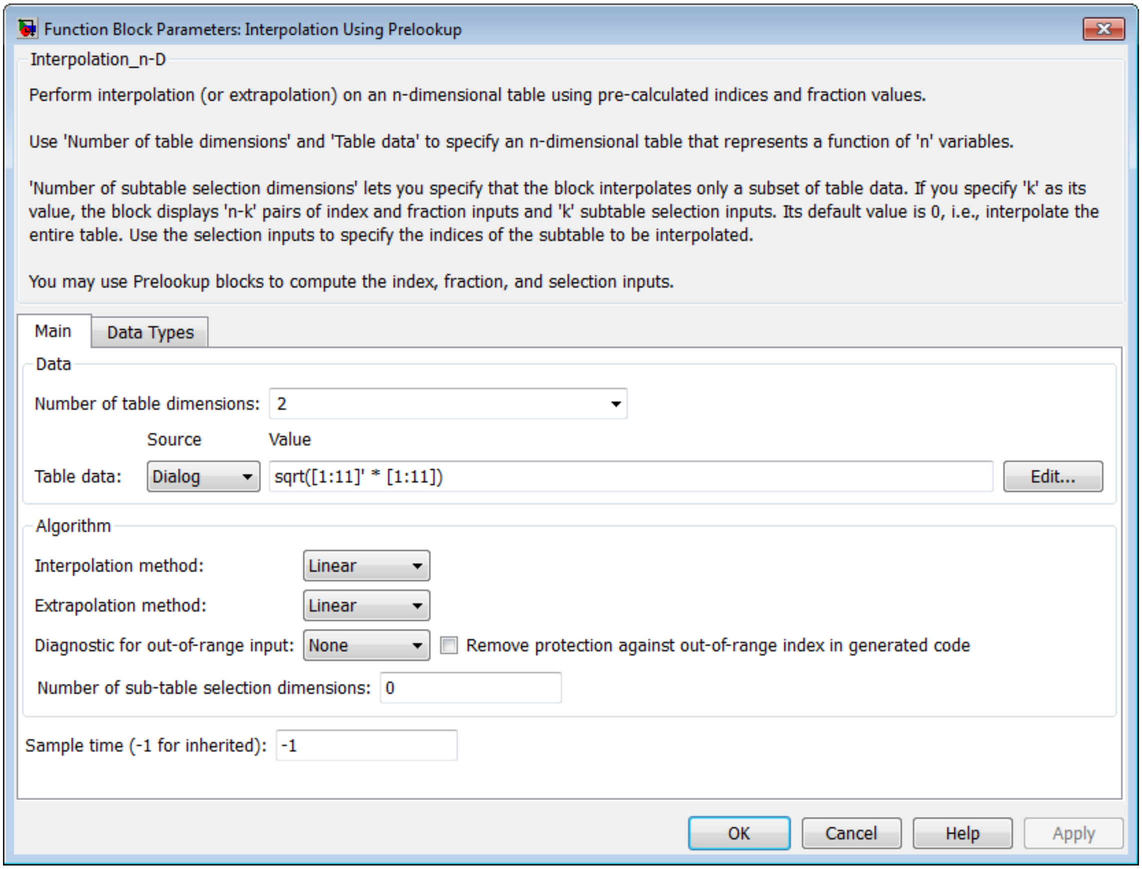
Data Type Support

The Interpolation Using Prelookup block accepts real signals of any numeric data type supported by Simulink software, except `Boolean`. The Interpolation Using Prelookup block supports fixed-point data types for signals, table data, and intermediate results.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Interpolation Using Prelookup block dialog box appears as follows:



Function Block Parameters: Interpolation Using Prelookup

Interpolation_n-D

Perform interpolation (or extrapolation) on an n-dimensional table using pre-calculated indices and fraction values.

Use 'Number of table dimensions' and 'Table data' to specify an n-dimensional table that represents a function of 'n' variables.

'Number of subtable selection dimensions' lets you specify that the block interpolates only a subset of table data. If you specify 'k' as its value, the block displays 'n-k' pairs of index and fraction inputs and 'k' subtable selection inputs. Its default value is 0, i.e., interpolate the entire table. Use the selection inputs to specify the indices of the subtable to be interpolated.

You may use Prelookup blocks to compute the index, fraction, and selection inputs.

Data

Number of table dimensions: 2

Table data: Source: Dialog; Value: sqrt([1:11]' * [1:11])

Algorithm

Interpolation method: Linear

Extrapolation method: Linear

Diagnostic for out-of-range input: None

☐ Remove protection against out-of-range index in generated code

Number of sub-table selection dimensions: 0

Sample time (-1 for inherited): -1

Number of table dimensions

Specify the number of dimensions that the table data must have. This value defines the number of independent variables for the table. Enter an integer between 1 and 30 into this field.

Table data

Specify whether to enter table data directly on the dialog box or to inherit the data from an input port.

- If you set **Source** to `Dialog`, enter table data in the edit field under **Value**. The size of the table data must match the **Number of table dimensions**. For this option, you specify table attributes on the **Data Types** pane.
- If you set **Source** to `Input port`, verify that an upstream signal supplies table data to the `T` input port. The size of the table data must match the **Number of table dimensions**. For this option, your block inherits table attributes from the `T` input port.

During block diagram editing, you can enter an empty matrix (specified as `[]`) or an undefined workspace variable in the edit field under **Value**. Use this behavior to postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram. For information about how to construct multidimensional arrays in MATLAB, see "Multidimensional Arrays" in the MATLAB documentation.

Click the **Edit** button to open the Lookup Table Editor (see "Edit Existing LookupTables" in the Simulink documentation).

Interpolation method

Select `Flat` or `Linear`. See "Interpolation Methods" in the Simulink documentation for more information.

Extrapolation method

Select `Clip` or `Linear`. See "Extrapolation Methods" in the Simulink documentation for more information. The **Extrapolation method** parameter is visible only when you select `Linear` as the **Interpolation method** parameter.

The Interpolation Using Prelookup block does not support `Linear` extrapolation when the input or output signals specify integer or fixed-point data types.

Valid index input may reach last index

Specify how block inputs for index (`k`) and interval fraction (`f`) access the last elements of *n*-dimensional table data. Index values are zero-based.

Check Box	Block Behavior
Selected	Returns the value of the last element in a dimension of its table when: • `k` indexes the last table element in the corresponding dimension • `f` is `0`
Cleared	Returns the value of the last element in a dimension of its table when: • `k` indexes the next-to-last table element in the corresponding dimension • `f` is `1`

This check box is visible only when:

- **Interpolation method** is `Linear`.
- **Extrapolation method** is `Clip`.

Tip When you select **Valid index input may reach last index** for an Interpolation Using Prelookup block, you must also select **Use last breakpoint for input at or above upper limit** for all Prelookup blocks that feed it. This action allows the blocks to use the same indexing convention when accessing the last elements of their breakpoint and table data sets.

Diagnostic for out-of-range input

Specify whether to produce a warning or error when the input `k` or `f` is out of range. Options include:

- `None` — no warning or error
- `Warning` — display a warning in the MATLAB Command Window and continue the simulation
- `Error` — halt the simulation and display an error in the Simulation Diagnostics Viewer

Remove protection against out-of-range index in generated code

Specify whether or not to include code that checks for out-of-range index inputs.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range index inputs.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range index inputs.	For safety-critical applications

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 Verification and Validation > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see "Consult the Model Advisor" in the Simulink documentation.

This check box has no effect on generated code when one of the following is true:

- The Prelookup block feeds index values to the Interpolation Using Prelookup block.

  Because index values from the Prelookup block are always valid, no check code is necessary.

- The data type of the input `k` restricts the data to valid index values.

  For example, unsigned integer data types guarantee nonnegative index values. Therefore, unsigned input values of `k` do not require check code for negative values.

Number of sub-table selection dimensions

Specify the number of dimensions of the subtable that the block uses to compute the output. Follow these rules:

- To enable subtable selection, enter a positive integer.

  This integer must be less than or equal to the **Number of table dimensions**.

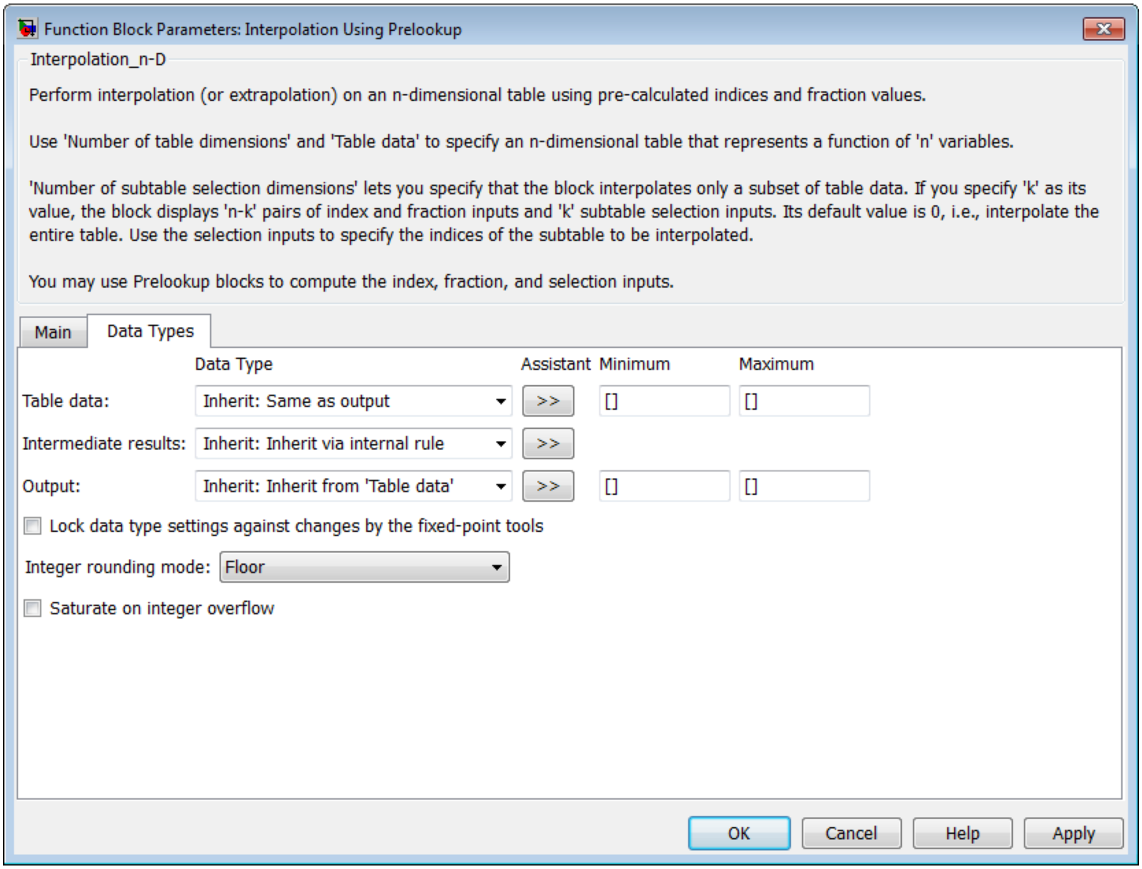
- To disable subtable selection, enter `0` to interpolate the entire table.

For more information, see "How The Block Interpolates a Subset of Table Data" on page 1-841.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” in the Simulink documentation for more information.

The **Data Types** pane of the Interpolation Using Prelookup block dialog box appears as follows:



Function Block Parameters: Interpolation Using Prelookup

Interpolation_n-D

Perform interpolation (or extrapolation) on an n-dimensional table using pre-calculated indices and fraction values.

Use 'Number of table dimensions' and 'Table data' to specify an n-dimensional table that represents a function of 'n' variables.

'Number of subtable selection dimensions' lets you specify that the block interpolates only a subset of table data. If you specify 'k' as its value, the block displays 'n-k' pairs of index and fraction inputs and 'k' subtable selection inputs. Its default value is 0, i.e., interpolate the entire table. Use the selection inputs to specify the indices of the subtable to be interpolated.

You may use Prelookup blocks to compute the index, fraction, and selection inputs.

	Data Type	Assistant	Minimum	Maximum
Table data:	Inherit: Same as output	>>	[]	[]
Intermediate results:	Inherit: Inherit via internal rule	>>		
Output:	Inherit: Inherit from 'Table data'	>>	[]	[]

☐ Lock data type settings against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate on integer overflow

Note The parameters for table attributes (data type, minimum, and maximum) are not available when you set **Source** to `Input port`. In this case, the block inherits all table attributes from the `T` input port.

Table data > Data Type

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the table data type.

Tip Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two Interpolation Using Prelookup blocks with different output data types
- Sharing of custom storage table data in Simulink Coder generated code for blocks with different output data types

---

Table data > Minimum

Specify the minimum value for table data. The default value is `[]` (unspecified).

Table data > Maximum

Specify the maximum value for table data. The default value is `[]` (unspecified).

Intermediate results > Data Type

Specify the intermediate results data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the intermediate results data type.

Tip Use this parameter to specify higher precision for internal computations than for table data or output data.

Output > Data Type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the output data type.

See “Specify Block Output Data Types” in the *Simulink User’s Guide* for more information.

Output > Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output > Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Locking the Output Data Type Setting”in the Fixed-Point Designer documentation.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” in the Fixed-Point Designer documentation.

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function in the mask field.

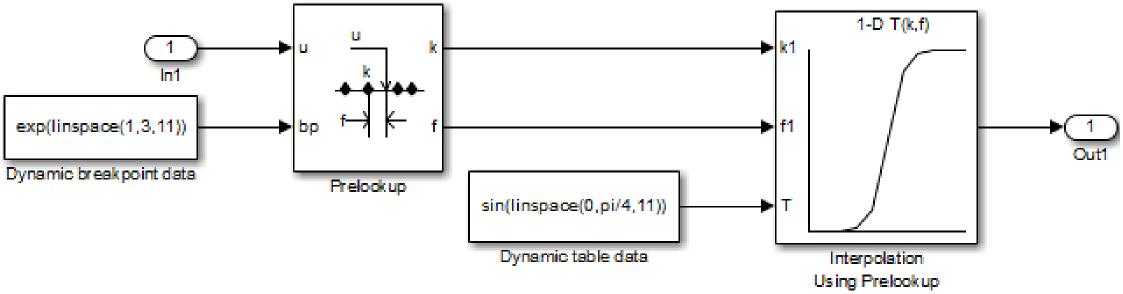
Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 “Checking 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.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

In the following model, a Constant block feeds the table data values to the `T` input port of the Interpolation Using Prelookup block.


1
In1
exp(linspace(1,3,11))
Dynamic breakpoint data
u
bp
u
k
f
k
f
Prelookup
sin(linspace(0,pi/4,11))
Dynamic table data
1-D T(k,f)
k1
f1
T
Interpolation
Using Prelookup
1
Out1

The Interpolation Using Prelookup block inherits the following table attributes from the `T` input port:

Table Attribute	Value
Minimum	`–Inf`
Maximum	`Inf`
Data type	`single`

Similarly, a Constant block feeds the breakpoint data set to the `bp` input port of the Prelookup block, which inherits the following breakpoint attributes:

Breakpoint Attribute	Value
Minimum	`–Inf`
Maximum	`Inf`
Data type	`single`

Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision, floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.

For other examples, see “Prelookup and Interpolation Blocks” in the Simulink documentation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also

Prelookup

Purpose

Apply MATLAB function or expression to input

Library

User-Defined Functions

Description

Interpreted
MATLAB Fcn

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 or an error occurs.

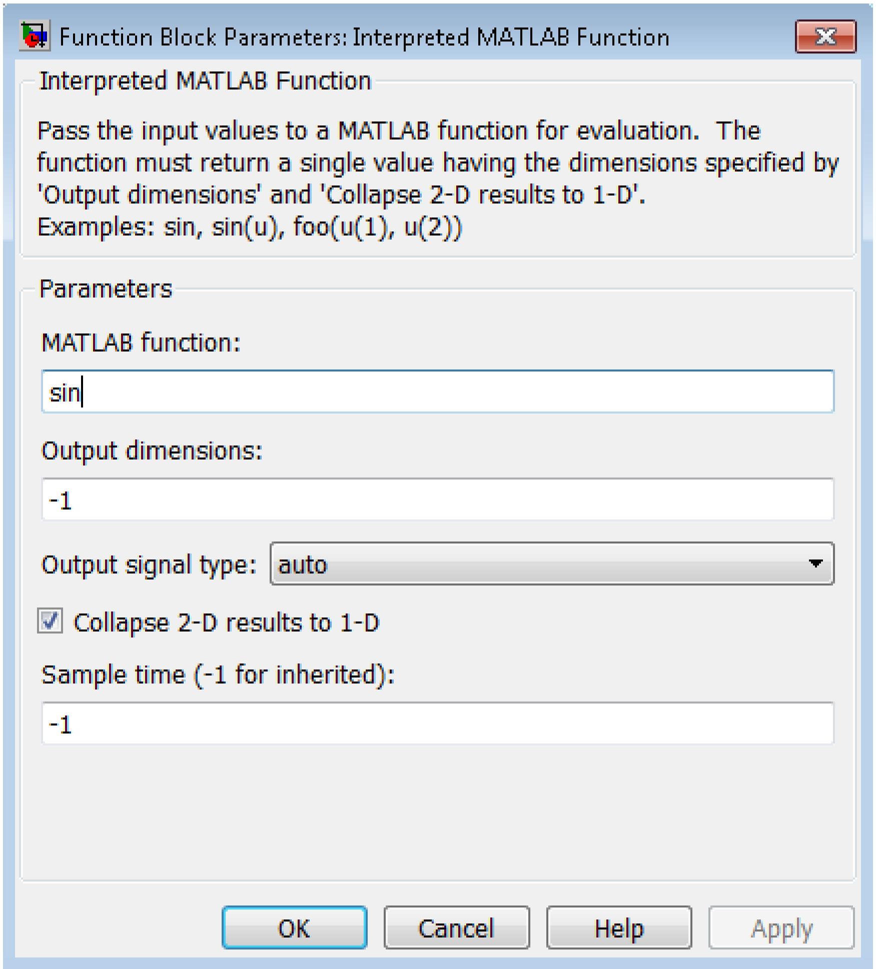
Some valid expressions for this block are:

```
sin
atan2(u(1), u(2))
u(1)^u(2)
```

Note This block is slower than the Fcn block because it calls the MATLAB parser during each integration step. Consider using built-in blocks (such as the Fcn block or the Math Function block) instead. Alternatively, you can write the function as a MATLAB S-function or MEX-file S-function, then access it using the S-Function block.

Data Type Support

The Interpreted MATLAB Function block accepts one real or complex input of type `double` and generates real or complex output of type `double`, depending on the setting of the **Output signal type** parameter.


Function Block Parameters: Interpreted MATLAB Function

Interpreted MATLAB Function

Pass the input values to a MATLAB function for evaluation. The function must return a single value having the dimensions specified by 'Output dimensions' and 'Collapse 2-D results to 1-D'.
Examples: sin, sin(u), foo(u(1), u(2))

Parameters

MATLAB function: sin

Output dimensions: -1

Output signal type: auto

☑ Collapse 2-D results to 1-D

Sample time (-1 for inherited): -1

OK
Cancel
Help
Apply

Parameters and Dialog Box

MATLAB function

Specify the function or expression. If you specify a function only, it is not necessary to include the input argument in parentheses.

Output dimensions

Specify the dimensions of the signal output by this block. If the output dimensions are to be the same as the dimensions of the input signal, specify –1. Otherwise, enter the dimensions of the output signal, for example, 2 for a two-element vector. In either case, the output dimensions must match the dimensions of the value returned by the function or expression in the **MATLAB function** field.

Output signal type

Specify the output signal type of the block as `real`, `complex`, or `auto`. A value of `auto` sets the output type to be the same as the type of the input signal.

Collapse 2-D results to 1-D

Select this check box to output a 2-D array as a 1-D array containing the 2-D array's elements in column-major order.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	N/A
Dimensionalized	Yes
Zero-Crossing Detection	No

Interval Test

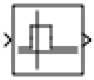
Purpose

Determine if signal is in specified interval

Library

Logic and Bit Operations

Description



The Interval Test block outputs TRUE if the input is between the values specified by the **Lower limit** and **Upper limit** parameters. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the **Lower limit** or the **Upper limit** is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are selected in the dialog box.

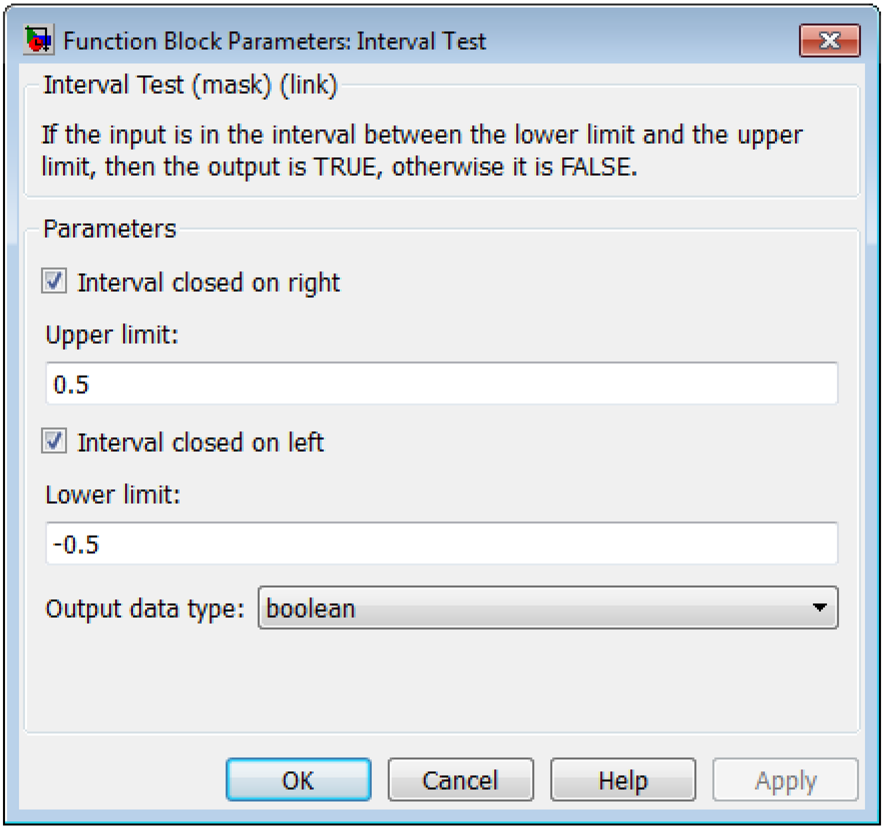
Data Type Support

The Interval Test block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

  In this case, the **Upper limit** and **Lower limit** values must be of the same enumerated type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Interval Test (mask) (link)

If the input is in the interval between the lower limit and the upper limit, then the output is TRUE, otherwise it is FALSE.

Parameters

☑ Interval closed on right

Upper limit: 0.5

☑ Interval closed on left

Lower limit: -0.5

Output data type: boolean

Parameters and Dialog Box

Interval closed on right

When you select this check box, the **Upper limit** is included in the interval for which the block outputs TRUE.

Upper limit

The upper limit of the interval for which the block outputs TRUE.

Interval closed on left

When you select this check box, the **Lower limit** is included in the interval for which the block outputs TRUE.

Lower limit

The lower limit of the interval for which the block outputs TRUE.

Output data type

Select the output data type: `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Interval Test Dynamic

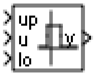
Purpose

Determine if signal is in specified interval

Library

Logic and Bit Operations

Description



The Interval Test Dynamic block outputs TRUE if the input is between the values of the external signals up and lo. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the signal up or the signal lo is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are selected in the dialog box.

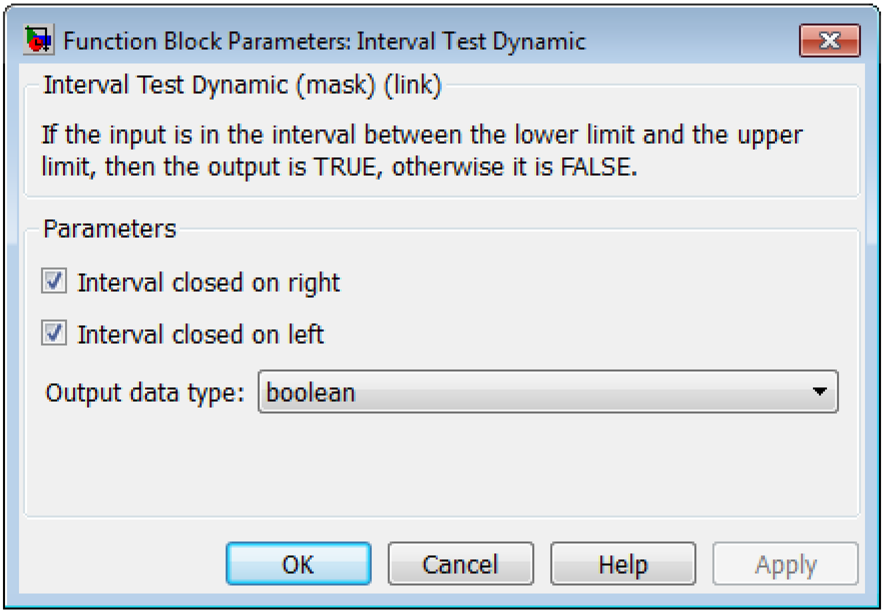
Data Type Support

The Interval Test Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

  In this case, all inputs must be of the same enumerated type.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Function Block Parameters: Interval Test Dynamic

Interval Test Dynamic (mask) (link)

If the input is in the interval between the lower limit and the upper limit, then the output is TRUE, otherwise it is FALSE.

Parameters

☑ Interval closed on right

☑ Interval closed on left

Output data type: boolean

Parameters and Dialog Box

Interval closed on right
When you select this check box, the value of the signal connected to the block's "up" input port is included in the interval for which the block outputs TRUE.

Interval closed on left
When you select this check box, the value of the signal connected to the block's "lo" input port is included in the interval for which the block outputs TRUE.

Output data type
Select the output data type: `boolean` or `uint8`.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also Interval Test

Level-2 MATLAB S-Function

Purpose

Use Level-2 MATLAB S-function in model

Library

User-Defined Functions

Description

matlabfile

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.

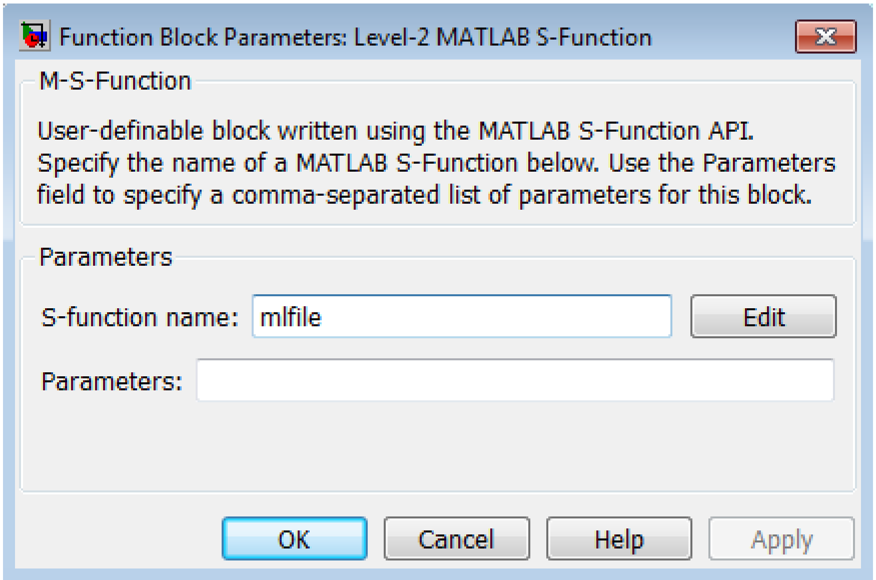
Note Use the S-Function block to include a Level-1 MATLAB S-function in a block.

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.

Data Type Support

Depends on the MATLAB file that defines the behavior of a particular instance of this block.



Function Block Parameters: Level-2 MATLAB S-Function

M-S-Function

User-definable block written using the MATLAB S-Function API. Specify the name of a MATLAB S-Function below. Use the Parameters field to specify a comma-separated list of parameters for this block.

Parameters

S-function name: mlfile

Parameters:

Parameters and Dialog Box

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).

Parameters

Specify values of the parameters of this block.

Characteristics

Direct Feedthrough	Depends on the MATLAB S-function
Sample Time	Depends on the MATLAB S-function
Scalar Expansion	Depends on the MATLAB S-function
Dimensionalized	Depends on the MATLAB S-function
Multidimensionalized	Yes
Zero Crossing	No

Logical Operator

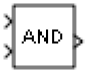
Purpose

Perform specified logical operation on input

Library

Logic and Bit Operations

Description

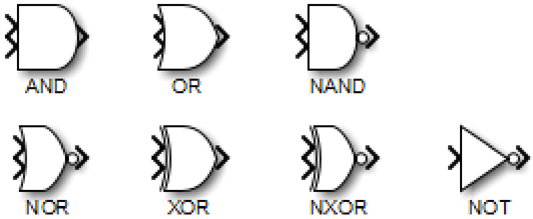


The Logical Operator block performs the specified logical operation on its inputs. An input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero.

You select the Boolean operation connecting the inputs with the **Operator** parameter list. If you select `rectangular` as the **Icon shape** property, the block updates to display the name of the selected operator. The supported operations are given below.

Operation	Description
AND	TRUE if all inputs are TRUE
OR	TRUE if at least one input is TRUE
NAND	TRUE if at least one input is FALSE
NOR	TRUE when no inputs are TRUE
XOR	TRUE if an odd number of inputs are TRUE
NXOR	TRUE if an even number of inputs are TRUE
NOT	TRUE if the input is FALSE

If you select `distinctive` as the **Icon shape**, the block's appearance indicates its function. Simulink software displays a distinctive shape for the selected operator, conforming to the IEEE Standard Graphic Symbols for Logic Functions:



The number of input ports is specified with the **Number of input ports** parameter. The output type is specified with the **Output data type** parameter. An output value is 1 if TRUE and 0 if FALSE.

Note The output data type should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers, and any floating-point data type.

The size of the output depends on input vector size and the selected operator:

- If the block has more than one input, any nonscalar inputs must have the same dimensions. For example, if any input is a 2-by-2 array, all other nonscalar inputs must also be 2-by-2 arrays.

  Scalar inputs are expanded to have the same dimensions as the nonscalar inputs.

  If the block has more than one input, the output has the same dimensions as the inputs (after scalar expansion) and each output element is the result of applying the specified logical operation to the corresponding input elements. For example, if the specified operation is AND and the inputs are 2-by-2 arrays, the output is a 2-by-2 array whose top left element is the result of applying AND to the top left elements of the inputs, etc.

- For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector. The output is always a scalar.
- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the logical complements of the input vector elements.

When configured as a multi-input XOR gate, this block performs an addition- modulo-two operation as mandated by the IEEE Standard for Logic Elements.

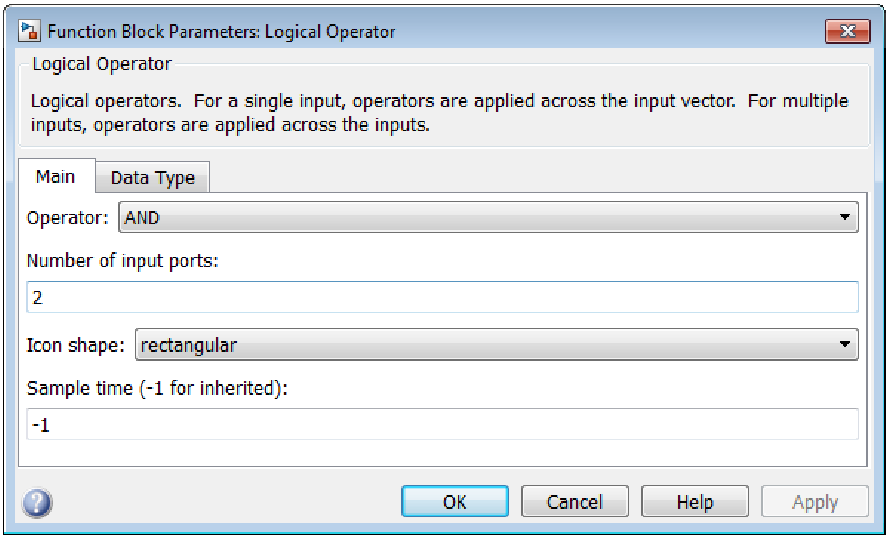
Data Type Support

The Logical Operator block accepts real signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Logical Operator block dialog box appears as follows:



Function Block Parameters: Logical Operator

Logical Operator

Logical operators. For a single input, operators are applied across the input vector. For multiple inputs, operators are applied across the inputs.

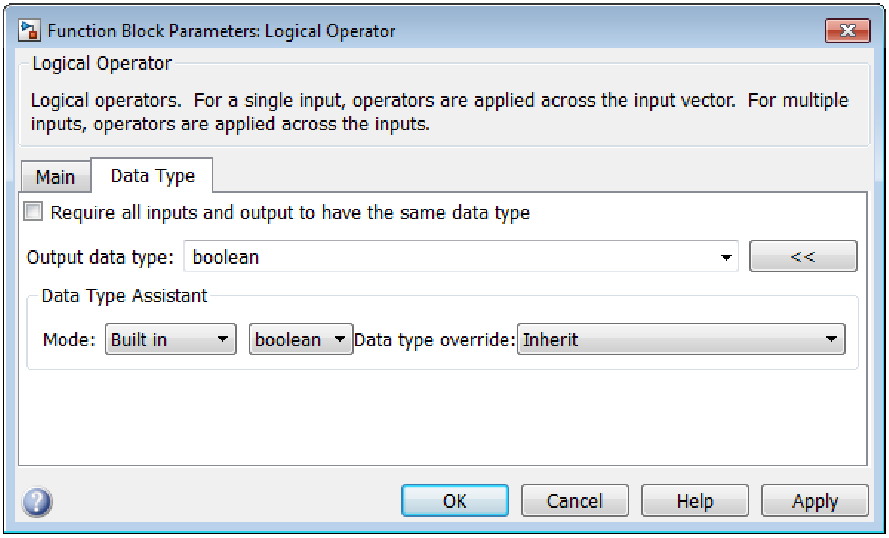
Operator: AND

Number of input ports: 2

Icon shape: rectangular

Sample time (-1 for inherited): -1

The **Data Type** pane of the Logical Operator block dialog box appears as follows:



Function Block Parameters: Logical Operator

Logical Operator

Logical operators. For a single input, operators are applied across the input vector. For multiple inputs, operators are applied across the inputs.

☐ Require all inputs and output to have the same data type

Output data type: boolean

Data Type Assistant

Mode: Built in — boolean — Data type override: Inherit

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Operator

Select logical operator to apply to block inputs.

Settings

Default: `AND`

`AND`
TRUE if all inputs are TRUE

`OR`
TRUE if at least one input is TRUE

`NAND`
TRUE if at least one input is FALSE

`NOR`
TRUE when no inputs are TRUE

`XOR`
TRUE if an odd number of inputs are TRUE

`NXOR`
TRUE if an even number of inputs are TRUE

`NOT`
TRUE if the input is FALSE

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Number of input ports

Specify number of block inputs.

Settings

Default: `2`

- The value must be appropriate for the selected operator.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Icon shape

Specify shape of the block icon.

Settings

Default: `rectangular`

`rectangular`
: Result in a rectangular block that displays the name of the selected operator.

`distinctive`
: Use the graphic symbol for the selected operator as specified by the IEEE standard.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in "Types of Sample Time".

See also "Specify Sample Time" in the online documentation for more information.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Require all inputs and output to have the same data type

Require all inputs and the output to have the same data type.

Settings

Default: Off

☑ On

Require all inputs and the output to have the same data type.

☐ Off

Do not require all inputs and the output to have the same data type.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `boolean`

`Inherit: Logical (see Configuration Parameters: Optimization)`

Uses the **Implement logic signals as Boolean data** configuration parameter (see "Implement logic signals as Boolean data (vs. double)") to specify the output data type.

Note This option supports models created before the `boolean` option was available. Use one of the other options, preferably `boolean`, for new models.

`boolean`

Specifies output data type is `boolean`.

`fixdt(1,16)`

Specifies output data type is `fixdt(1,16)`.

`<data type expression>`

Uses the name of a data type object, for example, `Simulink.NumericType`.

Tip To enter a built-in data type (`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`), enclose the expression in single quotes. For example, enter `'double'` instead of `double`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Built in`

`Inherit`
: Specifies inheritance rules for data types. Selecting `Inherit` enables `Logical (see Configuration Parameters: Optimization)`.

`Built in`
: Specifies built-in data types. Selecting `Built in` enables `boolean`.

`Fixed point`
: Specifies fixed-point data types.

`Expression`
: Specifies expressions that evaluate to data types.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Integer`

`Integer`

Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

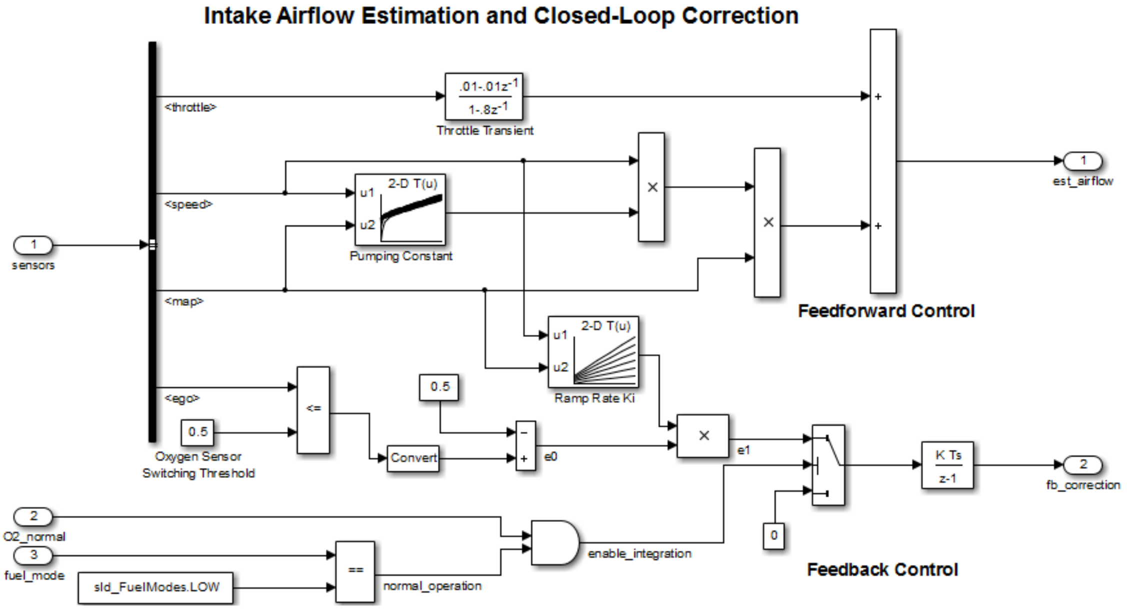
See Also

See “Specifying a Fixed-Point Data Type”.

Examples

Logical Operator Block: AND Operator

In the `sldemo_fuelsys` model, the `fuel_rate_control/airflow_calc` subsystem uses a Logical Operator block as an AND operator:

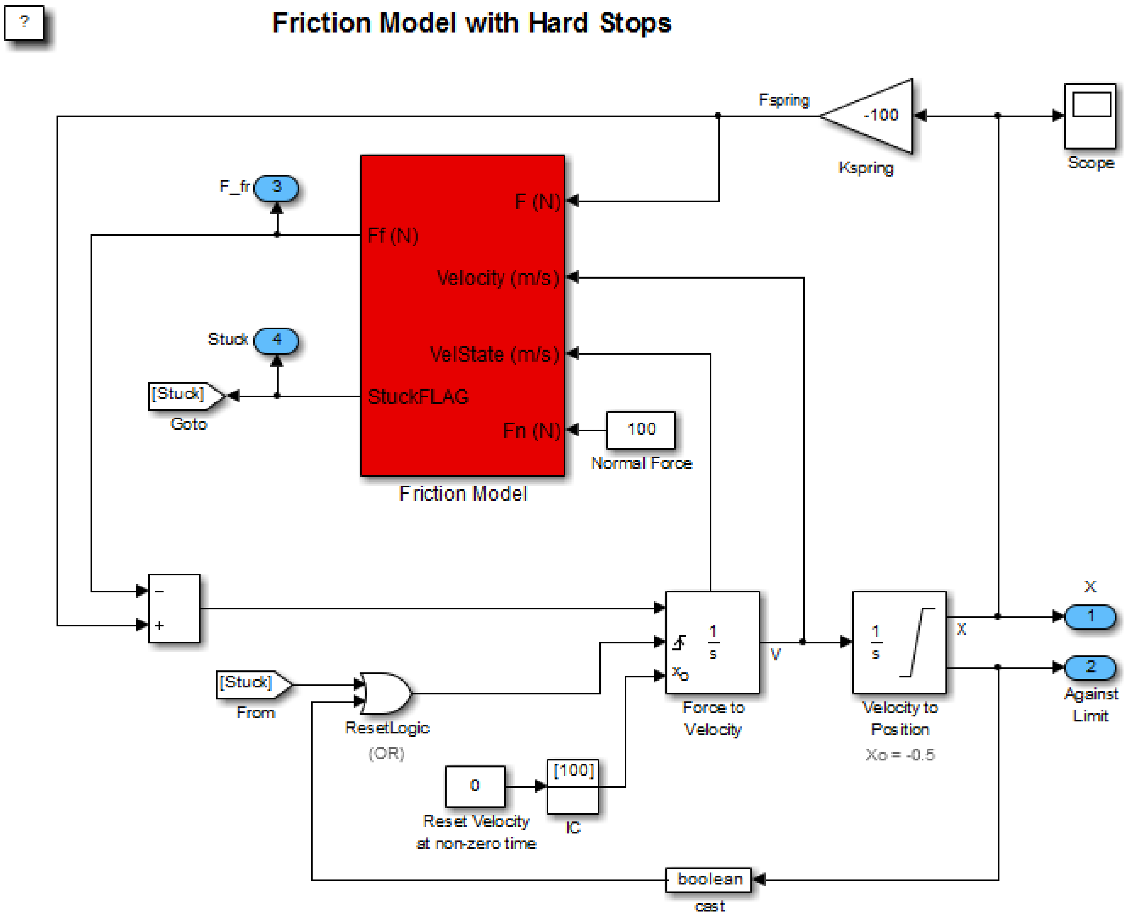

Intake Airflow Estimation and Closed-Loop Correction
1
sensors
<throttle>
<speed>
<map>
<ego>
.01-.01z-1
1-.8z-1
Throttle Transient
2-D T(u)
u1
u2
Pumping Constant
×
×
+
+
1
est_airflow
Feedforward Control
2-D T(u)
u1
u2
Ramp Rate Ki
<=
0.5
Oxygen Sensor
Switching Threshold
0.5
-
+
Convert
e0
×
e1
0
K Ts
z-1
2
fb_correction
Feedback Control
2
O2_normal
3
fuel_mode
sld_FuelModes.LOW
==
normal_operation
enable_integration

The output of the Logical Operator block (the `enable_integration` signal) feeds into the control port of a Switch block that activates feedback control.

When the Logical Operator block output is...	Feedback control...
1	Occurs
0	Does not occur

Logical Operator Block: OR Operator

In the `sldemo_hardstop` model, the Logical Operator block appears as an OR operator:

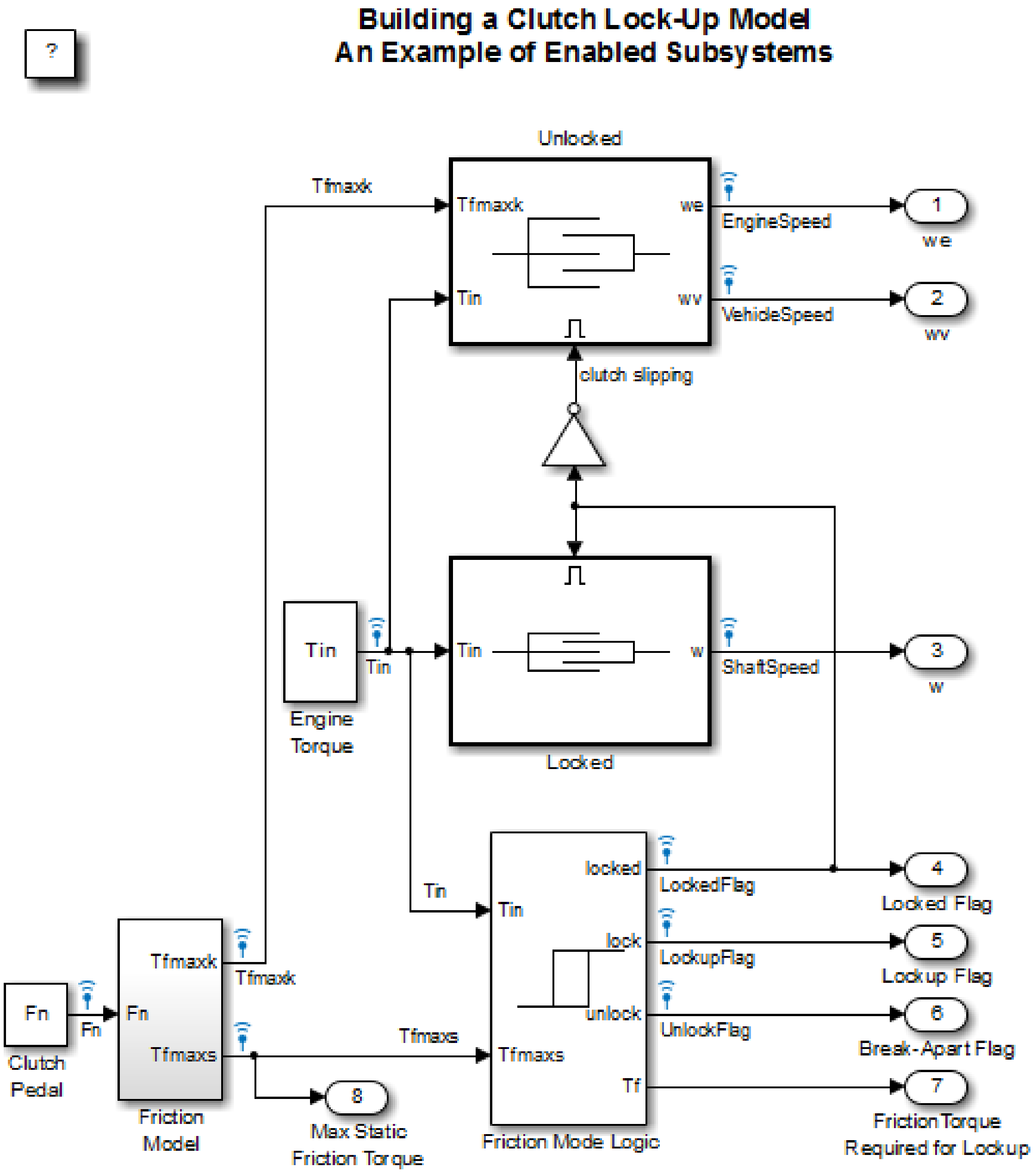

?
Friction Model with Hard Stops
Fspring
-100
Kspring
Scope
F_fr
3
F (N)
Ff (N)
Velocity (m/s)
Stuck
4
VelState (m/s)
[Stuck]
Goto
StuckFLAG
Fn (N)
100
Normal Force
Friction Model
-
+
1
s
x0
Force to Velocity
V
1
s
X
Velocity to Position
Xo = -0.5
X
1
2
Against Limit
[Stuck]
From
ResetLogic
(OR)
0
Reset Velocity at non-zero time
[100]
IC
boolean
cast

The output of the Logical Operator block feeds into the trigger port of an Integrator block to control whether velocity resets to the initial condition.

When the Logical Operator block output changes...	The Integrator block...
From 0 to 1	Resets the velocity
From 1 to 0	Does not reset velocity

Logical Operator Block: NOT Operator

In the `sldemo_clutch` model, the Logical Operator block appears as a NOT operator:


Building a Clutch Lock-Up Model
An Example of Enabled Subsystems
?
Unlocked
Tfmaxk
Tin
we
wv
EngineSpeed
VehicleSpeed
1
we
2
wv
clutch slipping
Tin
Engine Torque
Locked
w
ShaftSpeed
3
w
Friction Mode Logic
locked
lock
unlock
Tf
Tfmaxs
LockedFlag
LockupFlag
UnlockFlag
4
Locked Flag
5
Lockup Flag
6
Break-Apart Flag
7
Friction Torque Required for Lockup
Fn
Clutch Pedal
Friction Model
Tfmaxs
8
Max Static Friction Torque

The output of the Logical Operator block (the `clutch slipping` signal) feeds into the trigger port of an enabled subsystem.

When the Logical Operator block outputs...	The Unlocked subsystem is...
1	Enabled
0	Disabled

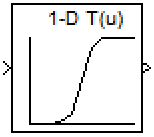
Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of inputs
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose Approximate one-dimensional function

Library Lookup Tables

Description The 1-D Lookup Table block is a one-dimensional version of the n-D Lookup Table block.



Purpose	Approximate two-dimensional function
Library	Lookup Tables
Description	The 2-D Lookup Table block is a two-dimensional version of the n-D Lookup Table block.

Purpose Approximate N-dimensional function

Library Lookup Tables

Description

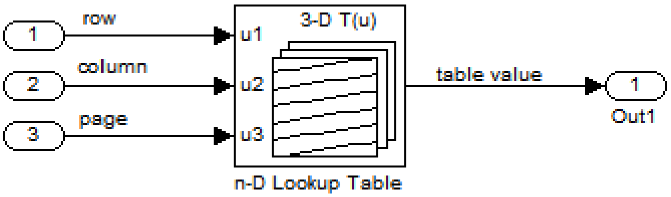
Supported Block Operations

The n-D Lookup Table block evaluates a sampled representation of a function in N variables

$$y = F(x_1, x_2, x_3, \ldots, x_N)$$

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, 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 “How to Rotate a Block” in the Simulink documentation for a description of the port order for various block orientations.

Specification of Breakpoint and Table Data

The following block parameters define the breakpoint and table data.

Block Parameter	Purpose
Number of table dimensions	Specifies the number of dimensions of your lookup table.
Breakpoints	Specifies a breakpoint vector that corresponds to each dimension of your lookup table.
Table data	Defines the associated set of output values.

Tip Evenly spaced breakpoints can make the generated code division-free. For more information, see `fixpt_evenspace_cleanup` in the Simulink documentation and "Identify questionable fixed-point operations" in the Simulink Coder documentation.

How the Block Generates Output

The n-D Lookup Table block generates output by looking up or estimating table values based on the input values:

When block inputs...	The n-D Lookup Table block...
Match the values of indices in breakpoint data sets	Outputs the table value at the intersection of the row, column, and higher dimension breakpoints
Do not match the values of indices in breakpoint data sets, but are within range	Interpolates appropriate table values, using the **Interpolation method** you select
Do not match the values of indices in breakpoint data sets, and are out of range	Extrapolates the output value, using the **Extrapolation method** you select

Other Blocks That Perform Equivalent Operations

You can use the Interpolation Using Prelookup block with the Prelookup block to perform the equivalent operation of one n-D Lookup Table block. This combination of blocks offers greater flexibility that can result in more efficient simulation performance for linear interpolations.

When the lookup operation is an array access that does not require interpolation, use the Direct Lookup Table (n-D) block. For example, if you have an integer value `k` and you want the `k`th element of a table, `y = table(k)`, interpolation is unnecessary.

Data Type Support

The n-D Lookup Table block supports all numeric data types that Simulink supports, including fixed-point data types. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

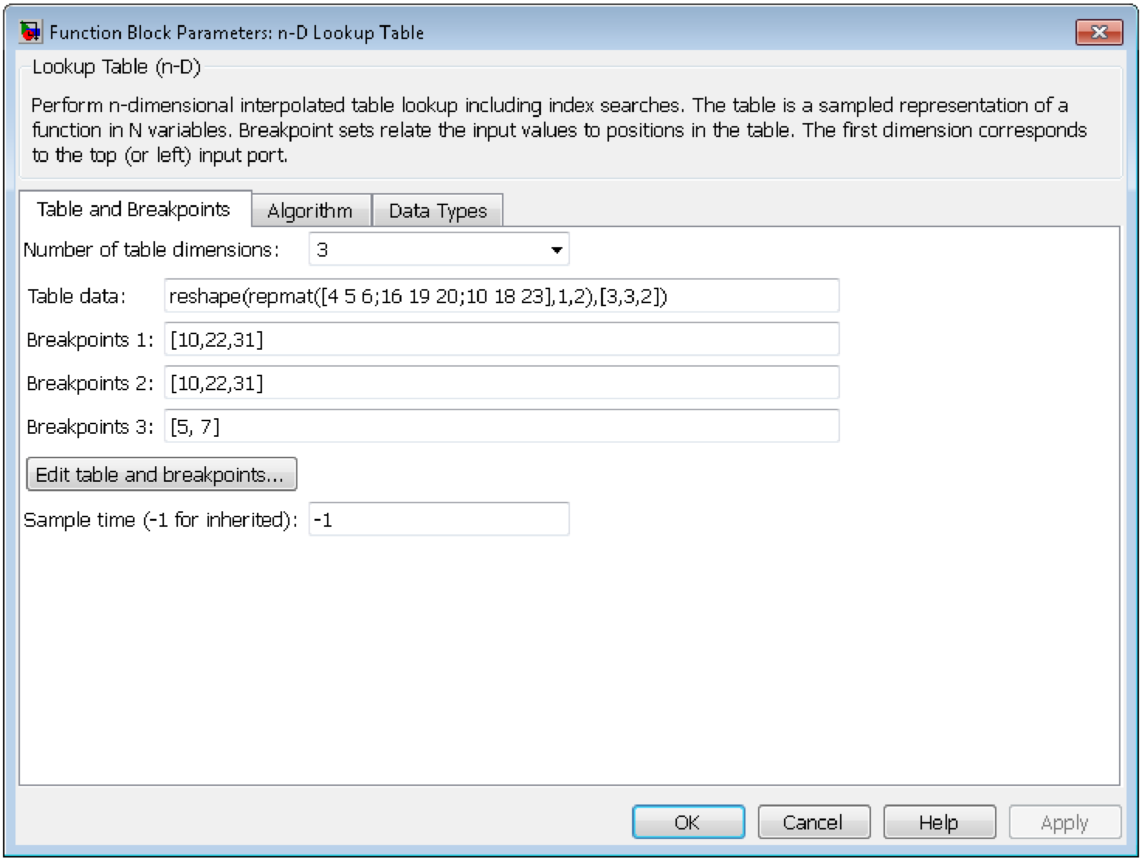
For cubic spline interpolation and linear extrapolation modes, the following parameters must use the same floating-point type:

- Table data
- Breakpoints
- Fraction
- Intermediate results
- Output

Inputs for indexing must be real, but table data can be complex.

Parameters and Dialog Box

The **Table and Breakpoints** pane of the n-D Lookup Table block dialog box appears as follows:


Function Block Parameters: n-D Lookup Table
Table and Breakpoints
Algorithm
Data Types
Edit table and breakpoints...
OK
Cancel
Help
Apply

Lookup Table (n-D)

Perform n-dimensional interpolated table lookup including index searches. The table is a sampled representation of a function in N variables. Breakpoint sets relate the input values to positions in the table. The first dimension corresponds to the top (or left) input port.

Number of table dimensions: 3

Table data: reshape(repmat([4 5 6;16 19 20;10 18 23],1,2),[3,3,2])

Breakpoints 1: [10,22,31]

Breakpoints 2: [10,22,31]

Breakpoints 3: [5, 7]

Sample time (-1 for inherited): -1

Number of table dimensions

Enter the number of dimensions of the lookup table by specifying an integer from 1 to 30. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

Table data

Enter the table of output values.

During simulation, the matrix size must match the dimensions defined by the **Number of table dimensions** parameter. However, during block diagram editing, you can enter an empty matrix (specified as `[]`) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram. For information about how to construct multidimensional arrays in MATLAB, see “Multidimensional Arrays” in the MATLAB online documentation.

Breakpoints

In each **Breakpoints** row, enter the breakpoint set that corresponds to each dimension of table data. For each dimension, specify breakpoints as a `1-by-n` or `n-by-1` vector whose values are strictly monotonically increasing.

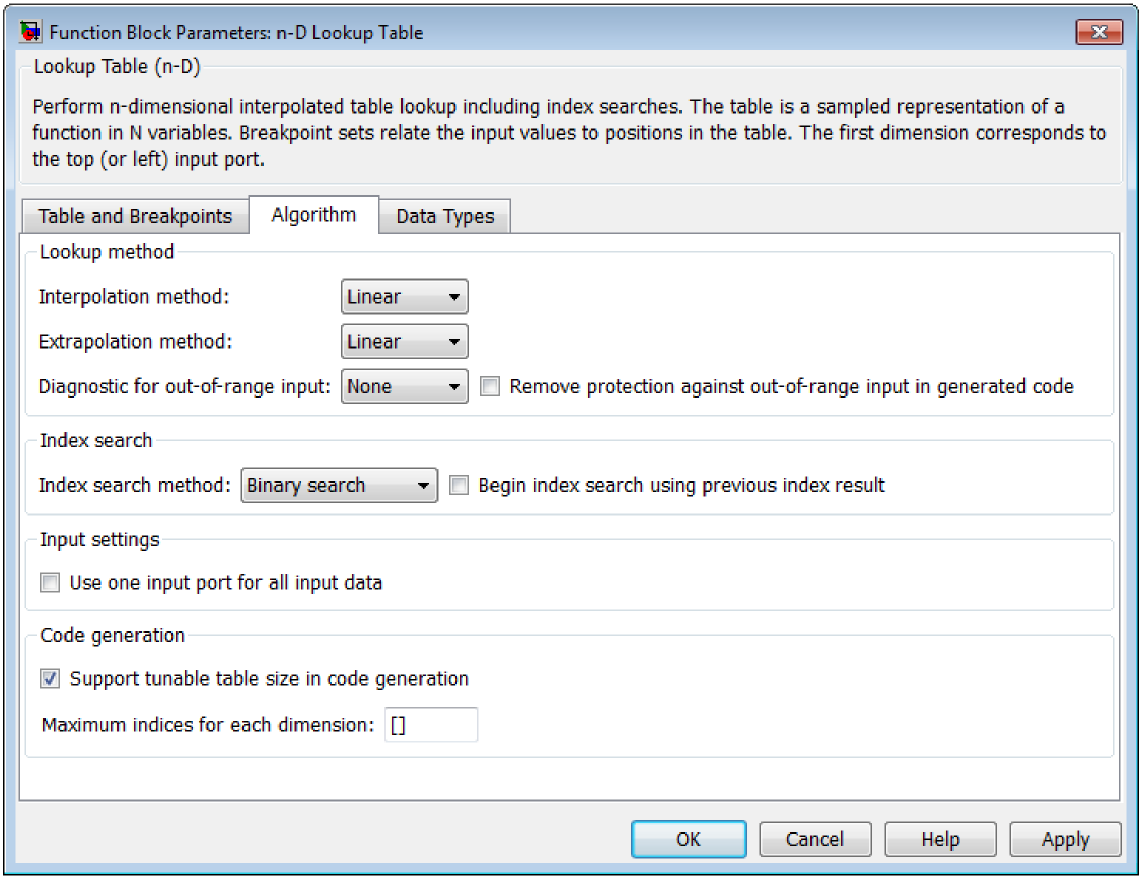
Edit table and breakpoints

Click this button to open the Lookup Table Editor. For more information, see “Edit Existing LookupTables” in the Simulink documentation.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the Simulink documentation.

The **Algorithm** pane of the n-D Lookup Table block dialog box appears as follows:



Lookup Table (n-D)

Perform n-dimensional interpolated table lookup including index searches. The table is a sampled representation of a function in N variables. Breakpoint sets relate the input values to positions in the table. The first dimension corresponds to the top (or left) input port.

Lookup method

Interpolation method: Linear

Extrapolation method: Linear

Diagnostic for out-of-range input: None ☐ Remove protection against out-of-range input in generated code

Index search

Index search method: Binary search ☐ Begin index search using previous index result

Input settings

☐ Use one input port for all input data

Code generation

☑ Support tunable table size in code generation

Maximum indices for each dimension: []

Interpolation method

Select `Flat`, `Linear`, or `Cubic spline`. See “Interpolation Methods” in the Simulink documentation for more information.

If you select `Cubic spline`, the block supports only scalar signals. The other interpolation methods support nonscalar signals.

Extrapolation method

Select `Clip`, `Linear`, or `Cubic spline`. See “Extrapolation Methods” in the Simulink documentation for more information.

To select `Cubic spline` for **Extrapolation method**, you must also select `Cubic spline` for **Interpolation method**.

Use last table value for inputs at or above last breakpoint

Specify the indexing convention that the block uses to address the last element of a breakpoint set and its corresponding table value.

Check Box	Index That the Block Uses	Interval Fraction
Selected	Last element of a breakpoint set	0
Cleared	Next-to-last element of a breakpoint set	1

This parameter is visible only when:

- **Interpolation method** is `Linear`.
- **Extrapolation method** is `Clip`.

Diagnostic for out-of-range input

Specify whether to produce a warning or error when the input is out of range. Options include:

- `None` — no warning or error
- `Warning` — display a warning in the MATLAB Command Window and continue the simulation
- `Error` — halt the simulation and display an error in the Simulation Diagnostics Viewer

Remove protection against out-of-range input in generated code

Specify whether or not to include code that checks for out-of-range breakpoint input values.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range breakpoint inputs.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range breakpoint inputs.	For safety-critical applications

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 Verification and Validation > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Consult the Model Advisor” in the Simulink documentation.

Index search method

Select `Evenly spaced points`, `Linear search`, or `Binary search`. Each search method has speed advantages in different circumstances:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting `Evenly spaced points` to calculate table indices.

  This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input signals do not vary much between time steps, selecting `Linear search` with **Begin index search using previous index result** produces the best performance.
  - If input signals jump more than one or two table intervals per time step, selecting `Binary search` produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

Note The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is `Evenly spaced points`.

Begin index search using previous index result

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

Use one input port for all input data

Select this check box to use only one input port that expects a signal that is N elements wide for an N-dimensional table. This option is useful for removing line clutter on a block diagram with many lookup tables.

Note When you select this check box, one input port with the label `u` appears on the block.

Support tunable table size in code generation

Select this check box to enable tunable table size in the generated code. This option enables you to change the size and values of the lookup table and breakpoint data without regenerating or recompiling the code.

If you set **Interpolation method** to `Cubic spline`, this check box is not available.

Maximum indices for each dimension

Specify the maximum index values for each table dimension using zero-based indexing. You can specify a scalar or vector of positive integer values using the following data types:

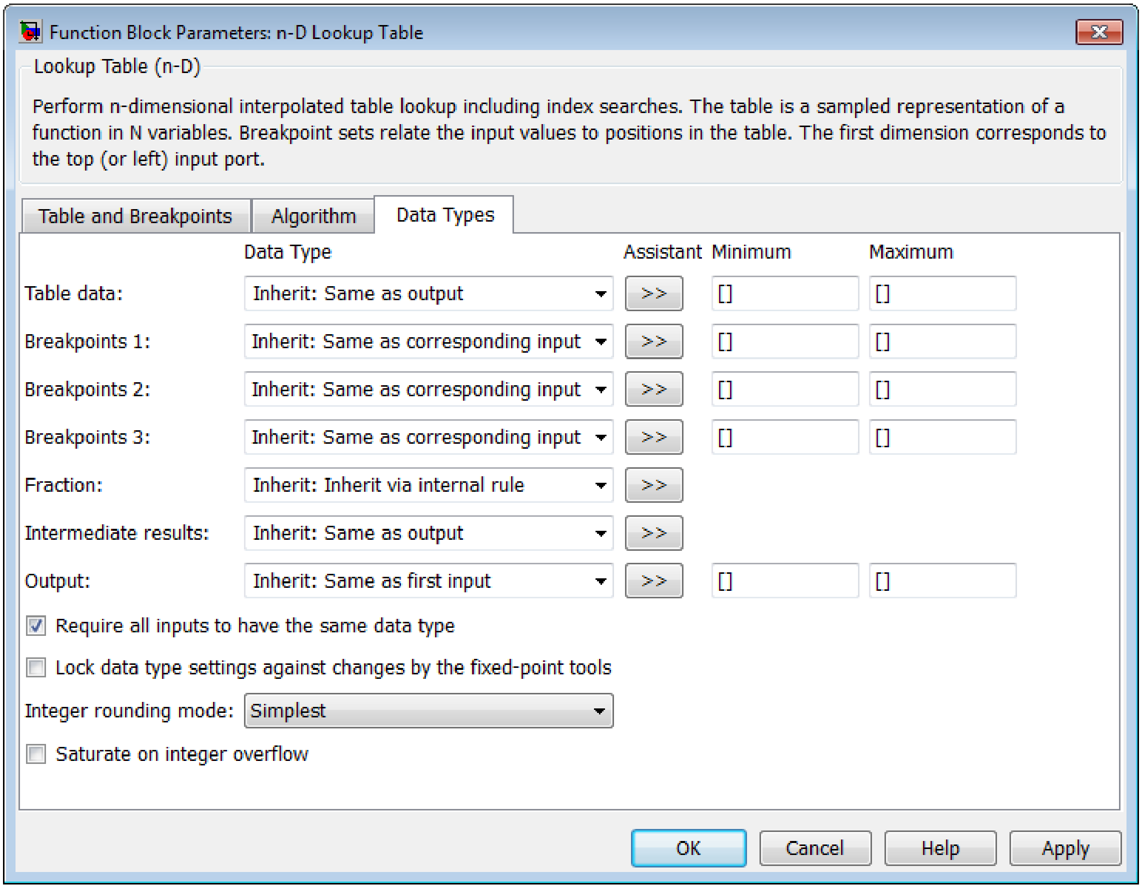
- Built-in floating-point types: `double` and `single`
- Built-in integer types: `int8`, `int16`, `int32`, `uint8`, `uint16`, and `uint32`

Here are some examples of valid specifications:

- `[4 6]` for a 5-by-7 table
- `[int8(2) int16(5) int32(9)]` for a 3-by-6-by-10 table

This parameter is available when you select **Support tunable table size in code generation**.

The **Data Types** pane of the n-D Lookup Table block dialog box appears as follows:



Function Block Parameters: n-D Lookup Table

Lookup Table (n-D)

Perform n-dimensional interpolated table lookup including index searches. The table is a sampled representation of a function in N variables. Breakpoint sets relate the input values to positions in the table. The first dimension corresponds to the top (or left) input port.

	Data Type	Assistant	Minimum	Maximum
Table data:	Inherit: Same as output	>>	[]	[]
Breakpoints 1:	Inherit: Same as corresponding input	>>	[]	[]
Breakpoints 2:	Inherit: Same as corresponding input	>>	[]	[]
Breakpoints 3:	Inherit: Same as corresponding input	>>	[]	[]
Fraction:	Inherit: Inherit via internal rule	>>		
Intermediate results:	Inherit: Same as output	>>		
Output:	Inherit: Same as first input	>>	[]	[]

☑ Require all inputs to have the same data type

☐ Lock data type settings against changes by the fixed-point tools

Integer rounding mode: Simplest

☐ Saturate on integer overflow

Note The dialog box can expand to show additional data type options. Up to 30 breakpoint data type specifications can appear.

Table data > Data Type

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the table data type.

Tip Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two n-D Lookup Table blocks with different output data types
- Sharing of custom storage table data in the generated code for blocks with different output data types

---

Table data > Minimum

Specify the minimum value for table data. The default value is `[]` (unspecified).

Table data > Maximum

Specify the maximum value for table data. The default value is `[]` (unspecified).

Breakpoints > Data Type

Specify the data type for a set of breakpoint data. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as corresponding input`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the breakpoint data type.

See “Specify Data Types Using Data Type Assistant” in the Simulink documentation for more information.

Tip Specify a breakpoint data type different from the corresponding input data type for these cases:

- Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal
- Sharing of prescaled breakpoint data between two n-D Lookup Table blocks with different input data types
- Sharing of custom storage breakpoint data in the generated code for blocks with different input data types

---

Breakpoints > Minimum

Specify the minimum value that a set of breakpoint data can have. The default value is `[]` (unspecified).

Breakpoints > Maximum

Specify the maximum value that a set of breakpoint data can have. The default value is `[]` (unspecified).

Fraction > Data Type

Specify the fraction data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the fraction data type.

See “Specify Data Types Using Data Type Assistant” in the Simulink documentation for more information.

Intermediate results > Data Type

Specify the intermediate results data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the intermediate results data type.

Tip Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.

Output > Data Type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the output data type.

See "Specify Block Output Data Types" for more information.

Output > Minimum

Specify the minimum value that the block outputs. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")
- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Output > Maximum

Specify the maximum value that the block outputs. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Require all inputs to have the same data type

Select to require all inputs to have the same data type.

Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Fixed-Point Tool” and “Preparation for Fixed-Point Conversion” in the Fixed-Point Designer documentation.

Integer rounding mode

Specify the rounding mode for fixed-point lookup table calculations that occur during simulation or execution of code generated from the model. For more information, see “Rounding” in the Fixed-Point Designer documentation.

This option does not affect rounding of values of block parameters. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to -128 or 127.
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 "Checking 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.

Examples

Interpolation and Extrapolation Behavior

For an example that illustrates linear interpolation and extrapolation methods of this block, see "Create a Logarithm Lookup Table" in the Simulink documentation.

For an example of entering breakpoint and table data, see "Entering Data in a Block Parameter Dialog Box" in the Simulink documentation.

Tunable Table Size in the Generated Code

Suppose that you have a lookup table and want to make the size tunable in the generated code. Assume that:

- You define a `Simulink.Parameter` structure in the preload function of your model:

```
p = Simulink.Parameter;
p.Value.MaxIdx = [2 2];
p.Value.BP1 = [1 2 3];
p.Value.BP2 = [1 4 16];
p.Value.Table = [4 5 6; 16 19 20; 10 18 23];
p.DataType = 'Bus: slLookupTable';
p.CoderInfo.StorageClass = 'ExportedGlobal';

% Create bus object slBus1 from MATLAB structure
Simulink.Bus.createObject(p.Value);
slLookupTable = slBus1;
slLookupTable.Elements(1).DataType = 'uint32';
```

- The following block parameters apply in the n-D Lookup Table block dialog box:

Parameter	Value
Number of table dimensions	`2`
Table data	`p.Table`
Breakpoints 1	`p.BP1`

Parameter	Value
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.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes, if you do not select `Cubic spline` for **Interpolation method**
Zero-Crossing Detection	No

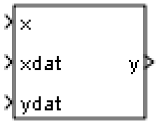
See Also

Prelookup, Interpolation Using Prelookup

Purpose Approximate one-dimensional function using dynamic table

Library Lookup Tables

Description



How This Block Differs from Other Lookup Table Blocks

The Lookup Table Dynamic block computes an approximation of a function `y = f(x)` using `xdat` and `ydat` vectors. The lookup method can use interpolation, extrapolation, or the original values of the input.

Using the Lookup Table Dynamic block, you can change the table data without stopping the simulation. For example, you can incorporate new table data if the physical system you are simulating changes.

Inputs for Breakpoint and Table Data

The `xdat` vector is the breakpoint data, which must be *strictly monotonically increasing*. The value of the next element in the vector must be greater than the value of the preceding element after conversion to a fixed-point data type. Due to quantization, `xdat` can be strictly monotonic for a floating-point data type, but not after conversion to a fixed-point data type.

The `ydat` vector is the table data, which is an evaluation of the function at the breakpoint values.

Note The inputs to `xdat` and `ydat` cannot be scalar (one-element array) values. If you provide a scalar value to either of these inputs, you see an error upon simulation. Provide a 1-by-n vector to both the `xdat` and `ydat` inputs.

Lookup Table Definition

You define the lookup table by feeding `xdat` and `ydat` as 1-by-n vectors to the block. To reduce ROM usage in the generated code for this block, you can use different data types for `xdat` and `ydat`. However, these restrictions apply:

- The `xdat` breakpoint data and the `x` input vector must have the same sign, bias, and fractional slope. Also, the precision and range for `x` must be greater than or equal to the precision and range for `xdat`.
- The `ydat` table data and the `y` output vector must have the same sign, bias, and fractional slope.

Tip Breakpoints with even spacing can make Simulink Coder generated code division-free. For more information, see `fixpt_evenspace_cleanup` in the Simulink documentation and "Identify questionable fixed-point operations" 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.

Lookup Method	Block Action
	Note If you select this lookup method, Simulink Coder software cannot generate code for this block.
`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.

Data Type Support

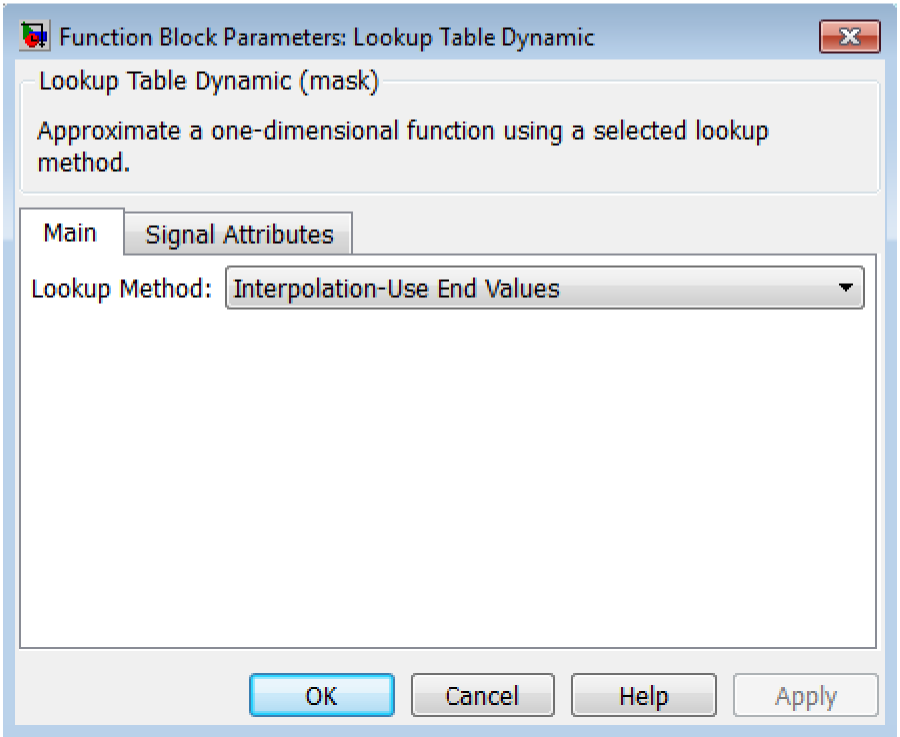
The Lookup Table Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Lookup Table Dynamic block dialog box appears as follows:



Function Block Parameters: Lookup Table Dynamic

Lookup Table Dynamic (mask)

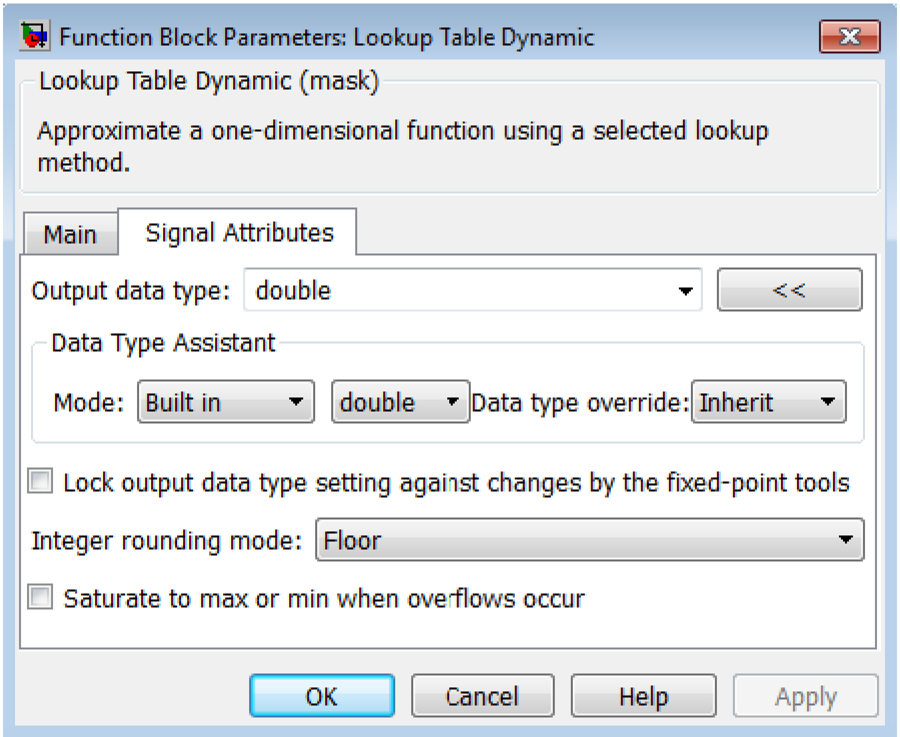
Approximate a one-dimensional function using a selected lookup method.

Lookup Method: Interpolation-Use End Values

Lookup Method

Specify the lookup method. For details, see “How the Block Generates Output” on page 1-910.

The **Signal Attributes** pane of the Lookup Table Dynamic block dialog box appears as follows:



Function Block Parameters: Lookup Table Dynamic

Lookup Table Dynamic (mask)

Approximate a one-dimensional function using a selected lookup method.

Output data type: double

Data Type Assistant

Mode: Built in double Data type override: Inherit

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt('double')`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Block Output Data Types” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

For an example of...	See...
Breakpoint and table data entry	“Entering Data Using Inports of the Lookup Table Dynamic Block”
Block output for different lookup methods	“Example Output for Lookup Methods”

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	No
Zero-Crossing Detection	No

See Also

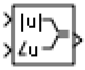
n-D Lookup Table

Purpose Convert magnitude and/or a phase angle signal to complex signal

Library Math Operations

Description

Supported Operations



The Magnitude-Angle to Complex block converts magnitude and phase angle inputs to a complex output. The angle input must be in radians.

The block supports the following combinations of input dimensions when there are two block inputs:

- Two inputs of equal dimensions
- One scalar input and the other an n-dimensional array

If the block input is an array, the output is an array of complex signals. The elements of a magnitude input vector map to the magnitudes of the corresponding complex output elements. Similarly, the elements of an angle input vector map to the angles of the corresponding complex output elements. If one input is a scalar, it maps to the corresponding component (magnitude or angle) of all the complex output signals.

Effect of Out-of-Range Input on CORDIC Approximations

If you use the CORDIC approximation method (see "Definitions" on page 1-918), 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) radians.
- For unsigned fixed-point types, the input angle must fall within the range [0, 2π) radians.

The following table summarizes what happens for an out-of-range input:

Block Usage	Effect of Out-of-Range Input
Simulation	An error appears.
Generated code	Undefined behavior occurs.
Accelerator modes	

Ensure that you use an in-range input for the Magnitude-Angle to Complex block when you use the CORDIC approximation. Avoid relying on undefined behavior for generated code or Accelerator modes.

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is among one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see [1], [2]). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

Data Type Support

The block accepts real input signals of the following data types:

- Floating point
- Fixed point (only when **Approximation method** is `CORDIC`)

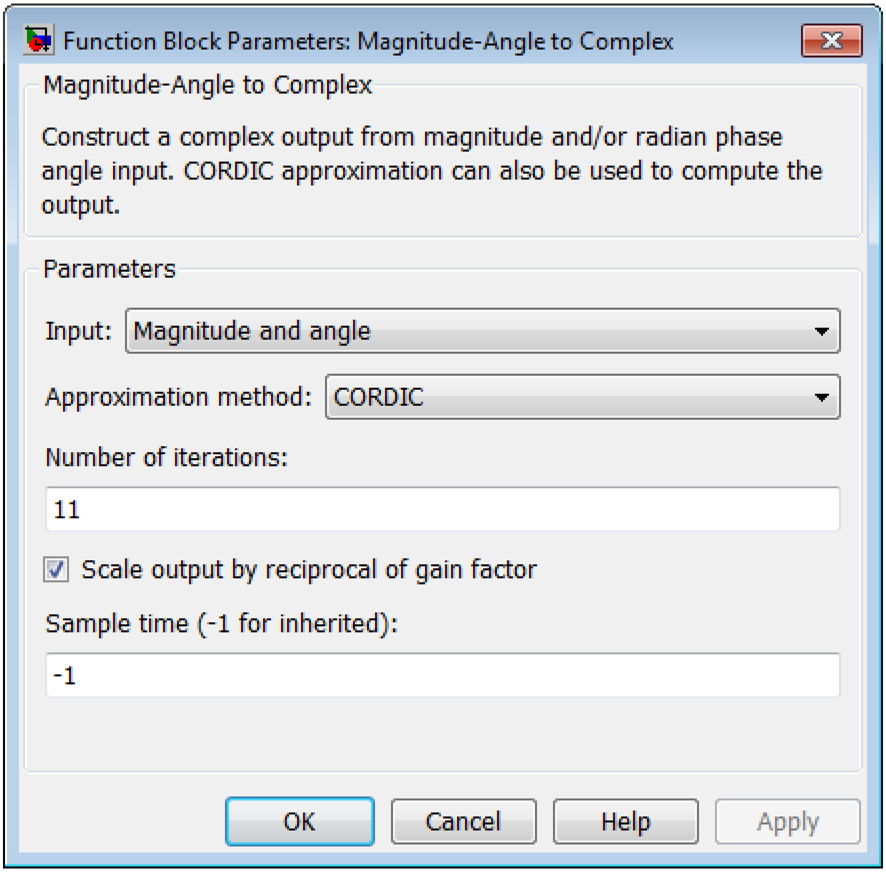
The following restrictions also apply:

- If one input uses a floating-point type, the other input must use the same data type. For example, both signals must be `double` or `single`.

- If one input uses a fixed-point type, the other input must also use a fixed-point type.

Parameters and Dialog Box

The dialog box for this block appears as follows:



Function Block Parameters: Magnitude-Angle to Complex

Magnitude-Angle to Complex

Construct a complex output from magnitude and/or radian phase angle input. CORDIC approximation can also be used to compute the output.

Parameters

Input: Magnitude and angle

Approximation method: CORDIC

Number of iterations: 11

☑ Scale output by reciprocal of gain factor

Sample time (-1 for inherited): -1

OK
Cancel
Help
Apply

Input

Specify the kind of input: a magnitude input, an angle input, or both.

Angle (Magnitude)

Input	What to Specify
`Magnitude`	The constant phase angle of the output signal in radians
`Angle`	The constant magnitude of the output signal

This parameter is not available when **Input** is `Magnitude and angle`.

Approximation method

Specify the type of approximation for computing output.

Approximation Method	Data Types Supported	When to Use This Method
`None` (default)	Floating point	You want to use the default Taylor series algorithm.
`CORDIC`	Floating point and fixed point	You want a fast, approximate calculation.

When you use the CORDIC approximation, follow these guidelines:

- For signed fixed-point types, the input angle must fall within the range [–2π, 2π) radians.
- For unsigned fixed-point types, the input angle must fall within the range [0, 2π) radians.

The block uses the following data type propagation rules:

Data Type of Magnitude Input	Approximation Method	Data Type of Complex Output
Floating point	`None` or `CORDIC`	Same as input
Signed, fixed point	`CORDIC`	`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

Number of iterations

Specify the number of iterations to perform the CORDIC algorithm. The default value is 11.

Data Type of Block Inputs	Value You Can Specify
Floating point	A positive integer
Fixed point	A positive integer that does not exceed the word length of the magnitude input or the word length of the phase angle input, whichever value is smaller

Entering a value that is not a positive integer causes an error.

This parameter is available when you set **Approximation method** to `CORDIC`.

Scale output by reciprocal of gain factor

Select this check box to scale the real and imaginary parts of the complex output by a factor of `(1/CORDIC gain)`. This value depends on the number of iterations you specify. As the number of iterations goes up, the value approaches 1.647.

This check box is selected by default, which leads to a more numerically accurate result for the complex output, `X + iY`. However, scaling the output adds two extra multiplication operations, one for `X` and one for `Y`.

This parameter is available when you set **Approximation method** to `CORDIC`.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified by the **Sample time** parameter
Scalar Expansion	Yes, of the input when the function requires two inputs
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

References

[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international*

symposium on Field programmable gate arrays. Feb. 22–24, 1998, pp. 191–200.

See Also Complex to Magnitude-Angle

Manual Switch

Purpose

Switch between two inputs

Library

Signal Routing

Description

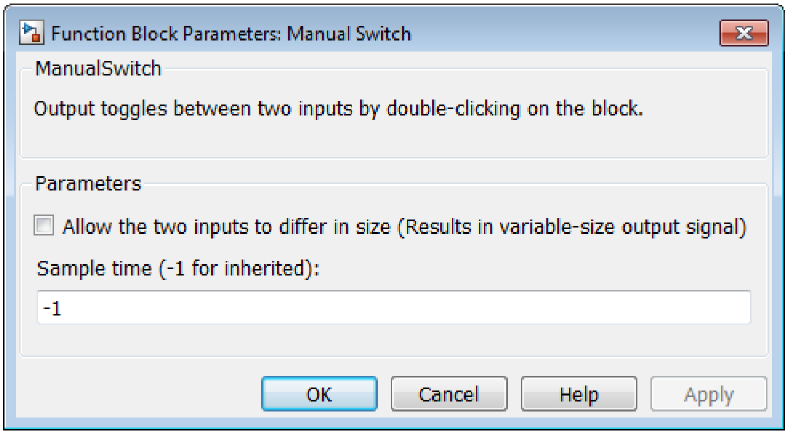
The Manual Switch block is a toggle switch that selects one of its two inputs to pass through to the output. To toggle between inputs, double-click the block. The block propagates the selected input to the output, while the block discards the unselected input. You can interactively control the signal flow by setting the switch before you start the simulation or by changing the switch while the simulation is executing. The Manual Switch block retains its current state when you save the model.

Data Type Support

The Manual Switch block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

Double-clicking the Manual Switch block toggles the input. To open the block dialog box, right-click the block and select **Block Parameters**.



Function Block Parameters: Manual Switch

ManualSwitch

Output toggles between two inputs by double-clicking on the block.

Parameters

☐ Allow the two inputs to differ in size (Results in variable-size output signal)

Sample time (-1 for inherited):

-1

- "Allow the two inputs to differ in size" on page 1-926

- “Sample time (-1 for inherited)” on page 1-1974

Allow the two inputs to differ in size

Select this check box to allow input signals with different sizes.

Settings

Default: Off

☑ On

Block allows input signals with different sizes, and propagates the input signal size to the output signal.

☐ Off

Block expands scalar inputs to have the same dimensions as nonscalar inputs. For more information, see "Scalar Expansion of Inputs and Parameters".

Command-Line Information

Parameter: `varsize`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in "Types of Sample Time".

See also "Specify Sample Time" in the online documentation for more information.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Examples

The following models show how to use the Manual Switch block:

- `sldemo_auto_climatecontrol`
- `sldemo_fuelsys`
- `sldemo_doublebounce`

Characteristics

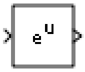
Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Math Function

Purpose Perform mathematical function

Library Math Operations

Description The Math Function block performs numerous common mathematical functions.



Tip To perform square root calculations, use the Sqrt block.

You can select one of the following functions from the **Function** parameter list.

Function	**Description**	**Mathematical Expression**	**MATLAB Equivalent**
`exp`	Exponential	e^u	`exp`
`log`	Natural logarithm	$\ln u$	`log`
`10^u`	Power of base 10	10^u	`10.^u` (see `power`)
`log10`	Common (base 10) logarithm	$\log u$	`log10`
`magnitude^2`	Complex modulus	$\|u\|^2$	`(abs(u)).^2` (see `abs` and `power`)
`square`	Power 2	u^2	`u.^2` (see `power`)
`pow`	Power	u^v	`power`
`conj`	Complex conjugate		`conj`
`reciprocal`	Reciprocal	$1/u$	`1./u` (see `rdivide`)
`hypot`	Square root of sum squares	$(u^2+v^2)^{0.5}$	`hypot`

Function	**Description**	**Mathematical Expression**	**MATLAB Equivalent**
`rem`	Remainder after division	—	`rem`
`mod`	Modulus after division	—	`mod`
`transpose`	Transpose	u^T	`u.'` (see `arithmetic operators`)
`hermitian`	Complex conjugate transpose	u^H	`u'` (see `arithmetic operators`)

The block output is the result of the operation of the function on the input or inputs. The functions support the following types of operations.

Function	**Scalar Operations**	**Element-Wise Vector and Matrix Operations**	**Vector and Matrix Operations**
`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

The following table shows the input data types that each function of the block can support.

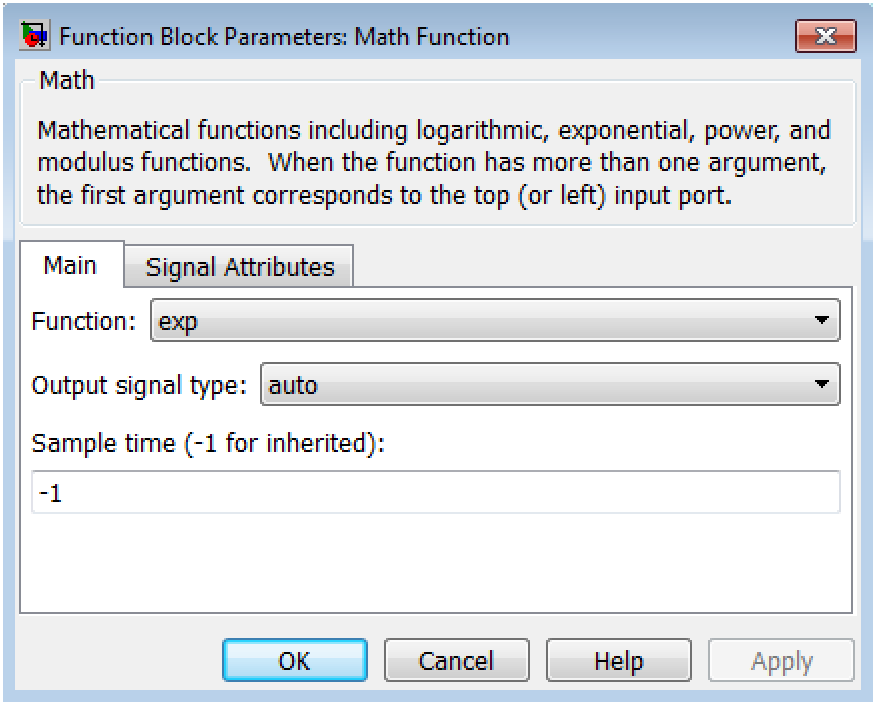
Function	single	double	boolean	built-in integer	fixed point
`exp`	yes	yes	—	—	—
`log`	yes	yes	—	—	—
`10^u`	yes	yes	—	—	—
`log10`	yes	yes	—	—	—
`magnitude^2`	yes	yes	—	yes	yes
`square`	yes	yes	—	yes	yes
`pow`	yes	yes	—	—	—
`conj`	yes	yes	—	yes	yes
`reciprocal`	yes	yes	—	yes	yes
`hypot`	yes	yes	—	—	—
`rem`	yes	yes	—	yes	—
`mod`	yes	yes	—	yes	—
`transpose`	yes	yes	yes	yes	yes
`hermitian`	yes	yes	—	yes	yes

All supported modes accept both real and complex inputs, except for `reciprocal`, which does not accept complex fixed-point inputs.

The block output is real or complex, depending on what you select for **Output signal type**.

Parameters and Dialog Box

The **Main** pane of the Math Function block dialog box appears as follows:



Function Block Parameters: Math Function

Math

Mathematical functions including logarithmic, exponential, power, and modulus functions. When the function has more than one argument, the first argument corresponds to the top (or left) input port.

Function: exp

Output signal type: auto

Sample time (-1 for inherited): -1

Function

Specify the mathematical function. See Description for more information about the options for this parameter.

Output signal type

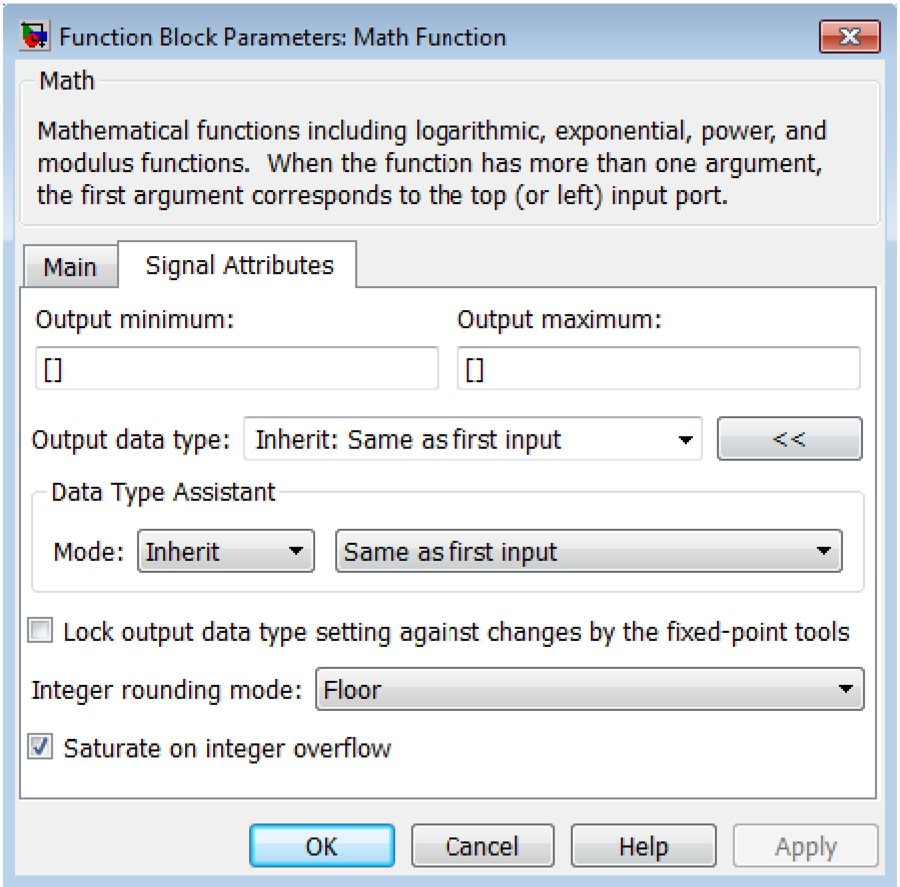
Specify the output signal type of the Math Function block as `auto`, `real`, or `complex`.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
`exp`, `log`, `10u`, `log10`, `square`, `pow`, `reciprocal`, `conjugate`, `transpose`, `hermitian`	`real` `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`

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

The **Signal Attributes** pane of the Math Function block dialog box appears as follows:



Function Block Parameters: Math Function

Math

Mathematical functions including logarithmic, exponential, power, and modulus functions. When the function has more than one argument, the first argument corresponds to the top (or left) input port.

Output minimum: []

Output maximum: []

Output data type: Inherit: Same as first input

Data Type Assistant

Mode: Inherit — Same as first input

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☑ Saturate on integer overflow

Note Some parameters on this pane are available only when the function you select in the **Function** parameter supports fixed-point data types.

Output minimum

Specify the minimum value that the block can output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block can output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Use Lock Output Data Type Setting".

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see "Rounding". in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 "Checking 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.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of the input when the function requires two inputs
Dimensionalized	Yes
Multidimensionalized	Yes, for all functions except `hermitian` and `transpose`
Zero-Crossing Detection	No

See Also

Sqrt, Trigonometric Function

MATLAB Function

Purpose

Include MATLAB code in models that generate embeddable C code

Library

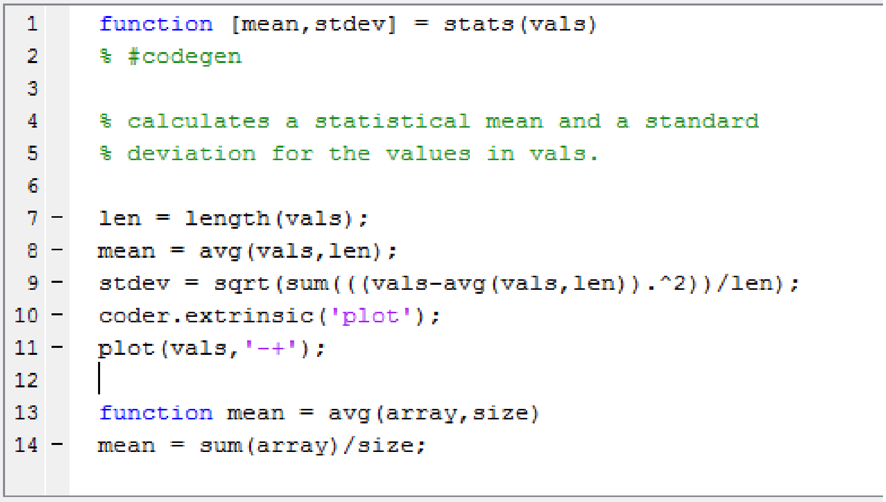
User-Defined Functions

Description

MATLAB Function

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.

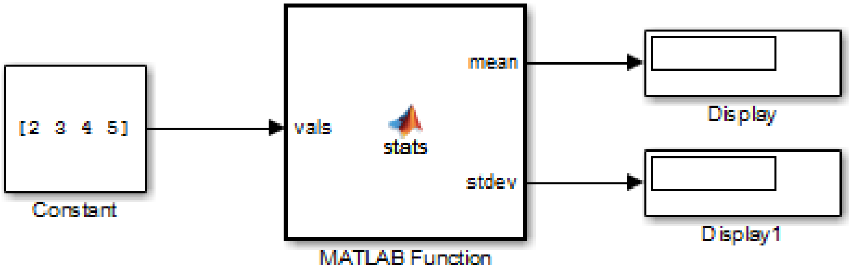
Double-clicking the MATLAB Function block opens its editor, where you write the MATLAB function, as in this example:



```
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 Supported for Code Generation — Alphabetical List”. These functions include functions in common categories, such as:

- Arithmetic operators like `plus`, `minus`, and `power`. For more information, see “Arithmetic Operators”.
- 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 Supported for Code Generation — Categorical 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.

To support visualization of data, the MATLAB Function block supports calls to MATLAB functions for simulation only. See “Call MATLAB 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.

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.

Data Type Support

The MATLAB Function block accepts inputs of any type that Simulink supports, including fixed-point and enumerated types. For more information, see “Data Types Supported by Simulink”.

For more information on fixed-point support for this block, refer to “Fixed-Point Data Types with MATLAB Function Block” and “MATLAB Function Block with Data Type Override”.

The MATLAB Function block supports Simulink frames. For more information, see “Sample- and Frame-Based Concepts”.

Parameters and Dialog Box

The block dialog box for a MATLAB Function block is identical to the dialog box for a Subsystem block. See the reference page for the Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem blocks for information about each block parameter.

Examples

The following models shows how to use the MATLAB Function block:

- `sldemo_radar_eml`
- `sldemo_eml_galaxy`

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Memory

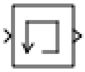
Purpose

Output input from previous time step

Library

Discrete

Description



The Memory block holds and delays its input by one integration time step. This block accepts and outputs continuous signals. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds and delays all elements of the vector by the same time step.

You specify the block output for the first time step with the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior. However, you cannot specify the sample time. This block can only inherit the sample time from the driving block or the solver used for the entire model. For more information, see the description for the **Inherit sample time** parameter.

Tip Avoid using the Memory block when both these conditions are true:

- Your model uses the variable-step solver `ode15s` or `ode113`.
- The input to the block changes during simulation.

When the Memory block inherits a discrete sample time, the block is analogous to the Unit Delay block. However, the Memory block does not support state logging. If logging the final state is necessary, use a Unit Delay block instead.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
Unit Delay	Implement a delay using a discrete sample time that you specify. Ideally, the block accepts and outputs signals with a discrete sample time.	• `sldemo_enginewc` (Compression subsystem)
Memory	Implement a delay by one integration time step. Ideally, the block accepts and outputs signals where the sample time is continuous or fixed in minor time step. For more information, see “Types of Sample Time” in the Simulink documentation.	• `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`

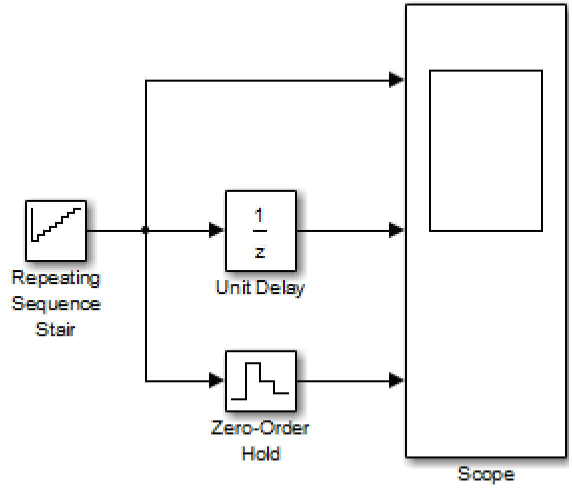
Overview of Block Capabilities

Capability	Block		
	Unit Delay	**Memory**	**Zero-Order Hold**
Specification of initial condition	Yes	Yes	No, because the block output at time t = 0 must match the input value.
Specification of sample time	Yes	No, because the block can only inherit sample time (from the driving block or the solver used for the entire model).	Yes
Support for frame-based signals	Yes	No	Yes
Support for state logging	Yes	No	No

Effect of Solver Specification on Block Output

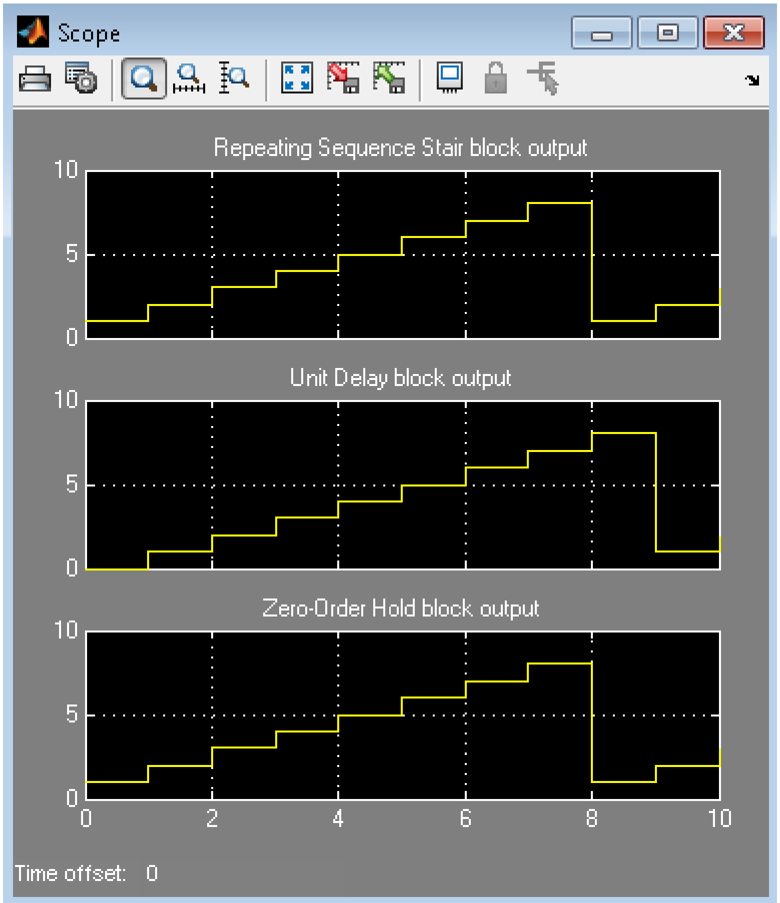
When you specify a discrete sample time in the dialog box for a Unit Delay or Zero-Order Hold block, the block output can differ depending on the solver specification for the model.

Suppose that you have a model with Unit Delay and Zero-Order Hold blocks, which both use a discrete sample time of 1:

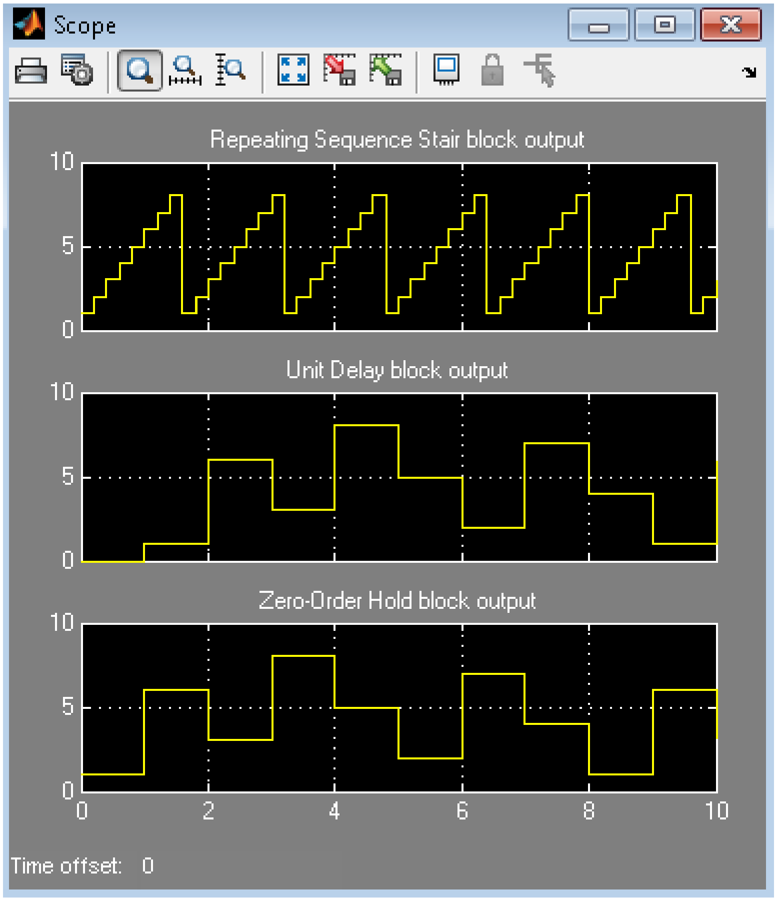

Repeating
Sequence
Stair
1
z
Unit Delay
Zero-Order
Hold
Scope

The Repeating Sequence Stair block uses a continuous sample time of 0 to provide input signals to the Unit Delay and Zero-Order Hold blocks.

If the model uses a fixed-step solver with a step size of 1, the scope shows the following simulation results:


Scope
Repeating Sequence Stair block output
Unit Delay block output
Zero-Order Hold block output
0
5
10
0
2
4
6
8
10
Time offset: 0

If the model uses a variable-step solver, the scope shows the following simulation results:


Scope
Repeating Sequence Stair block output
Unit Delay block output
Zero-Order Hold block output
0
5
10
0
2
4
6
8
10
Time offset: 0

The Zero-Order Hold block takes the input value of the Repeating Sequence Stair block at t = 0, 1, 2, ... , 9 and holds each input value for a sample period (1 second). The Unit Delay block applies the same 1-second hold to each input value of the Repeating Sequence Stair block, but also delays each value by a sample period. The **Initial conditions**

parameter specifies the output for the Unit Delay block during the first sample period. For more information about sample time, see “What Is Sample Time?” and “Specify Sample Time”.

Solver specification for a model also affects the behavior of the Memory block. For details, see “Examples of Memory Block Usage” on page 1-952.

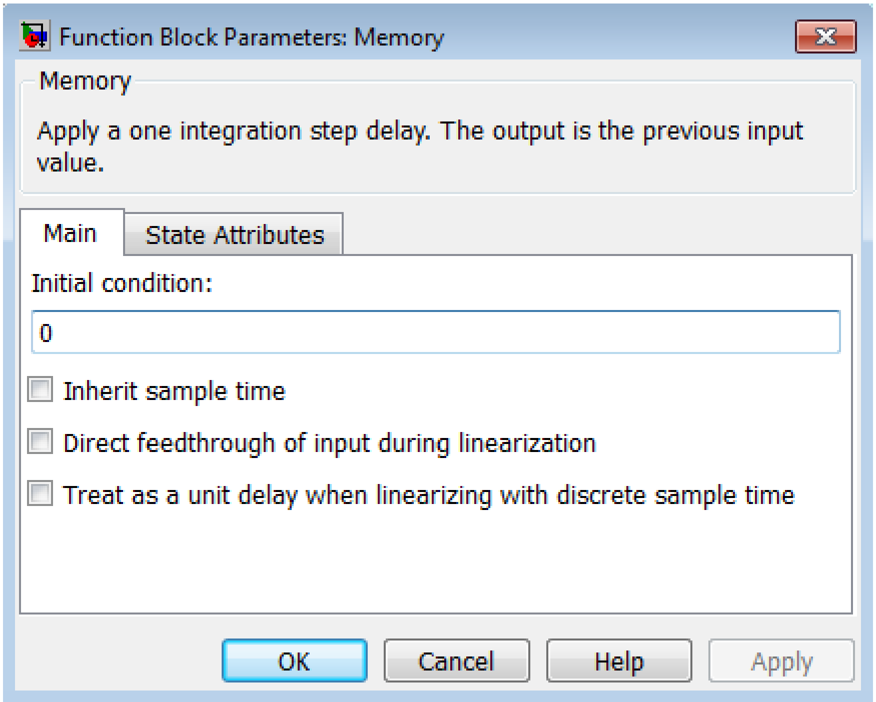
Data Type Support

The Memory block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Memory block dialog box appears as follows:



Function Block Parameters: Memory

Memory

Apply a one integration step delay. The output is the previous input value.

Initial condition: 0

☐ Inherit sample time

☐ Direct feedthrough of input during linearization

☐ Treat as a unit delay when linearizing with discrete sample time

Initial condition

Specify the output at the initial integration step. This value must be 0 when you do not use a built-in input data type. Simulink does not allow the initial output of this block to be `inf` or `NaN`.

Inherit sample time

Select to inherit the sample time from the driving block. If you do not select this check box, the block sample time depends on the type of solver for simulating the model.

- If the solver is a variable-step solver, the sample time is continuous but fixed in minor time step: `[0, 1]`.
- If the solver is a fixed-step solver, this `[0, 1]` sample time converts to the solver step size after sample-time propagation.

Direct feedthrough of input during linearization

Select to output the input during linearization and trim. This selection sets the block mode to direct feedthrough.

Selecting this check box can cause a change in the ordering of states in the model when using the functions `linmod`, `dlinmod`, or `trim`. To extract this new state ordering, use the following commands.

First compile the model using the following command, where `model` is the name of the Simulink model.

```
[sizes, x0, x_str] = model([],[],[],'lincompile');
```

Next, terminate the compilation with the following command.

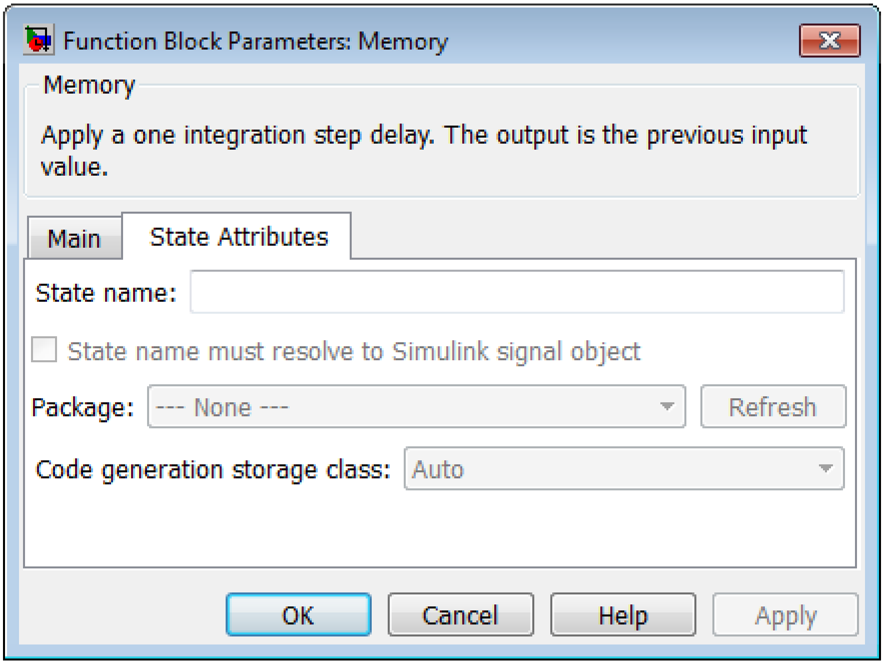
```
model([],[],[],'term');
```

The output argument, `x_str`, which is a cell array of the states in the Simulink model, contains the new state ordering. When passing a vector of states as input to the `linmod`, `dlinmod`, or `trim` functions, the state vector must use this new state ordering.

Treat as a unit delay when linearizing with discrete sample time

Select to linearize the Memory block to a unit delay when the Memory block is driven by a signal with a discrete sample time.

The **State Attributes** pane of the Memory block dialog box appears as follows:



Function Block Parameters: Memory

Memory

Apply a one integration step delay. The output is the previous input value.

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Code generation storage class: Auto

State name

Use this parameter to assign a unique name to the block state. The default is `'  '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “States” in the Simulink Coder documentation.

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Package

Select a package that defines the custom storage class you want to apply. If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Code generation storage class

From the list, select a state storage class.

`Auto`

Auto is the storage class to use for states that do not need to interface to external code.

`ExportedGlobal`

The state is stored in a global variable.

`ImportedExtern`

`model_private.h` declares the state as an extern variable.

`ImportedExternPointer`

`model_private.h` declares the state as an extern pointer.

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Code generation storage type qualifier

Specify a Simulink Coder storage type qualifier. The default is `' '`. When this field is blank, no qualifier is assigned.

The Simulink Coder product does not check this string for errors. Thus, whatever value you enter appears automatically in the variable declaration.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

During simulation, the block uses the following values:

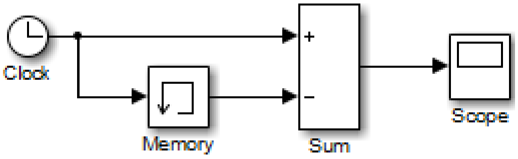
- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

See "States" in the Simulink Coder documentation for more information.

Examples of Memory Block Usage

Usage with the Clock Block

The following model shows how to display the step size in a simulation. The Sum block subtracts the time at the previous step, which the Memory block generates, from the current time, which the Clock block generates.

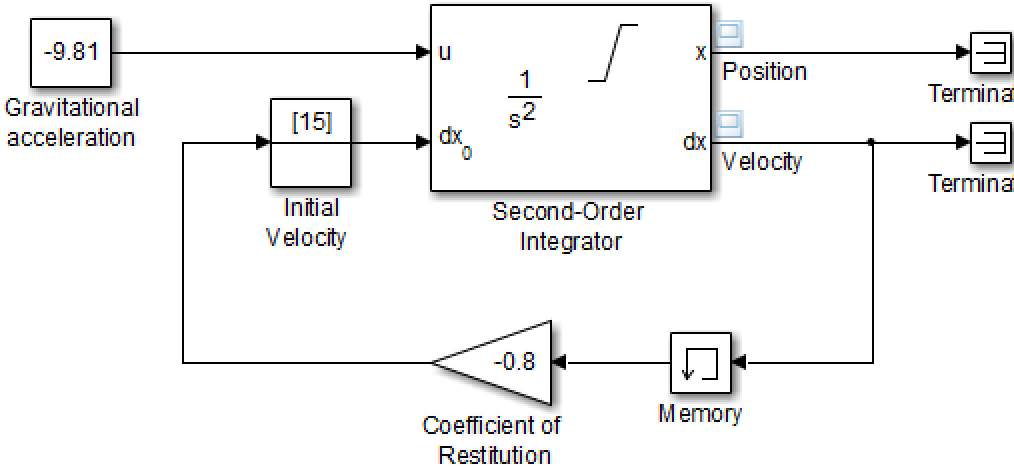


Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a fixed-step solver. Therefore, the sample time of the Memory block is the solver step size, or 1.

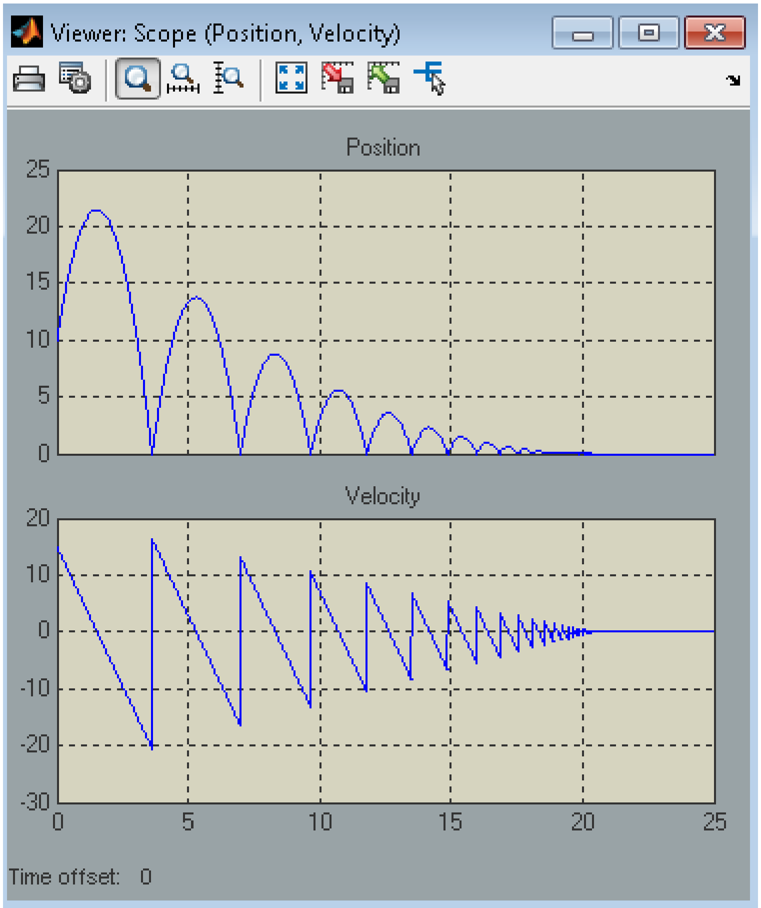
If you replace the Memory block with a Unit Delay block, you get the same results. The Unit Delay block inherits a discrete sample time of 1.

Usage with the Second-Order Integrator Block

The `sldemo_bounce` model shows how a bouncing ball reacts after being tossed into the air. The **dx** port of the Second-Order Integrator block and the Memory block capture the velocity of the ball just before it hits the ground.


-9.81
Gravitational acceleration
[15]
Initial Velocity
u
dx0
$\frac{1}{s^2}$
x
dx
Second-Order Integrator
Position
Velocity
Termina
Termina
-0.8
Coefficient of Restitution
Memory

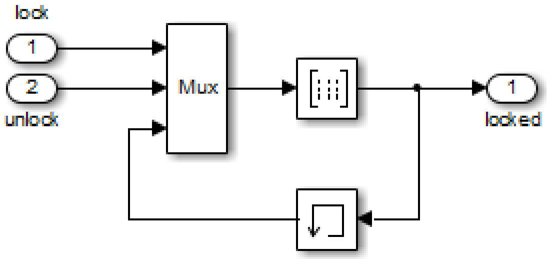
Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step (`ode23`) solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: `[0, 1]`. When you run the model, you get the following results:


Viewer: Scope (Position, Velocity)
Position
25
20
15
10
5
0
Velocity
20
10
0
-10
-20
-30
0
5
10
15
20
25
Time offset: 0

If you replace the Memory block with a Unit Delay block, you get the same results. However, a warning also appears due to the discrete Unit Delay block inheriting a continuous sample time.

Usage with the Combinatorial Logic Block

The `sldemo_clutch` model shows how you can use the Memory block with the Combinatorial Logic block to implement a finite-state machine. This construct appears in the `Friction Mode Logic/Lockup FSM` subsystem.



Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step `(ode23)` solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: `[0, 1]`.

Bus Support

The Memory block is a bus-capable block. The input can be a virtual or nonvirtual bus signal subject to the following restrictions:

- **Initial condition** must be zero, a nonzero scalar, or a finite numeric structure.
- If **Initial condition** is zero or a structure, and you specify a **State name**, the input cannot be a virtual bus.
- If **Initial condition** is a nonzero scalar, you cannot specify a **State name**.

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Memory block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See "Nonvirtual Bus Sample Times" and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Memory block. Only the **Initial condition** parameter (which may be, but does not have to be, a structure) is scalar-expanded to match the dimensions of the array of buses. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

Characteristics

Bus-capable	Yes, with restrictions as noted above
Direct Feedthrough	No, except when you select **Direct feedthrough of input during linearization**
Sample Time	Continuous, but inherited from the driving block when you select the **Inherit sample time** check box
Scalar Expansion	Yes, of the **Initial condition** parameter
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Zero-Order Hold

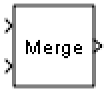
Purpose

Combine multiple signals into single signal

Library

Signal Routing

Description



The Merge block combines its inputs into a single output line whose value at any time is equal to the most recently computed output of its driving blocks. You can specify any number of inputs by setting the block's **Number of inputs** parameter.

Use Merge blocks only to interleave input signals that update at different times into a combined signal in which the interleaved values retain their separate identities and times. To combine signals that update at the same time into an array or matrix signal, use a Concatenate block.

Merge blocks assume that all driving signals share the same signal memory. The shared signal memory should be accessed only in mutually exclusive fashion. Therefore, always use alternately executing subsystems to drive Merge blocks. See "Creating Alternately Executing Subsystems" for an example.

All signals that connect to a Merge block, or exist anywhere in a network of Merge blocks, are functionally the same signal, and are therefore subject to the restriction that a given signal can have at most one associated signal object. See `Simulink.Signal` for more information.

Guidelines for Using the Merge Block

When you use the Merge block, follow these guidelines:

- Always use conditionally-executed subsystems to drive Merge blocks.
- Write your control logic to ensure that at most one of the driving conditionally-executed subsystems executes at any time step.
- Do not connect more than one input of a Merge block to the same conditionally-executed subsystem.
- Always connect a Merge block to at least two input signals.
- Ensure that all input signals have the same sample time.

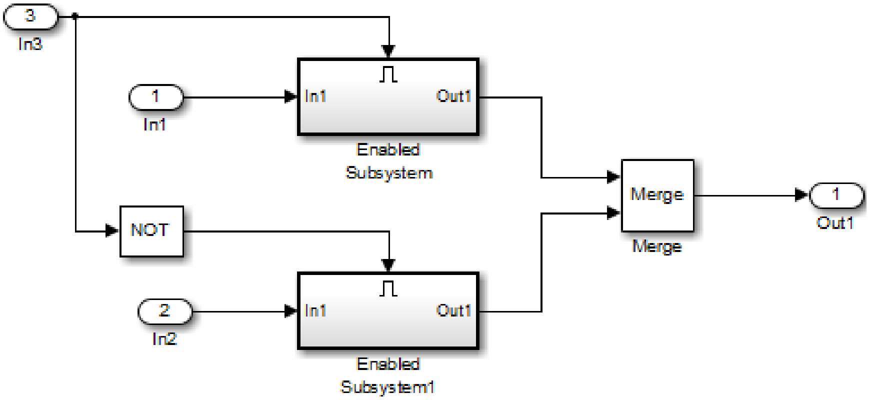
- Always set the **Initial output** parameter of the Merge block, unless the output port of the Merge block connects to another Merge block.
- Do not branch a signal that inputs to a Merge block, if you use the default setting of `Classic` for the **Model Configuration Parameters > Diagnostics > Underspecified initialization detection** parameter. See the last example in "Merge Block Usage" on page 1-958 for additional usage guidelines relating to branched signals.
- For all conditionally-executed subsystem Outport blocks that drive Merge blocks, set the **Output when disabled** parameter to `held`.

Merge Block Usage

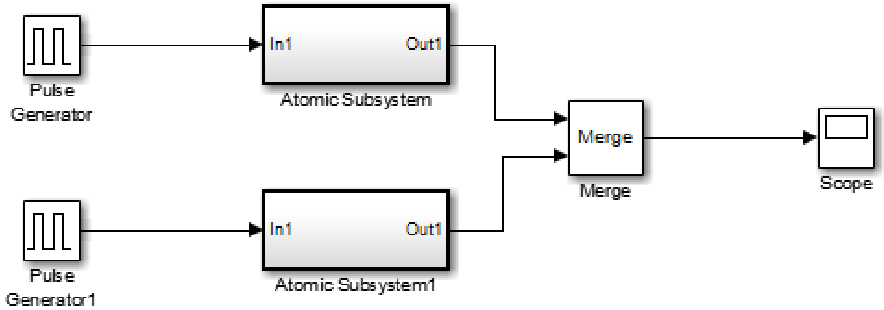
For each input of a Merge block, the topmost non-atomic and nonvirtual source must be a conditionally-executed subsystem that is not an Iterator Subsystem.

You can use the Model Advisor to check for Merge block usage in your model. For more information, see "Check Merge block usage" on page 8-36.

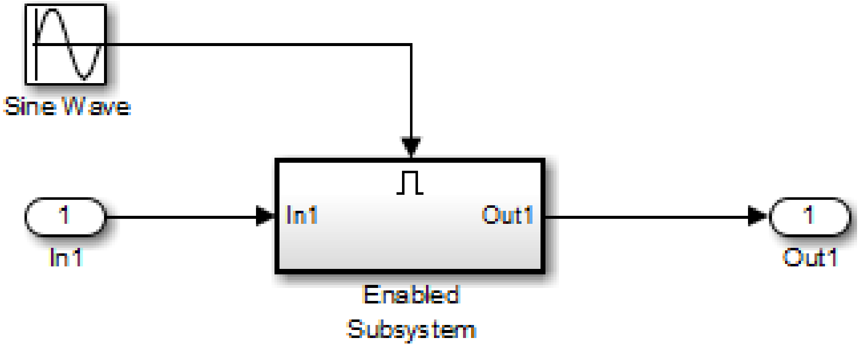
The following schematic shows valid Merge block usage, merging signals from two conditionally-executed subsystems.



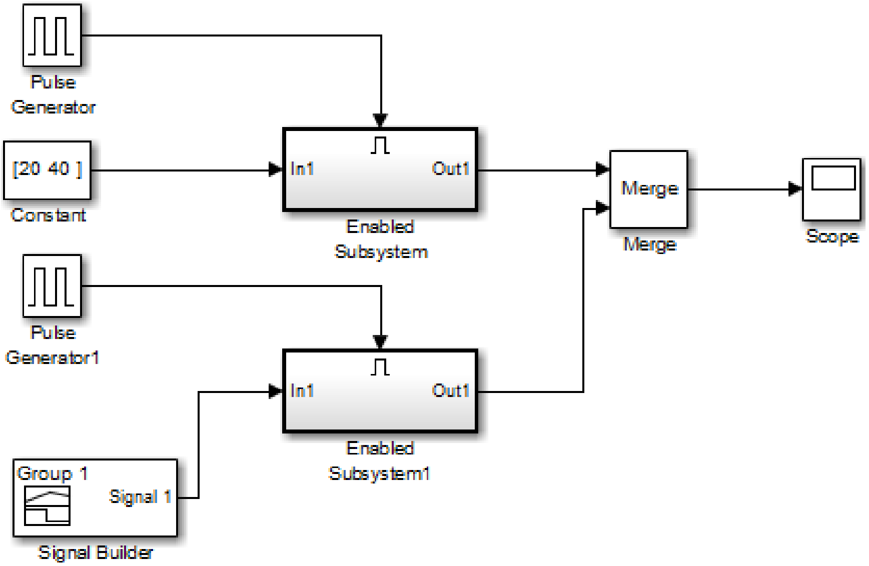
The following example is also a valid Merge block usage, where the topmost nonatomic, nonvirtual source is a conditionally executed subsystem.


Pulse Generator
In1
Out1
Atomic Subsystem
Merge
Merge
Scope
Pulse Generator1
In1
Out1
Atomic Subsystem1

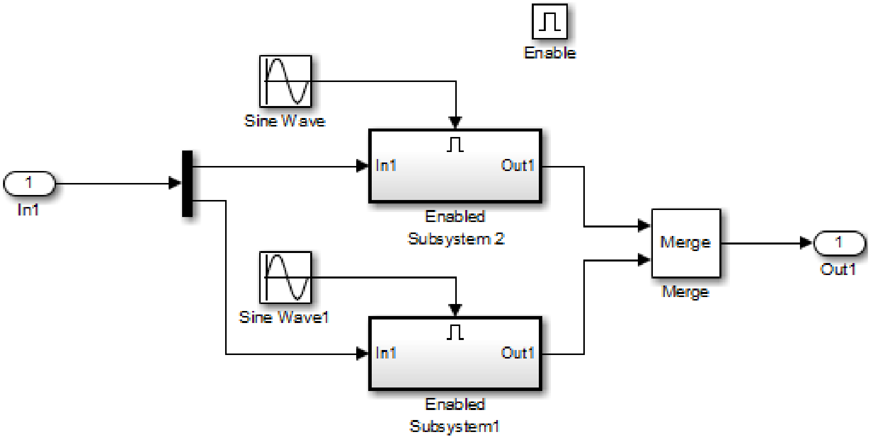
Each Atomic Subsystem block contains an enabled subsystem.


Sine Wave
1
In1
In1
Out1
Enabled Subsystem
1
Out1

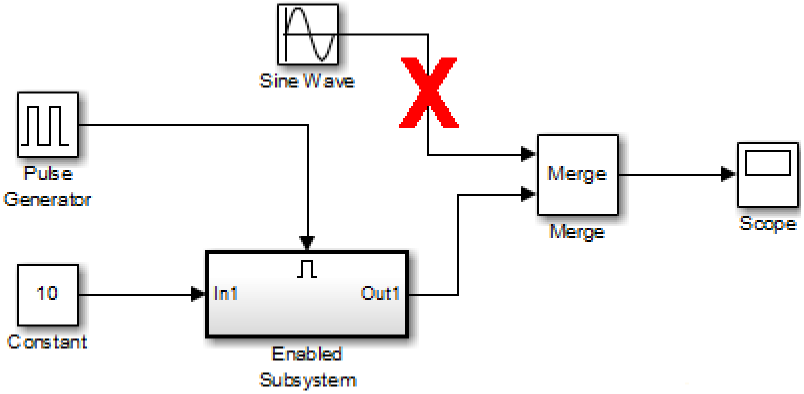
You can also use multiple Merge blocks at different levels of the model hierarchy. The following example contains a Merge block at the model root.


Pulse Generator
[20 40]
Constant
In1
Out1
Enabled Subsystem
Merge
Merge
Scope
Pulse Generator1
In1
Out1
Enabled Subsystem1
Group 1
Signal 1
Signal Builder

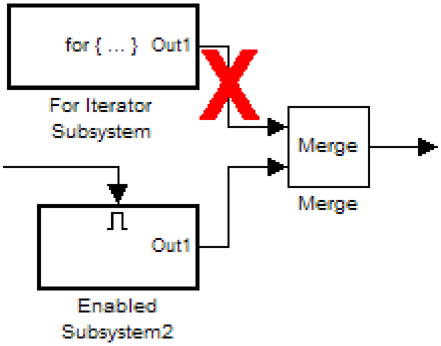
A Merge block is also located inside the Enabled Subsystem block, one level down.


Enable
Sine Wave
1
In1
In1
Out1
Enabled Subsystem 2
Merge
Merge
1
Out1
Sine Wave1
In1
Out1
Enabled Subsystem1

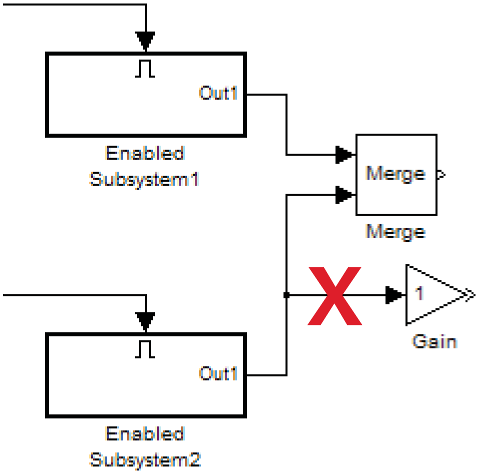
A Merge block *cannot* connect to a Sine Wave block because that source block is not a conditionally-executed subsystem.


Sine Wave
X
Pulse Generator
Merge
Merge
Scope
10
Constant
In1
Out1
Enabled Subsystem

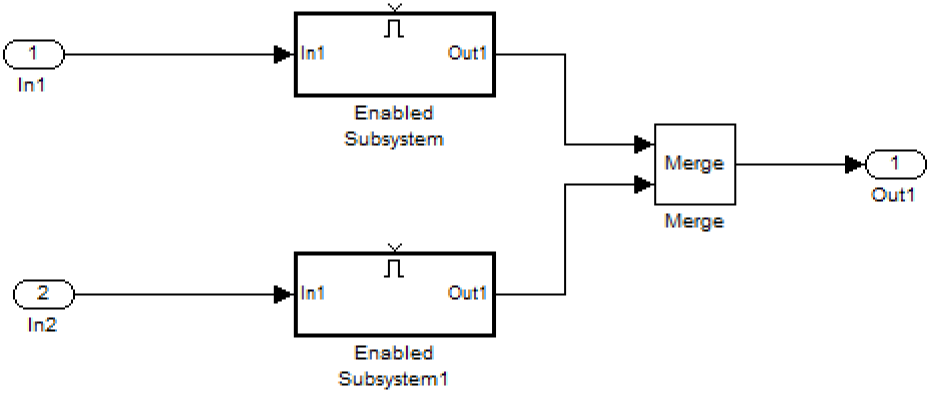
A Merge block *cannot* connect to a For Iterator Subsystem.


for { ... }
Out1
For Iterator Subsystem
X
Merge
Merge
Out1
Enabled Subsystem2

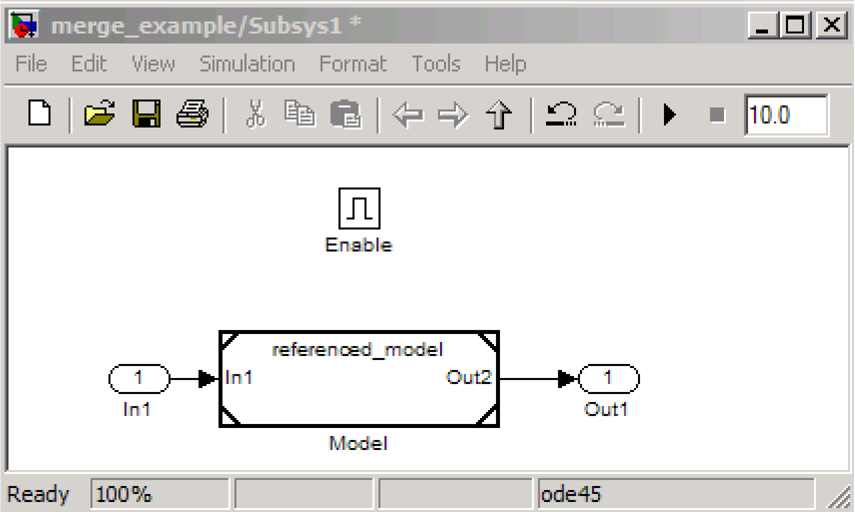
A Merge block *cannot* connect to a branched signal.


Out1
Enabled
Subsystem1
Merge
Merge
1
Gain
Out1
Enabled
Subsystem2

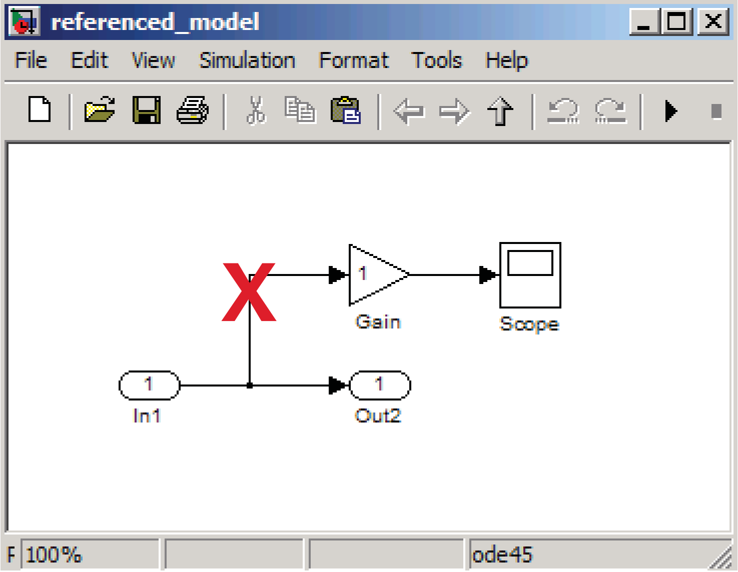
In the following model, the referenced model has a signal that branches. It inputs to a block in the referenced model and also inputs to the Merge block that is outside of the referenced model.


1
In1
In1
Out1
Enabled
Subsystem
Merge
Merge
1
Out1
2
In2
In1
Out1
Enabled
Subsystem1

Although the top-level model appears that it would work, the `Subsys1` subsystem includes a Model block that references `referenced_model`.

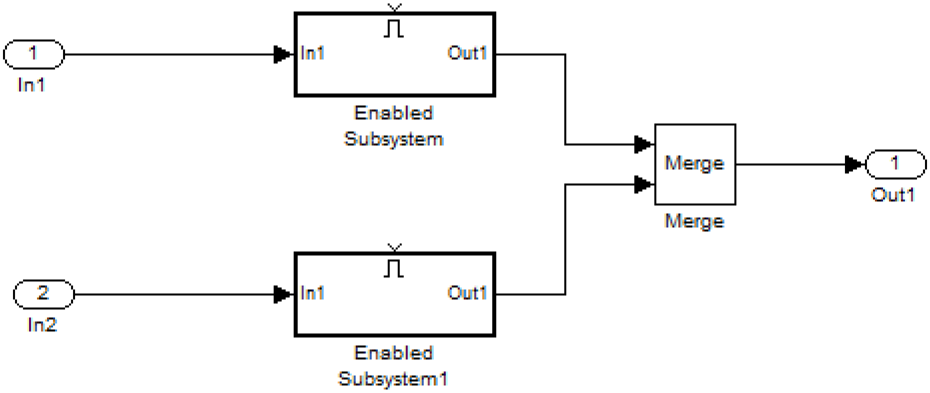

merge_example/Subsys1 *
File Edit View Simulation Format Tools Help
10.0
Enable
referenced_model
In1
Out2
1
In1
1
Out1
Model
Ready
100%
ode45

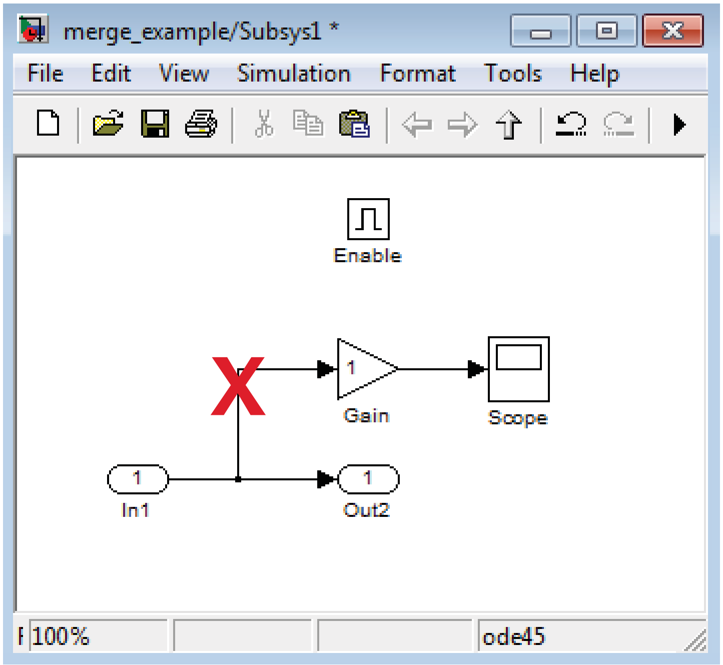
The referenced model includes a signal that incorrectly branches to a Gain block and to the `Out2` Outport block, which connects to the Merge block that is outside of the referenced model.


referenced_model
File Edit View Simulation Format Tools Help
1
Gain
Scope
1
In1
1
Out2
100%
ode45

The following example also shows a branched signal in a subsystem that connects to a Merge block, which is *not allowed* if you use the default setting of `Classic` for the **Model Configuration Parameters > Diagnostics > Underspecified initialization detection** parameter.

If you set the **Underspecified initialization detection** parameter to `Simplified`, then the following example does *not* generate an error. For more information on simplified initialization mode, see "Underspecified initialization detection".


1
In1
In1
Out1
Enabled Subsystem
Merge
Merge
1
Out1
2
In2
In1
Out1
Enabled Subsystem1


merge_example/Subsys1 *
File
Edit
View
Simulation
Format
Tools
Help
Enable
1
Gain
Scope
1
In1
1
Out2
100%
ode45

Initial Output Value

You can specify an initial output value for the Merge block by setting the **Initial output** parameter. If you do not specify an initial output value and one or more of the driving blocks do, the initial output of the Merge block equals the most recently evaluated initial output of the driving blocks.

When two or more sources initialize the same Merge block, the initialization ordering for these sources may vary. If you leave the initial output unspecified, initialization may be inconsistent for the simulation and the code generation of a model. For example, the following model can produce inconsistent initialization:

- The model contains a Merge block with two inputs: one driven by a Stateflow chart and the other driven by a conditionally executed subsystem (such as an Enabled Subsystem).
- The Merge block **Initial output** parameter is unspecified (that is, specified as empty matrix (`[]`)).
- The Stateflow chart initializes the output being merged to `val1`.
- The conditionally executed subsystem initializes the output being merged to different value `val2`.
- Both the Stateflow chart and the conditionally executed susystem do not execute at the first time step.

Because the initialization ordering may vary, the output of the Merge block at the first time step is `val1` if the Stateflow chart initializes last and `val2` if the conditionally executed subsystem initializes last. The initialization ordering is different for simulation and code generation.

To address this issue, use one of the following approaches:

- Set the **Initial output** parameter of the Merge block, unless the output port of the Merge block connects to another Merge block.

- Turn on simplified initialization mode: set the **Model Configuration Parameters > Diagnostics > Data Validity > Underspecified initialization detection** parameter to `Simplified`.

  To use the `Simplified` initialization setting, specify the **Initial output** value for all *root* Merge blocks. A root Merge block is any Merge block with an output port that does not connect to another Merge block.

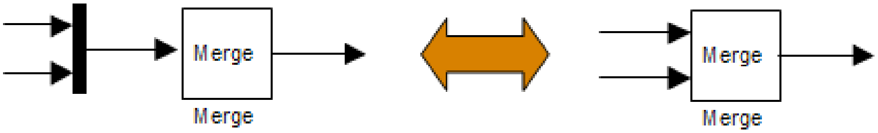
  To upgrade your model to simplified initialization mode, use the Model Advisor **Check consistency of initialization parameters for Outport and Merge blocks** check.

  For more information on simplified initialization mode, see "Underspecified initialization detection".

Single-Input Merge

Single-input merge is not supported. Each Merge block must have at least two inputs.

Use Merge blocks only for signals that require merging. If you were previously connecting a Merge block input to a Mux block, use a multi-input Merge block instead.



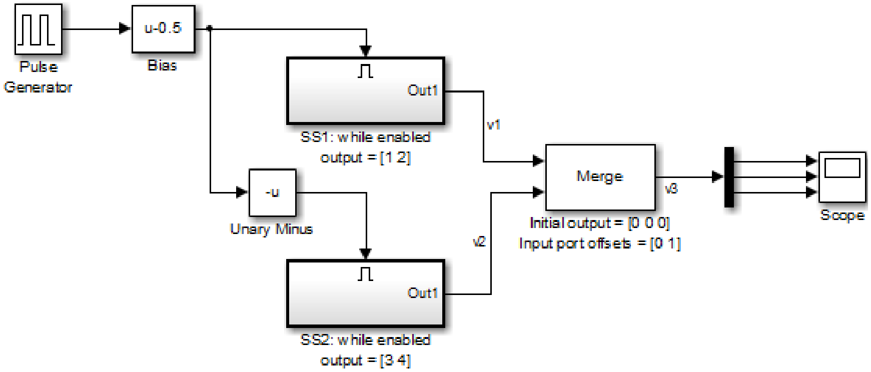
Input Dimensions and Merge Offsets

The Merge block accepts only inputs of equal dimensions and outputs a signal of the same dimensions as the inputs, unless you select the **Allow unequal port widths** parameter.

If you select **Allow unequal port widths**, the block accepts scalars and vectors (but not matrices) having differing numbers of elements. Further, the block allows you to specify an offset for each input signal relative to the beginning of the output signal. The width of the output signal is `max(w1+o1, w2+o2, ... wn+on)`

where w_1, ... w_n are the widths of the input signals and o_1, ... o_n are the offsets for the input signals.

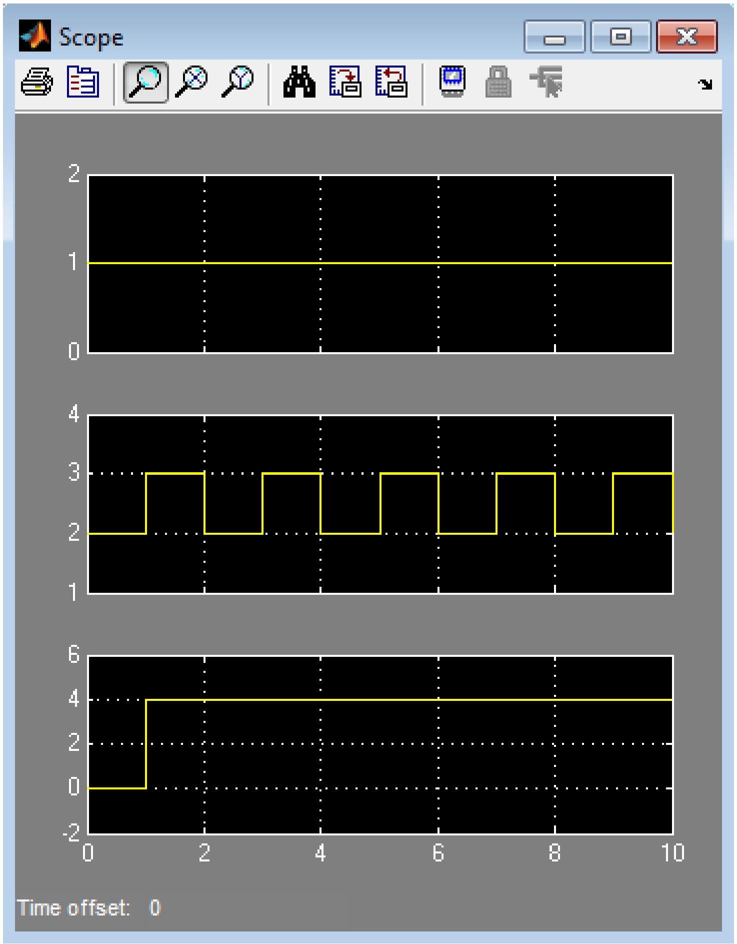
Suppose that you have the following block diagram:


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:

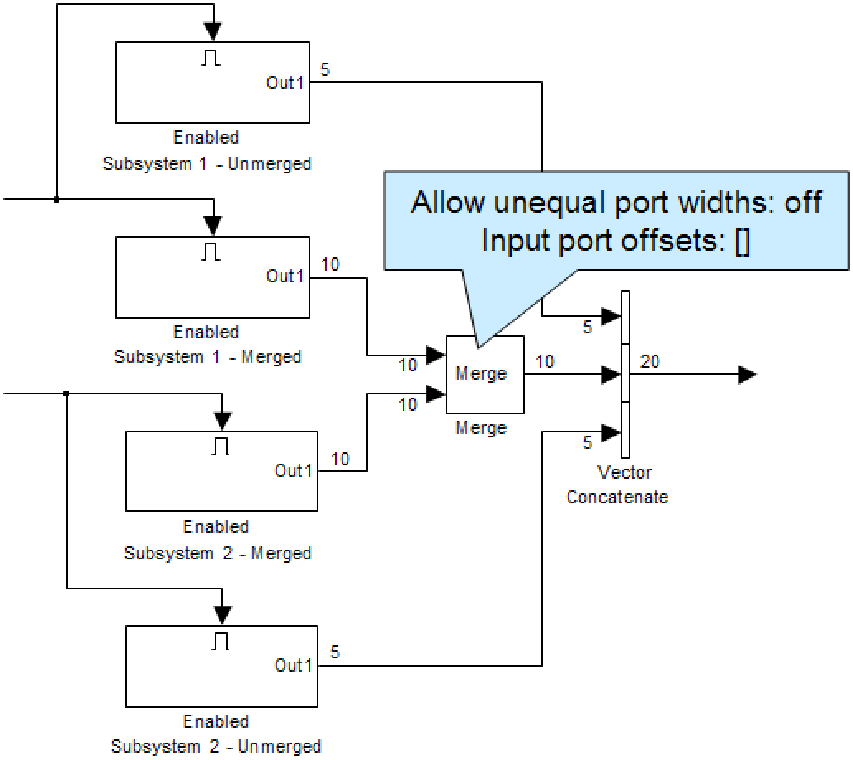
`max(2+0,2+1)=3`

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 shown in the scope output:


Scope
2
1
0
4
3
2
1
6
4
2
0
-2
0
2
4
6
8
10
Time offset: 0

If you use Simplified Initialization Mode, you must clear the **Allow unequal port widths** check box. The input port offsets for all input signals must be zero.

Consider using Merge blocks only for signal elements that require true merging. Other elements can be combined with merged elements using the Concatenate block, as shown in the following example.

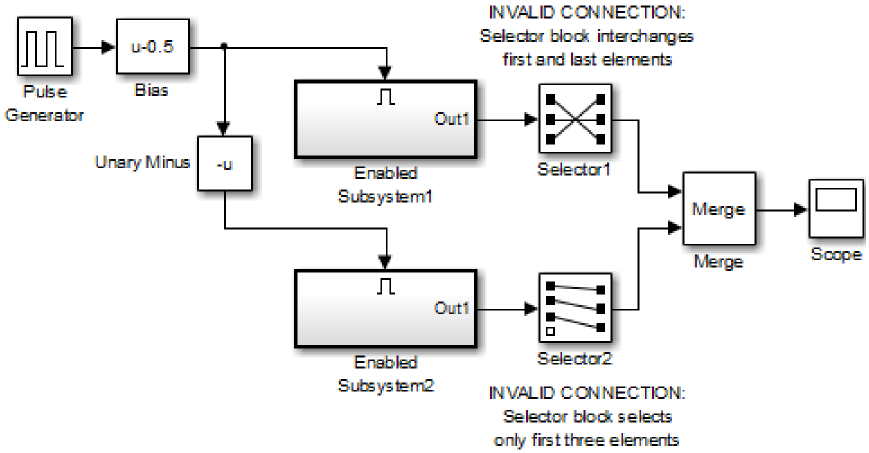

Out1
5
Enabled Subsystem 1 - Unmerged
Allow unequal port widths: off
Input port offsets: []
Out1
10
Enabled Subsystem 1 - Merged
10
10
Merge
Merge
10
5
20
5
Vector Concatenate
Out1
10
Enabled Subsystem 2 - Merged
Out1
5
Enabled Subsystem 2 - Unmerged

For more information on simplified initialization mode, see “Underspecified initialization detection”.

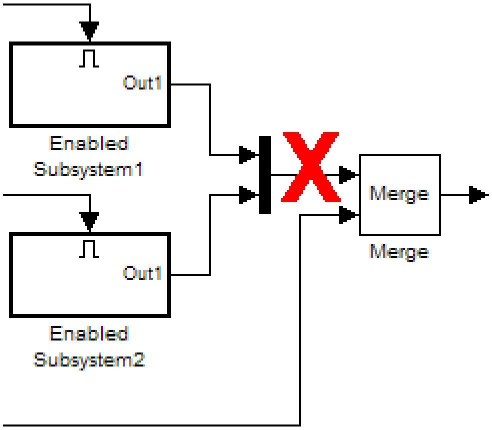
Combining or Reordering of Input Signals

A Merge block does not accept input signals whose elements have been reordered or partially selected. In addition, you should not connect input signals to the Merge block that have been combined outside of a conditionally-executed subsystem.

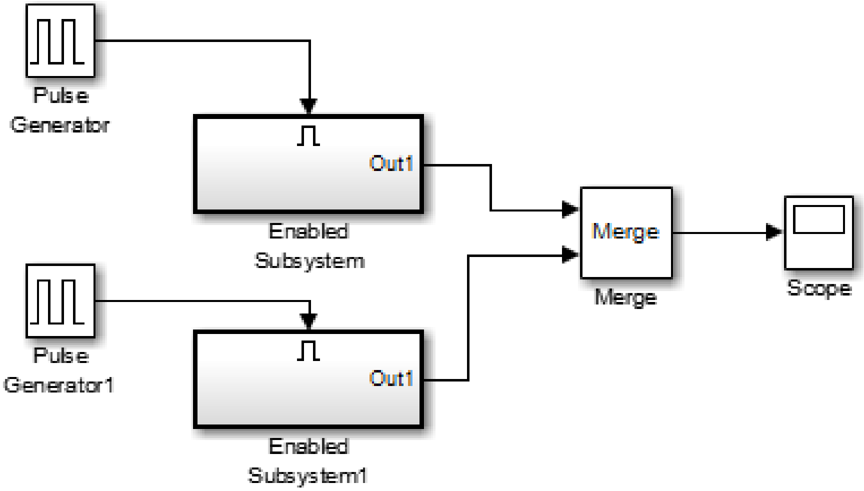
For example, in the following block diagram, the Merge block does not accept the output of the first Selector block because the Selector block interchanges the first and last elements of the vector signal. Similarly, the Merge block does not accept the output of the second Selector block because the Selector block selects only the first three elements.


INVALID CONNECTION:
Selector block interchanges
first and last elements
Pulse Generator
u-0.5
Bias
Unary Minus
-u
Out1
Enabled Subsystem1
Selector1
Merge
Merge
Scope
Out1
Enabled Subsystem2
Selector2
INVALID CONNECTION:
Selector block selects
only first three elements

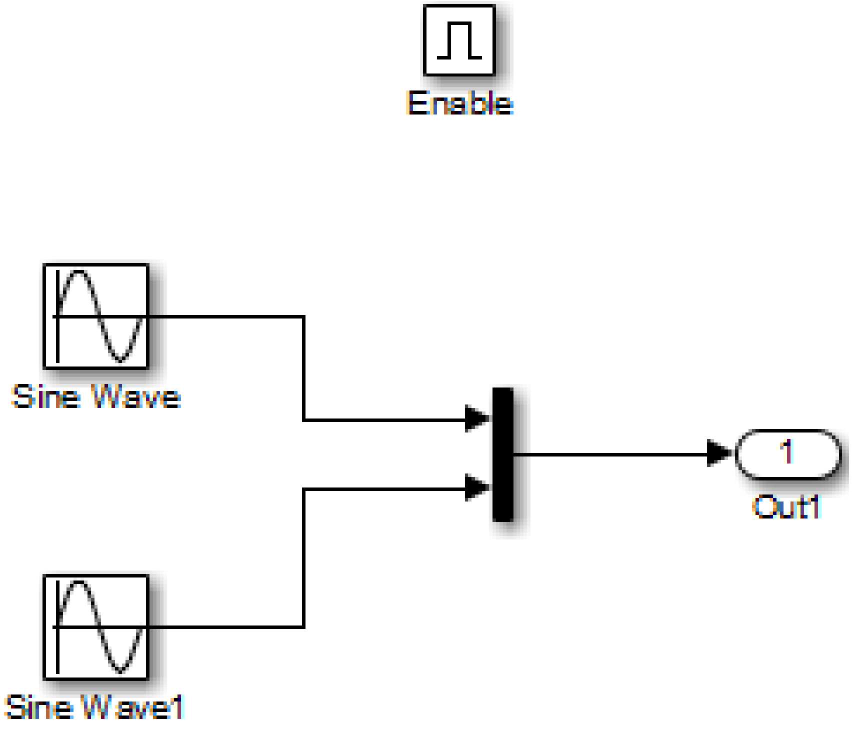
If you use simplified initialization mode, the following arrangement is *not allowed* because two signals are being combined outside of a conditionally-executed subsystem.


Out1
Enabled Subsystem1
Out1
Enabled Subsystem2
Merge
Merge

You can, however, combine or reorder Merge block input signals within a conditionally-executed subsystem. For example, the following model is valid.


Pulse Generator
Enabled Subsystem
Out1
Merge
Merge
Scope
Pulse Generator1
Enabled Subsystem1
Out1

Each Enabled Subsystem contains the following blocks.


Enable
Sine Wave
1
Out1
Sine Wave1

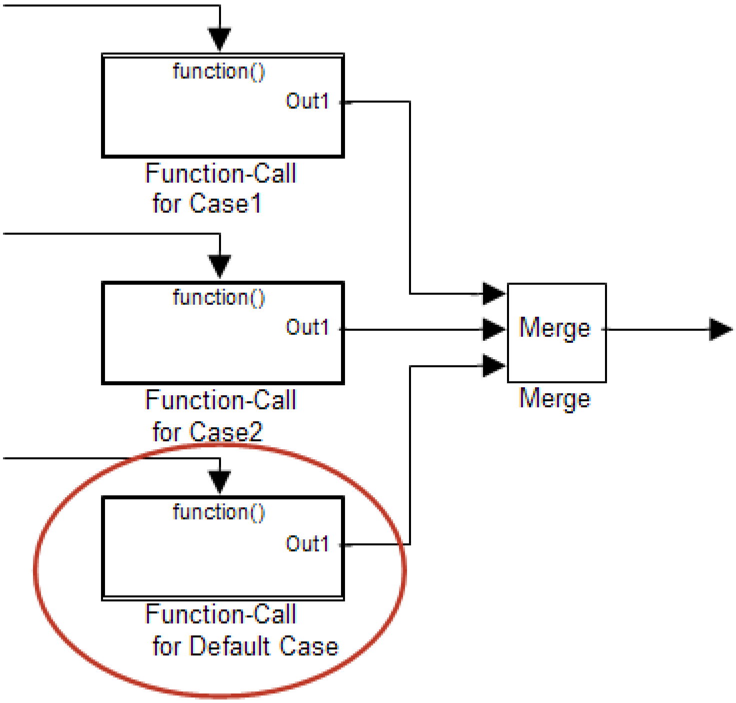
Conditionally-Executed Subsystem Outport Reset

The Outports of conditionally-executed subsystems being merged should not reset when disabled. This action can cause multiple subsystems to update the Merge block at the same time. Specifically, the disabled subsystem updates the Merge block by resetting its output, while the enabled subsystem updates the Merge block by computing its output.

To prevent this behavior, set the Outport block parameter **Output when disabled** to `held` for each conditionally-executed subsystem being merged.

Note If you are using Simplified Initialization Mode, you *must* set the Outport block parameter **Output when disabled** to `held`.

Instead of resetting the subsystem output when it is disabled, add an additional subsystem for the default case, and use control logic to run this subsystem if nothing else runs. For example, see the following block layout:


function()
Out1
Function-Call for Case1
function()
Out1
Function-Call for Case2
function()
Out1
Function-Call for Default Case
Merge
Merge

For more information on simplified initialization mode, see “Underspecified initialization detection”.

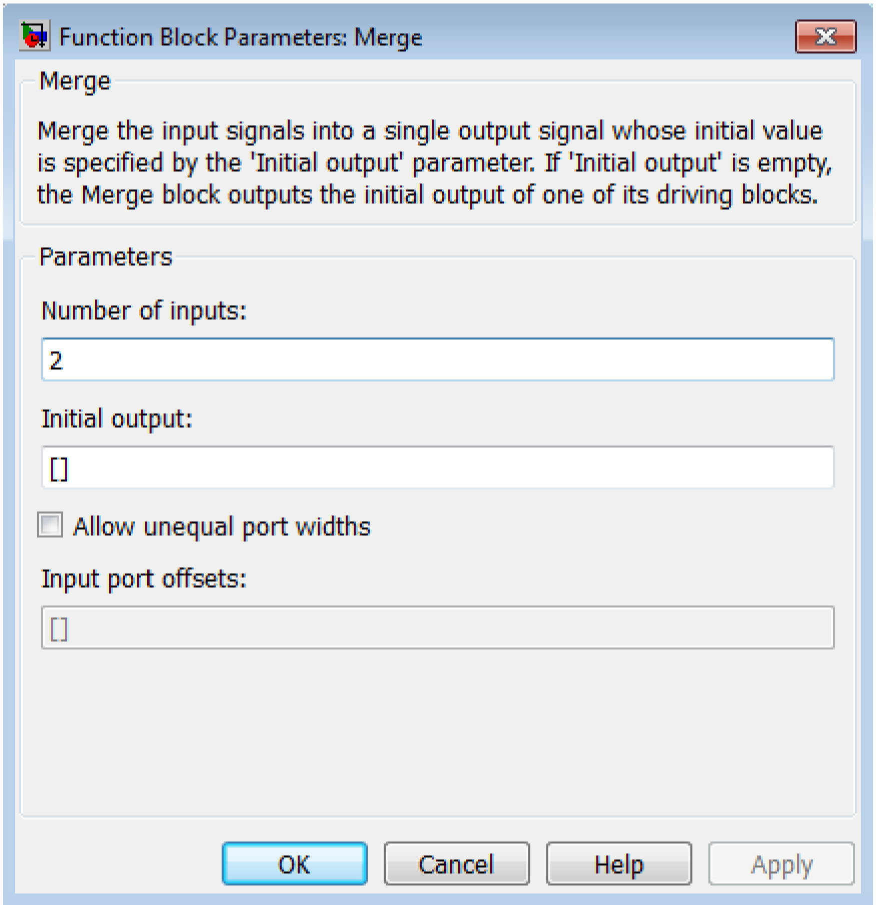
Merging S-Function Outputs

The Merge block can merge a signal from an S-Function block only if the memory used to store the S-Function block’s output is reusable. Simulink software displays an error message if you attempt to update or simulate a model that connects a nonreusable port of an S-Function block to a Merge block. See `ssSetOutputPortOptimOpts` for more information.

Data Type Support

The Merge block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. All inputs must be of the same data type and numeric type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Merge

Merge the input signals into a single output signal whose initial value is specified by the 'Initial output' parameter. If 'Initial output' is empty, the Merge block outputs the initial output of one of its driving blocks.

Parameters

Number of inputs: 2

Initial output: []

☐ Allow unequal port widths

Input port offsets: []

Parameters and Dialog Box

Number of inputs

Specify the number of input ports to merge.

Initial output

Specify the initial value of output. If unspecified, the initial output equals the initial output, if any, of one of the driving blocks. You cannot set the initial output of this block to `inf` or `NaN`.

Allow unequal port widths
Select this check box to allow the block to accept inputs having different numbers of elements.

Input port offsets
Enter a vector to specify the offset of each input signal relative to the beginning of the output signal.

Bus Support

The Merge block is a bus-capable block. The inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- The number of inputs must be greater than one.
- **Initial output** must be zero, a nonzero scalar, or a finite numeric structure.
- **Allow unequal port widths** must be disabled.
- All inputs to the merge must be buses and must be equivalent (same hierarchy with identical names and attributes for all elements).

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Merge block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Composite Signals” and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Merge block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”. Using an array of buses with a Merge block involves these limitations:

- **Allow unequal port widths** — Clear this parameter.
- **Number of inputs** — Set to a value of 2 or greater.

- **Initial condition** — Only this parameter (which might be, but does not have to be, a structure) is scalar-expanded to match the dimensions of the array of buses.

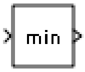
Characteristics

Bus-capable	Yes, with restrictions as noted above
Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose Output minimum or maximum input value

Library Math Operations

Description The MinMax block outputs either the minimum or the maximum element or elements of the inputs. You can choose the function to apply by selecting one of the choices from the **Function** parameter list.



If the block has one input port, the input must be a scalar or a vector. The block outputs a scalar equal to the minimum or maximum element of the input vector.

If the block has multiple input ports, all nonscalar inputs must have the same dimensions. The block expands any scalar inputs to have the same dimensions as the nonscalar inputs. The block outputs a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

The MinMax block ignores any input value that is `NaN`, except when every input value is `NaN`. When all input values are `NaN`, the output is `NaN`, either as a scalar or the value of each output vector element.

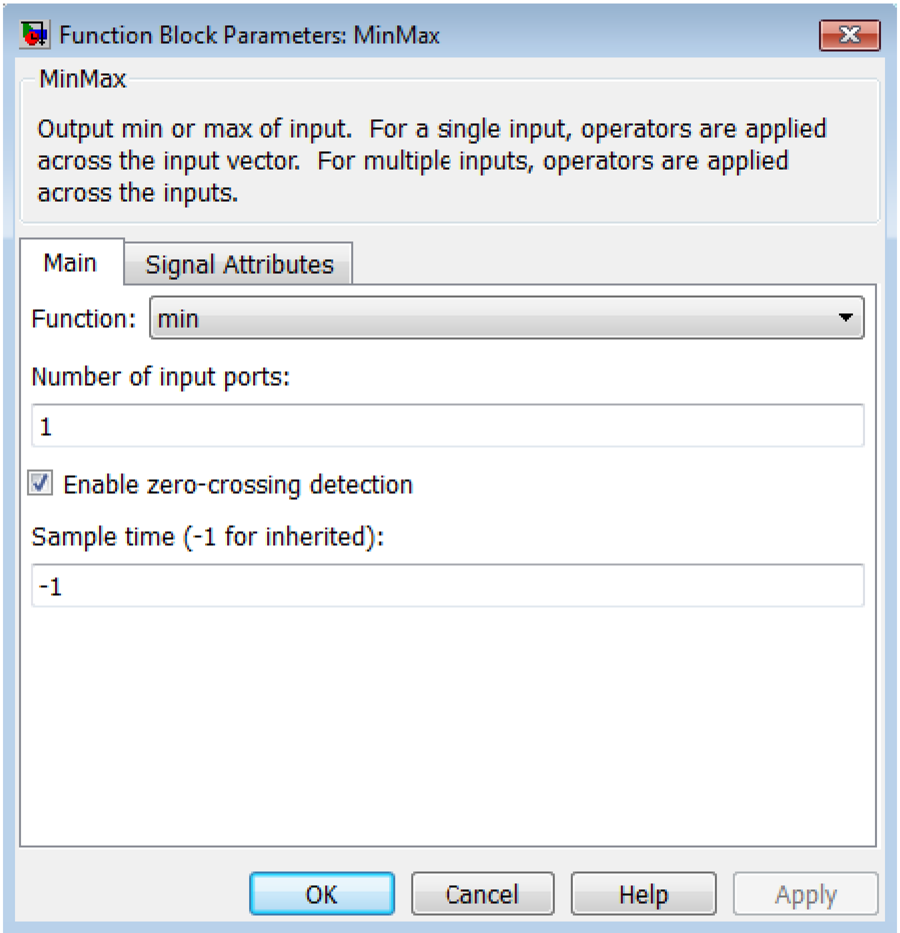
Data Type Support The MinMax block accepts and outputs real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the MinMax block dialog box appears as follows:



Function Block Parameters: MinMax

MinMax

Output min or max of input. For a single input, operators are applied across the input vector. For multiple inputs, operators are applied across the inputs.

Function: min

Number of input ports: 1

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

Function

Specify whether to apply the function `min` or `max` to the input.

Number of input ports

Specify the number of inputs to the block.

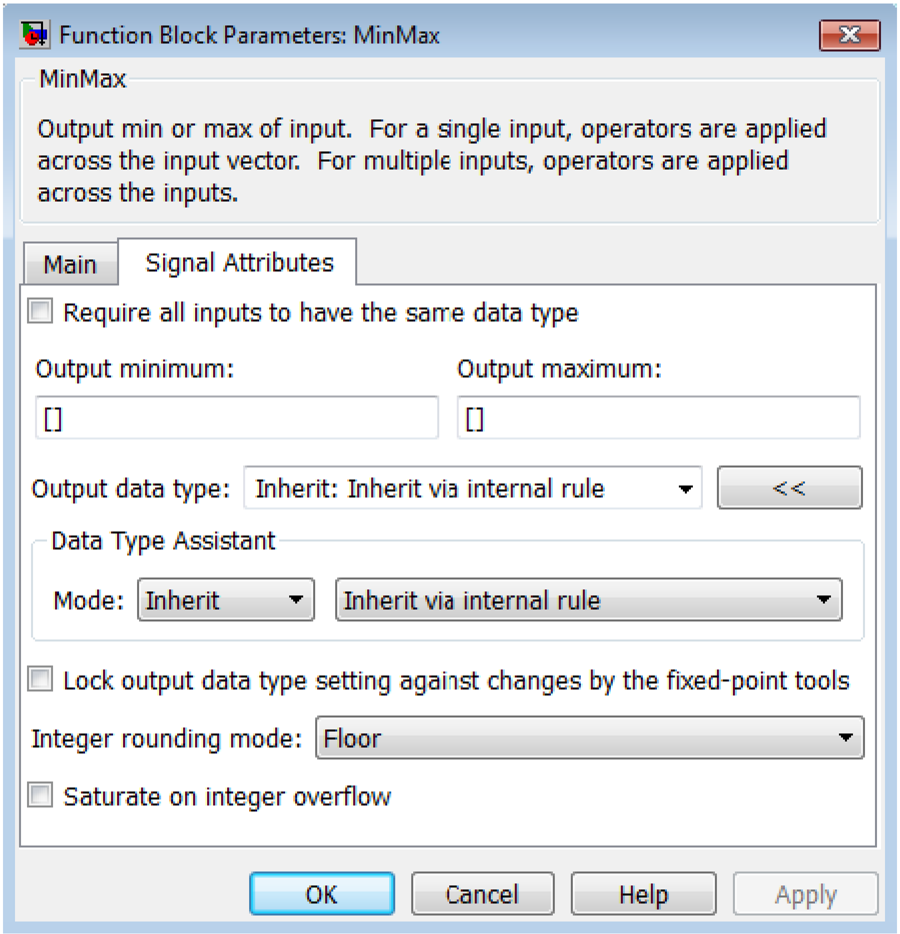
Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

The **Signal Attributes** pane of the MinMax block dialog box appears as follows:



Function Block Parameters: MinMax

MinMax

Output min or max of input. For a single input, operators are applied across the input vector. For multiple inputs, operators are applied across the inputs.

☐ Require all inputs to have the same data type

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

Data Type Assistant

Mode: Inherit — Inherit via internal rule

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate on integer overflow

Require all inputs to have the same data type

Select this check box to require that all inputs have the same data type.

Output minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block should output. The default value is`[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Block Output Data Types” in the *Simulink User’s Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate on integer overflow

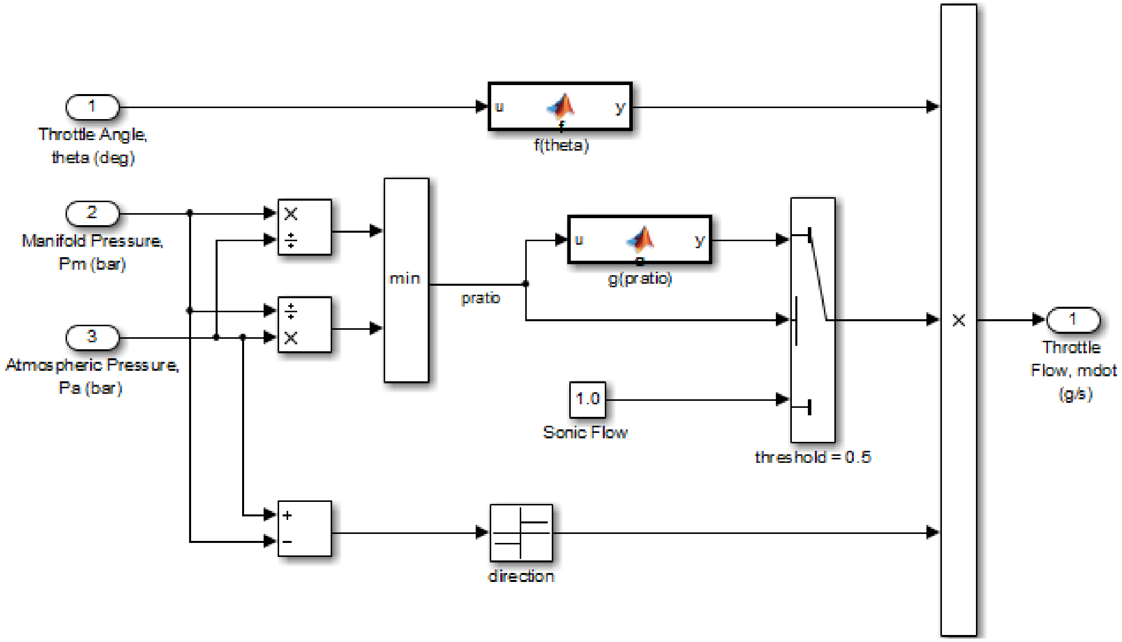
Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 “Checking 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.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

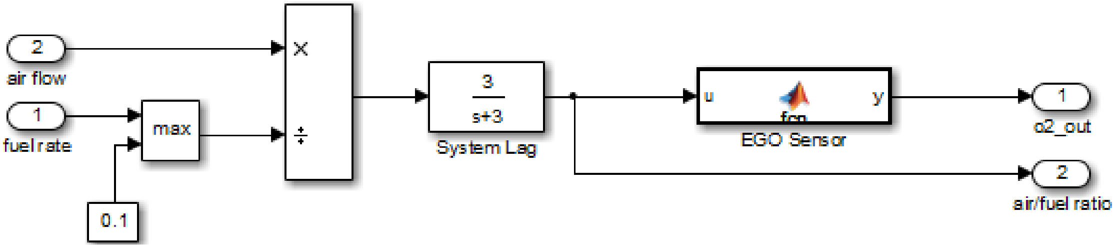
Examples

The `sldemo_fuelsys` model shows how to use the MinMax block.

In the `Engine Gas Dynamics/Throttle & Manifold/Throttle` subsystem, the MinMax block uses the `min` operator:


1
Throttle Angle, theta (deg)
2
Manifold Pressure, Pm (bar)
3
Atmospheric Pressure, Pa (bar)
u
y
f
f(theta)
min
pratio
u
y
g
g(pratio)
1.0
Sonic Flow
threshold = 0.5
direction
1
Throttle Flow, mdot (g/s)

In the `Engine Gas Dynamics/Mixing & Combustion` subsystem, the MinMax block uses the `max` operator:


2
air flow
1
fuel rate
max
0.1
3
s+3
System Lag
u
y
fcn
EGO Sensor
1
o2_out
2
air/fuel ratio

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of the inputs
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled

See Also

MinMax Running Resettable

Purpose

Determine minimum or maximum of signal over time

Library

Math Operations

Description

The MinMax Running Resettable block outputs the minimum or maximum of all past inputs u. You specify whether the block outputs the minimum or the maximum with the **Function** parameter.

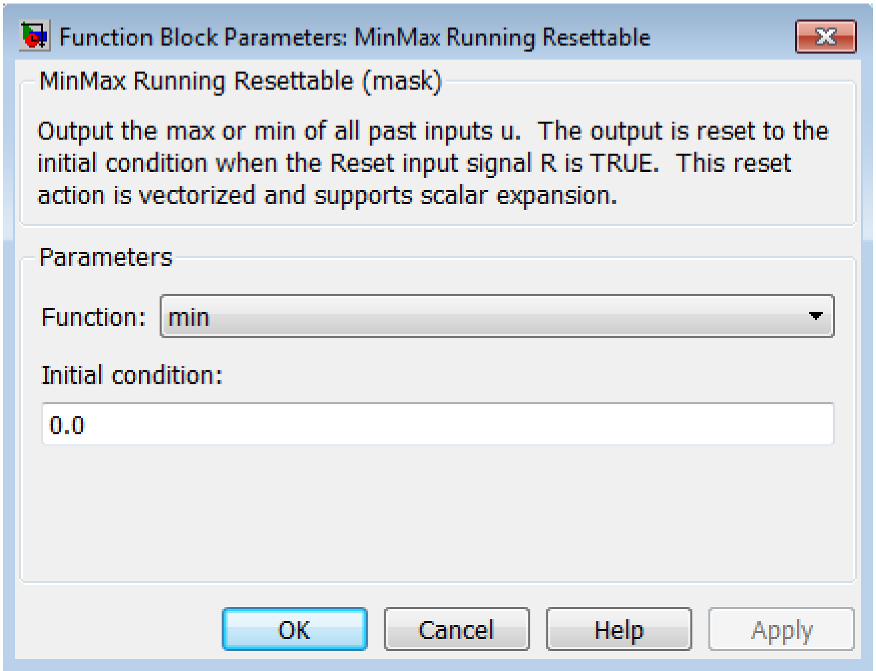
The block can reset its state based on an external reset signal R. When the reset signal R is TRUE, the block resets the output to the value of the **Initial condition** parameter.

The input can be a scalar, vector, or matrix signal. If you specify a scalar **Initial condition** parameter, the block expands the parameter to have the same dimensions as a nonscalar input. The block outputs a signal having the same dimensions as the input. Each output element equals the running minimum or maximum of the corresponding input elements.

Data Type Support

The MinMax Running Resettable block accepts and outputs real signals of any numeric data type that Simulink supports, except `Boolean`. The MinMax Running Resettable block supports fixed-point data types.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.


Function Block Parameters: MinMax Running Resettable

MinMax Running Resettable (mask)

Output the max or min of all past inputs u. The output is reset to the initial condition when the Reset input signal R is TRUE. This reset action is vectorized and supports scalar expansion.

Parameters

Function: min

Initial condition:

0.0

Parameters and Dialog Box

Function

Specify whether the block outputs the minimum or the maximum.

Initial condition

Specify initial condition.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

MinMax

Model, Model Variants

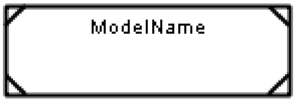
Purpose

Include model as block in another model

Library

Ports & Subsystems

Description



The Model block allows you to include a model as a block in another model. The included model is called a *referenced model*, and the model containing it (via the Model block) is called the *parent model*.

The Model block displays input ports 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 Reference" for more information.

A Model block can specify the referenced model:

- Statically, as a Model block parameter value, which must name the model literally
- Dynamically, depending on base workspace values

A Model Variants block is a Model block with variants enabled. The Model block parameter dialog box contains the **Enable Variants** button by default. If you click the **Enable Variants** button, the Model Variants block parameter dialog opens. The Model Variants block parameter dialog contains the **Disable Variants** button by default. Therefore, you can use either the Model block or the Model Variants block for implementing model variants. For more information about how to specify a referenced model for multiple specifications, see "Set Up Model Variants".

By default, the contents of a referenced model are user-visible, but you can hide the contents as described in “Protected Model”.

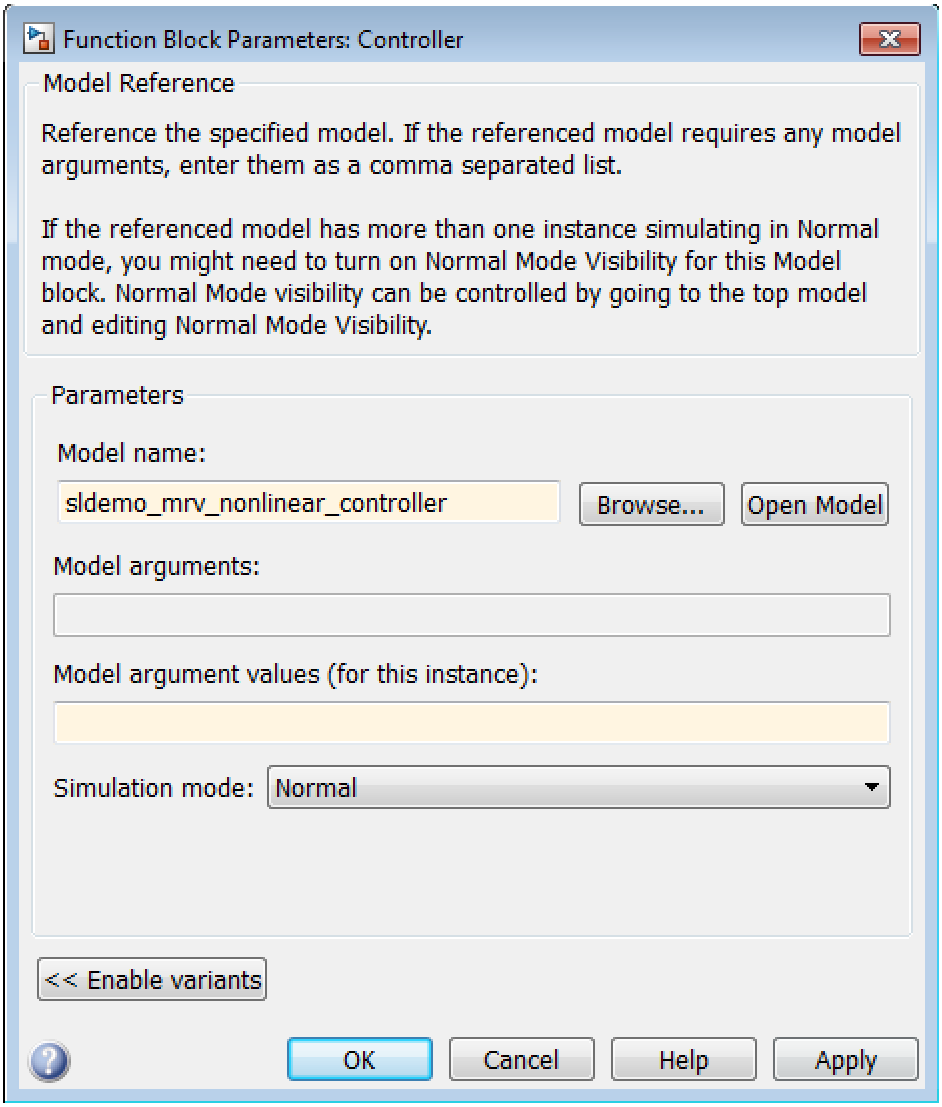
A signal that connects to a Model block is functionally the same signal outside and inside the block. A given signal can have at most one associated signal object, so the signal connected to the Model block cannot have a signal object in both the parent and the referenced models. For more information, see `Simulink.Signal`.

The Model block supports signal label propagation. For details specific to model referencing and model variants, see:

- “Processing for Referenced Models”
- “Processing for Variants and Configurable Subsystems”

Data Type Support

Determined by the root-level inputs and outputs of the model referenced by the Model block.



Function Block Parameters: Controller

Model Reference

Reference the specified model. If the referenced model requires any model arguments, enter them as a comma separated list.

If the referenced model has more than one instance simulating in Normal mode, you might need to turn on Normal Mode Visibility for this Model block. Normal Mode visibility can be controlled by going to the top model and editing Normal Mode Visibility.

Parameters

Model name: sldemo_mrv_nonlinear_controller

Model arguments:

Model argument values (for this instance):

Simulation mode: Normal

Browse...
Open Model
<< Enable variants
OK
Cancel
Help
Apply

Parameters and Dialog Box

- "Model name" on page 1-994
- "Model arguments" on page 1-995

- “Model argument values (for this instance)” on page 1-996
- “Simulation mode” on page 1-997
- “Enable variants” on page 1-999
- “Variant choices” on page 1-1001
- “Variant control” on page 1-1004
- “Condition (read only)” on page 1-1005
- “Model name” on page 1-1006
- “Model name” on page 1-1007
- “Model arguments” on page 1-1008
- “Model argument values (for this instance)” on page 1-1009
- “Simulation mode” on page 1-1010
- “Override variant conditions and use following variant” on page 1-1012
- “Variant” on page 1-1013
- “Generate preprocessor conditionals” on page 1-1014
- “Disable variants” on page 1-1014

Model name

Name of the model this block references.

Settings

Default: `<Enter Model Name>`

The value must be a valid MATLAB identifier.

The extension, for example, `.SLX`, is optional.

Tips

- To navigate to the model that you want to reference from this block, use the **Browse** button to the right of the **Model name** parameter.
- To confirm that the model you specify is the one you intended, you can use the **Open Model** button to the right of the **Model name** parameter.

Command-Line Information

Parameter: `ModelNameDialog`
Type: string
Value: Any valid value
Default: The name of the referenced model exactly as you typed it in, with any surrounding white space removed. When you set `ModelNameDialog` programmatically or from the dialog, Simulink automatically sets the values of `ModelName` and `ModelFile` based on the value of `ModelNameDialog`.

Model arguments

Display model arguments accepted by the model referenced by this block.

Declaring a variable to be a model argument allows each instance of the model to use a different value for that variable.

Settings

Default: `''`

This is a read-only parameter that displays model arguments for the model referenced by this block. To create model arguments, refer to "Using Model Arguments".

Model argument values (for this instance)

Specify values to be passed as model arguments to the model referenced by this block each time the simulation invokes the model.

Settings

Enter the values in this parameter as a comma-separated list in the same order as the corresponding argument names in the **Model arguments** field.

Command-Line Information

Parameter: `ParameterArgumentValues`
Type: string
Value: Any valid value
Default: `''`

Simulation mode

Set the simulation mode for the model referenced by this block. This setting specifies whether to simulate the model by generating and executing code or by interpreting the model in Simulink.

Settings

Default: `Accelerator`

`Accelerator`
: Creates a MEX-file for the sub model, then executes the sub model by running the S-function.

`Normal`
: Executes the sub model interpretively, as if the sub model were an atomic subsystem implemented directly within the parent model.

`Software-in-the-loop (SIL)`
: This option requires the Embedded Coder software. Generates production code using model reference target for the sub model. This code is compiled for, and executed on, the host platform.

`Processor-in-the-loop (PIL)`
: This option requires the Embedded Coder software. Generates production code using model reference target for the sub model. This code is compiled for, and executed on, the target platform. A documented target connectivity API supports exchange of data between the host and target at each time step during the PIL simulation.

Command-Line Information

Parameter: `SimulationMode`
Type: string
Value: `'Accelerator' | 'Normal' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'`
Default: `'Accelerator'`

See Also

- "Using Model Arguments"
- "Choosing a Simulation Mode"

Model, Model Variants

- “Overview of the Target Connectivity API”
- “Numerical Equivalence Testing”

Enable variants

Enables variants and opens the Model Variants block parameter dialog box, which is hidden by default. The Model Variants block parameter dialog is the default block parameter dialog for the Model Variants block.

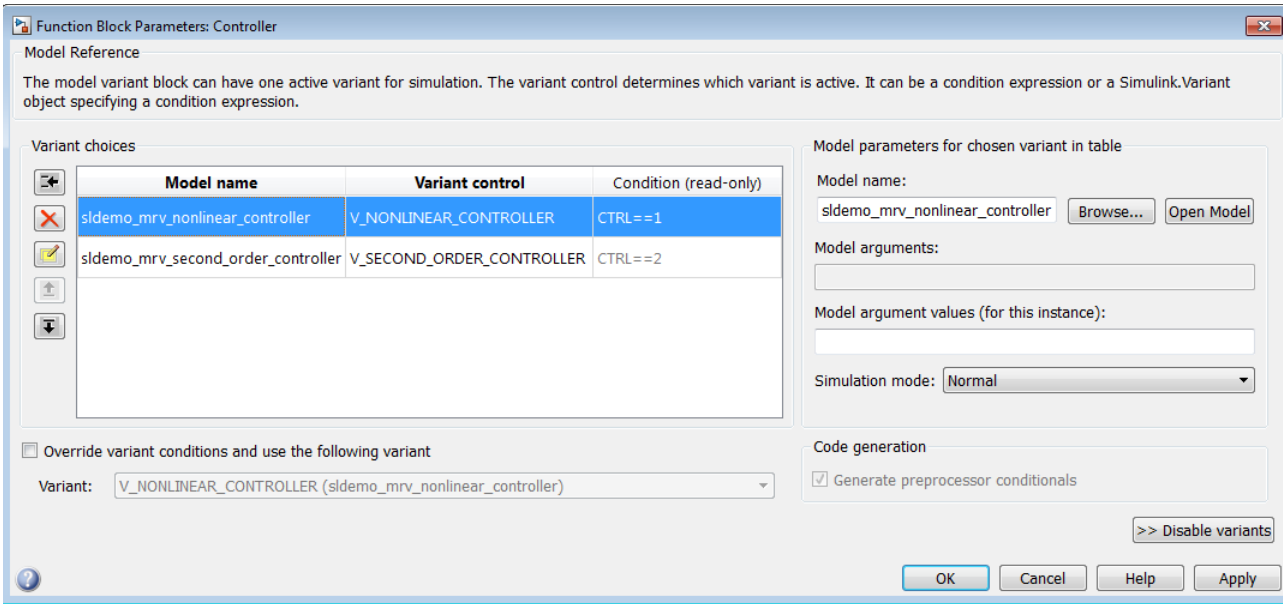
Settings

Default: Disabled

Dependencies

This button enables the Model Variants Sections, which include: **Variant choices** table, **Model parameters for the chosen variant in table** section, parameters to override variants, and a **Code generation** section.

The following example shows the Model variants options from the example model `sldemo_mdlref_variants`.



Function Block Parameters: Controller

Model Reference

The model variant block can have one active variant for simulation. The variant control determines which variant is active. It can be a condition expression or a Simulink.Variant object specifying a condition expression.

Variant choices

Model name	Variant control	Condition (read-only)
sldemo_mrv_nonlinear_controller	V_NONLINEAR_CONTROLLER	CTRL==1
sldemo_mrv_second_order_controller	V_SECOND_ORDER_CONTROLLER	CTRL==2

Model parameters for chosen variant in table

Model name: sldemo_mrv_nonlinear_controller

Model arguments:

Model argument values (for this instance):

Simulation mode: Normal

☐ Override variant conditions and use the following variant

Variant: V_NONLINEAR_CONTROLLER (sldemo_mrv_nonlinear_controller)

Code generation

☑ Generate preprocessor conditionals

See Also

"Set Up Model Variants"

Variant choices

Displays a table of variant choices, associated model names, variant controls, and conditions. 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.

Settings

Default: The table has a row for each variant object in the base workspace. The **Variant choices** table includes the **Model name**, its associated **Variant control**, and **Condition (read-only)** columns.

Use the **Add a new variant** button to add a new row to the table. See the description of the **Model name**, **Variant control,** and **Condition (read-only)** table columns for information about how to set values for table rows.

Tips

You can use buttons to the left of the **Variant choices** table to modify the table.

Function	Button
Add a new variant: Add a new, empty row below the currently selected row	⇥
Delete selected variant: Delete the currently selected row. (Models and objects are not affected.)	✕
Create/Edit selected variant object: Creates a `Simulink.Variant` object in the base workspace and opens the `Simulink.Variant` object parameter dialog in order to specify the variant **Condition** . This button will only be enabled for valid `Simulink.Variant` objects.	✎

Function	Button
Move variant up: Move the currently selected row up one slot in the table	↑
Move variant down: Move the currently selected row down one slot in the table	↓

Dependency

Enable variants enables this parameter.

Command-Line Information

Parameter: `Variants`
Type: array
Value: array of variant structures where each element specifies one variant. The structure fields are:

- `variant.Name` (string) — The variant control can be a boolean condition expression, or a `Simulink.Variant` object representing a boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.
- `variant.ModelName` (string) — The name of the referenced model associated with the specified variant control in the Model block.
- `variant.ParameterArgumentNames` (string) — Read-only string containing the names of the model arguments for which the Model block must supply values.
- `variant.ParameterArgumentValues` (string) — The values to supply for the model arguments when this variant is the active variant.
- `variant.SimulationMode` (string) — The execution mode to use when this variant is the active variant.
  - Possible values are `'Accelerator' | 'Normal' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'`

See Also

“Configure the Model Variants Block”

Variant control

Displays the variant controls in the base workspace. The variant control can be a boolean condition expression, or a `Simulink.Variant` object representing a boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.

Settings

Default: `Variant1`

To enter a variant control name, double-click **Variant control** column in a new row and enter the variant control expression.

Dependency

Enable variants enables this parameter.

Command-Line Information

Structure field: Represented by the `variant.Name` field in the `Variants` parameter structure
Type: string
Value: Variant control associated with the variant.
Default: `''`

See Also

- `Simulink.Variant`

Condition (read only)

Display the condition for the `Simulink.Variant` object.

Settings

This read-only field displays the condition for the associated model variant in the base workspace. Click the **Edit selected variant object** button to specify the condition for the selected variant object.

Tips

The variant condition must be a Boolean expression that references at least one base workspace variable or parameter. For example, `FUEL== 2 && EMIS == 1`. Do not surround the condition with parentheses or single quotes. The expression can include:

- MATLAB variables defined in the base workspace
- Simulink parameter objects defined in the base workspace
- Scalar variables
- Enumerated values
- Operators `==`, `~=`, `&&`, `||`, `~`
- Parentheses for grouping

Dependency

Enable variants enables this parameter.

See Also

"Configure the Model Variants Block"

Model name

Display or enter the name of the model associated with the variant control in the **Variant choices** table.

Settings

Default: `''`

Double-click the table cell to enter a model name. The name must be a valid MATLAB identifier.

The extension, for example, `.SLX`, is optional.

Tips

You can type the model name into the table, or you can use the **Model parameters for chosen variant in table** controls to find and open models:

- To navigate to the model that you want to reference for the selected variant in the table, click the **Browse** button to the right of the **Model name** edit box.
- To confirm that the model you specify is the one you intended, click the **Open Model** button to the right of the **Model name** edit box.

Dependency

Enable variants enables this parameter.

Command-Line Information

Structure field: represented by the `variant.ModelName` field in the `Variants` parameter structure
Type: string
Value: any valid value
Default: name of the referenced model exactly as you typed it, with any surrounding white space removed. When you set the model name programmatically or using the GUI, Simulink automatically sets the values of `ModelName` and `ModelFile` based on the value of `ModelNameDialog`.

See Also

“Set Up Model Variants”

Model name

Display or enter the name of the model associated with the selected variant object in the **Variant choices** table.

Settings

Default: `''`

Must be a valid MATLAB identifier.

The extension, for example, `.SLX`, is optional.

Tips

You can type the model name in either the table cell or the **Model name** edit box for the selected variant in the table. Use the controls to find and open models as follows:

- To navigate to the model that you want to reference from this block, use the **Browse** button to the right of the **Model name** parameter.
- To confirm that the model you specify is the one you intended, you can use the **Open Model** button to the right of the **Model name** parameter.

Dependency

Enable variants enables this parameter.

Command-Line Information

Structure field: represented by the `variant.ModelName` field in the `Variants` parameter structure
Type: string
Value: any valid value
Default: name of the referenced model exactly as you typed it, with any surrounding white space removed. When you set the model name programmatically or using the dialog, Simulink automatically sets the values of `ModelName` and `ModelFile` based on the value of `ModelNameDialog`.

See Also

"Set Up Model Variants"

Model arguments

Display model arguments for the variant control highlighted in the **Variant choices** table.

Declaring a variable to be a model argument allows each instance of the model to use a different value for that variable.

Settings

Default: `''`

This is a read-only parameter that displays model arguments for the variant control highlighted in the **Variant choices** table. To create model arguments, refer to "Using Model Arguments".

Dependency

Enable variants enables this parameter.

Command-Line Information

Structure field: Represented by the `variant.ParameterArgumentNames` field in the `Variants` parameter structure`OneArgName`
Type: string
Value: Enter model arguments as a comma separated list
Default: `''`

See Also

- "Using Model Arguments"
- "Set Up Model Variants"

Model argument values (for this instance)

Specify values to be passed as model arguments for the model variant control highlighted in the **Variant choices** table, each time the simulation invokes the model.

Settings

Enter the values in this parameter as a comma-separated list in the same order as the corresponding argument names in the **Model arguments** field.

Dependency

Enable variants enables this parameter.

Command-Line Information

Structure field: Represented by the `variant.ParameterArgumentValues` field in the `Variants` parameter structure`OneArgName`
Type: string
Value: Any valid value
Default: `''`

See Also

- "Using Model Arguments"
- "Set Up Model Variants"

Simulation mode

Set the simulation mode for the model variant control highlighted in the **Variant choices** table. This setting specifies whether to simulate the model by generating and executing code or by interpreting the model in Simulink.

Settings

Default: `Accelerator`

`Accelerator`
: Creates a MEX-file for the sub model and then executes the sub model by running the S-function.

`Normal`
: Executes the sub model interpretively, as if the sub model were an atomic subsystem implemented directly within the parent model.

`Software-in-the-loop (SIL)`
: This option requires the Embedded Coder software. Generates production code using model reference target for the sub model. This 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 using model reference target for the sub model. This code is compiled for, and executed on, the target platform. A documented target connectivity API supports exchange of data between the host and target at each time step during the PIL simulation.

Dependency

Enable variants enables this parameter.

Command-Line Information

Structure field: Represented by the `variant.SimulationMode` field in the `Variants` parameter structure

Type: string

Value: `'Accelerator' | 'Normal'| 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'`

Default: `'Accelerator'`

See Also

- "Using Model Arguments"
- "Choosing a Simulation Mode"
- "Numerical Equivalence Testing"
- "Set Up Model Variants"

Override variant conditions and use following variant

Specify whether to override the variant conditions and make the specified **Variant** parameter the active variant.

Settings

Default: Off

☑ On

Override the variant conditions and set the active variant to the value of the **Variant** .

☐ Off

Determine the active variant by the value of the variant conditions.

Tip

Both this GUI parameter and the **Variant** GUI parameter (following) use the same API parameter, `OverrideUsingVariant`.

Dependencies

Enable variants enables this parameter.

This parameter enables variants.

Command-Line Information

Parameter: `OverrideUsingVariant`
Type: string
Value: `''` if no overriding variant control is specified.
Default: `''`

See Also

“Override Variants”

Variant

Specify the variant control associated with the model to use if you select **Override variant conditions and use the following variant**.

Settings

Default: `''`

Must be a valid non empty or non commented name.

Tips

- You can use the **Variant** drop down to view a list of all variant controls currently available and their associated models.
- Both this GUI parameter and the **Override variant conditions and use following variant** GUI parameter (above) use the same API parameter, `OverrideUsingVariant`.

Dependencies

Enable variants and **Override variant conditions and use the following variant** enable this parameter.

Command-Line Information

Parameter: `OverrideUsingVariant`
Type: string (read-only)
Value: Variant control

See Also

- “Override Variants”
- `Simulink.Variant`

Generate preprocessor conditionals

Control preprocessor conditionals. This check box is relevant only to code generation, and has no effect on the behavior of a model in Simulink. See "Generate Preprocessor Conditionals for Variant Systems" for more information.

Settings

Default: Disabled

Dependencies

- The check box is available for generating only ERT targets.
- **Override variant conditions and use following variant** is cleared (`'off'`)
- **Model Configuration Parameters > Optimization > Inline parameters** is selected (`'on'`)
- **Model Configuration Parameters > Code Generation > Interface > Generate preprocessor conditionals** is set to `Use local settings`.

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`
Type: string
Value: `'off'` | `'on'`
Default: `'off'`

See Also

"Variant Systems"

Disable variants

Disable model reference variants and hide the Model Variants Section. The block retains any information you have entered and approved by clicking **Apply** or **OK**.

Command-Line Information

Parameter: `Variant`
Type: string
Value: `'off'` | `'on'`

Default: `'off'`

Navigating a Model Block

- Double-clicking the prototype Model block in the **Ports & Subsystems** library opens its Block Parameters dialog box for inspection, but does not allow you to specify parameter values.
- Double-clicking an unresolved Model block opens its Block Parameters dialog box. You can resolve the block by specifying a **Model name**.
- Double-clicking a resolved Model block opens the model that the block references. You can also open the model by choosing **Open Model** from the **Context** or **Edit** menu.

To display the Block Parameters dialog box for a resolved Model block, choose **Model Reference Parameters** from the **Context** or **Edit** menu.

Model Blocks and Direct Feed through

When a Model block is part of a cycle, and the block is a direct feed through block, an algebraic loop can result. An algebraic loop in a model is not necessarily an error, but it may not give the expected results.

- "Algebraic Loops" for information about direct feed through and algebraic loops.
- "Highlighting Algebraic Loops Using the Algebraic Loop Diagnostic" for information about seeing algebraic loops graphically.
- "Display Algebraic Loop Information" for information about tracing algebraic loops in the debugger.
- The "Diagnostics Pane: Solver" pane "Algebraic loop" option for information on detecting algebraic loops automatically.

Direct Model Block Feed through Caused by Sub model Structure

A Model block may be a direct feed through block due to the structure of the referenced model. Where direct feed through results from sub model structure, and causes an unwanted algebraic loop, you can:

- Automatically eliminate the algebraic loop using techniques described in:
  - “Minimize algebraic loop”
  - “Minimize algebraic loop occurrences”
  - “How Simulink Eliminates Artificial Algebraic Loops”
- Manually insert one or more Unit Delay blocks as needed to break the algebraic loop.

Direct Model Block Feed through Caused by Model Configuration

Embedded Real Time (`ert`) based targets provide the option **Model Configuration Parameters > Code Generation > Interface > Single output/update function**. This option controls whether generated code has separate output and update functions, or a combined output/update function. See:

- “Entry Point Functions and Scheduling” for information about separate and combined output and update functions.
- “Single output/update function” for information about specifying whether code has separate or combined functions.

When **Single output/update function** is enabled (default), a Model block has a combined output/update function. The function makes the block a direct feed through block for all inports, regardless of the structure of the referenced model. Where an unwanted algebraic loop results, you can:

- Disable **Single output/update function**. The code for the Model block then has separate output and update functions, eliminating the direct feed through and hence the algebraic loop.
- Automatically eliminate the algebraic loop using techniques described in:
  - “Minimize algebraic loop”

  - “Minimize algebraic loop occurrences”
  - “How Simulink Eliminates Artificial Algebraic Loops”
- Manually insert one or more Unit Delay blocks as needed to break the algebraic loop.

Characteristics

Direct Feed through	If “Single output/update function” is enabled (default), a Model block is a direct feed through block regardless of the structure of the referenced model. If “Single output/update function” is disabled, a Model block may or may not be a direct feed through block, depending on the structure of the referenced model.
Scalar Expansion	Depends on model referenced by this block.
Multi dimensional	Yes

Requirements, Limitations, and Tips for Model Variants

A Model Variants block and its referenced models must satisfy the requirements in “Simulink Model Referencing Requirements” and “Model Referencing Limitations”. You can nest Model Variants blocks to any level.

Note For information on requirements and limitations that apply to *code generation* see “Limitations on Generating Code for Variants”.

Tips

- A Model Variants block can log only those signals that the referenced model specifies as logged. If a model is a variant model, or contains a variant model, then you can either log all logged signals or log no logged signals. The Signal Logging Selector configuration for the model must be in one of these states:

- The Logging Mode is set to Log all signals as specified in model.
- The Logging Mode is set to **Override signals** and the check box for the model block is either checked (☑) or empty (☐). The check box cannot be filled (▣).

  For more information about logging referenced models, see “Models with Model Referencing: Overriding Signal Logging Settings”.

To enable logging programmatically, use the `DefaultDataLogging` parameter.

- You can enable or suppress warning messages about mismatches between a Model Variants block and its referenced model by setting diagnostics on the **Diagnostics Pane: Model Referencing**.
- During model compilation, Simulink evaluates variant objects before calling the `InitFcn` callback. Therefore, do not modify the condition of the variant object in the `InitFcn` callback.
- Each variant must have an associated variant control specified in the **Variant control** column. 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 variant controls as`Simulink.Parameter` objects.

See Also

- “Model Reference”
- “Set Up Model Variants”

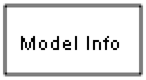
Purpose

Display model properties and text in model

Library

Model-Wide Utilities

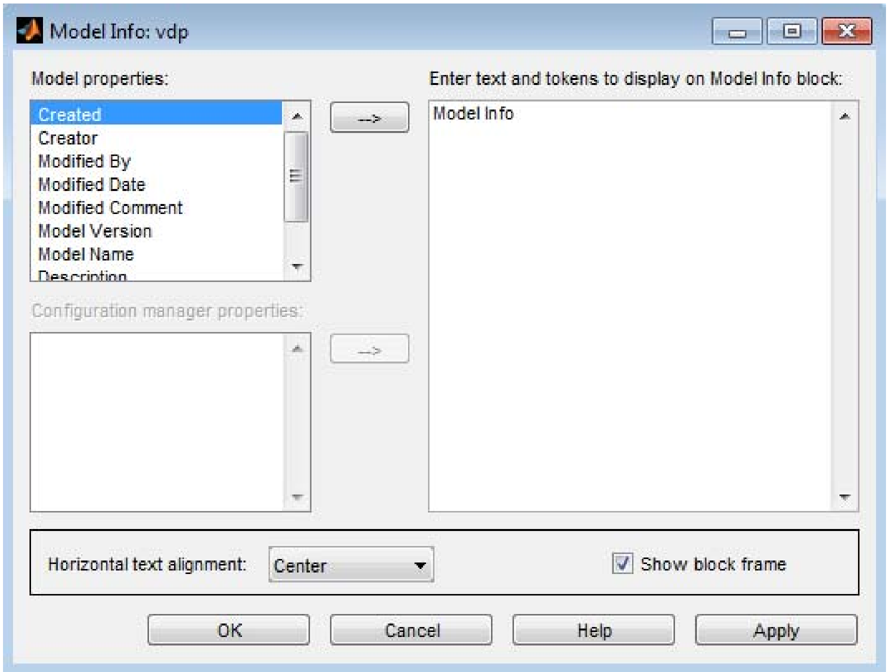
Description



The Model Info block displays model properties and text about a model on the mask of the block. Use the Model Info block dialog box to specify the content and format of the text that the block displays. You can select model properties to display on the block. In the text displayed on the block mask, Simulink replaces the property name with the current value of the property in the model.

Data Type Support

Not applicable.


Model Info: vdp
Model properties:
Created
Creator
Modified By
Modified Date
Modified Comment
Model Version
Model Name
Description
-->
Configuration manager properties:
-->
Enter text and tokens to display on Model Info block:
OK
Cancel
Help
Apply

Enter text and tokens to display on Model Info block: Model Info

Horizontal text alignment: Center

☑ Show block frame

Parameters and Dialog Box

Specify Text and Properties to Display

Use the **Enter text and tokens to display on Model Info block** edit box to specify the text and properties to display.

- In the edit box, enter any text you want to display on the block mask. Edit the default text `Model Info`.
- To display a model property on the block mask, select a property in the **Model properties** list and click the right arrow button.

  The block adds a token of the form `%<modelpropertyname>` to the edit box. In the text the block mask displays, Simulink will replace the token with the value of the property.

  1 For example, if you select `Model Version` in the **Model properties** list and click the right arrow button, then the token

```
%<ModelVersion>
```

appears in the right edit box.

2. You could add some explanatory text before the model property, e.g. "`Model version is:`".
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 like this example:

```
Model version is:
1.6
```

See "Version Information Properties" for descriptions of the model properties.

Caution

Using third-party source control tool keyword expansion within model properties tokens might corrupt your model files when you submit them. See "Register Model Files with Source Control Tools".

If you save your model in SLX format, third-party tools cannot perform keyword substitution. Any information in the model file from such third-party tool keyword substitution is cached when you first save the MDL file as SLX, and is never updated again. The Model Info block shows stale information from then on, so remove third-party tool keyword substitution from Model Info blocks to ensure up-to-date displays.

Configuration Manager Properties

The **Configuration manager properties** field is enabled only if you previously specified an external configuration manager for this model on the MATLAB **Preferences** dialog box for the model. The field lists version control information maintained by the external system that you can include in the Model Info block. To include an item from the list, select it and then click the adjacent arrow button.

Note The selected item does not appear in the Model Info block until you check the model in or out of the repository maintained by the configuration manager and you have closed and reopened the model.

If you save your model in SLX format, keyword substitution of version information is not available and you cannot add new configuration manager properties in the Model Info block. For existing Configuration manager properties, the block detects the problem, removes stale version information and instead displays: “Not available in SLX model file format”.

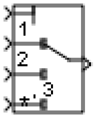
For a more flexible interface to source control tools, use the Simulink Project Tool instead of the Model Info block. See “Simulink Projects”.

Purpose Choose between multiple block inputs

Library Signal Routing

Description

Types of Block Inputs



The Multiport Switch block chooses among several inputs. The first input is the *control input*, while the others are *data inputs*. The value of the control input determines which data input passes to the output port.

Note This block is the same as the Index Vector block. If you set **Number of data ports** to `1`, you get the Index Vector block.

When you set **Data port for default case** to `Additional data port`, an extra data port with a `*` label appears below the other data inputs. For more information, see "How the Block Handles an Out-of-Range Control Input" on page 1-1025.

Rules That Determine the Block Output

You specify the number of data inputs with **Number of data ports**. Data inputs can be scalar or vector.

- If you specify only one data input and that input is a vector, the block behaves as an *index selector*, and not as a multiport switch. The block output is the input vector element whose index matches the control input.
- If you specify more than one data input, 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 the data inputs are scalar, the output is a scalar.

How the Block Interprets the Control Input

The following table summarizes how the block interprets the control input and passes data to the output.

Control Input	**Truncation**	**Setting for Data Port Order**	**Block Behavior During Simulation**	
			The specified data input passes to the output based on...	**And an out-of-range error occurs when...**
Integer value	None	`Zero-based contiguous`	Zero-based indexing	The control input is less than `0` or greater than the number of data inputs minus one
		`One-based contiguous`	One-based indexing	The control input is less than `1` or greater than the number of data inputs
		`Specify indices`	Indices you specify	The control input does not correspond to any specified data port index
Not an integer value	The block truncates the value to an integer by rounding to floor.	`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

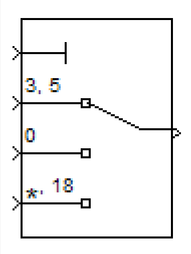
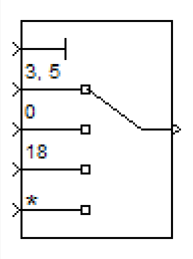
Control Input	Truncation	Setting for Data Port Order	Block Behavior During Simulation	
			The specified data input passes to the output based on...	And an out-of-range error occurs when...
				specified data port index

How the Block Handles an Out-of-Range Control Input

For a control input with an integer value, 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**.

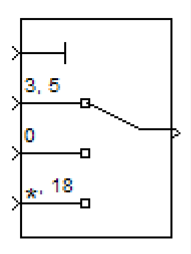
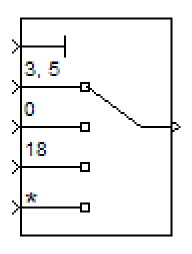
Behavior for Simulation

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.

Behavior for Code Generation

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.

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.
The enumerated type contains only valid enumerated values. Data input ports can never get invalid values	• Set **Data port for default case** to `Last data port`. • Set **Diagnostic for default case** to `None`.	This block configuration avoids unnecessary diagnostic action.

Scenario	What to Do	Rationale
of enumerated type.		
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.

Data Type Support

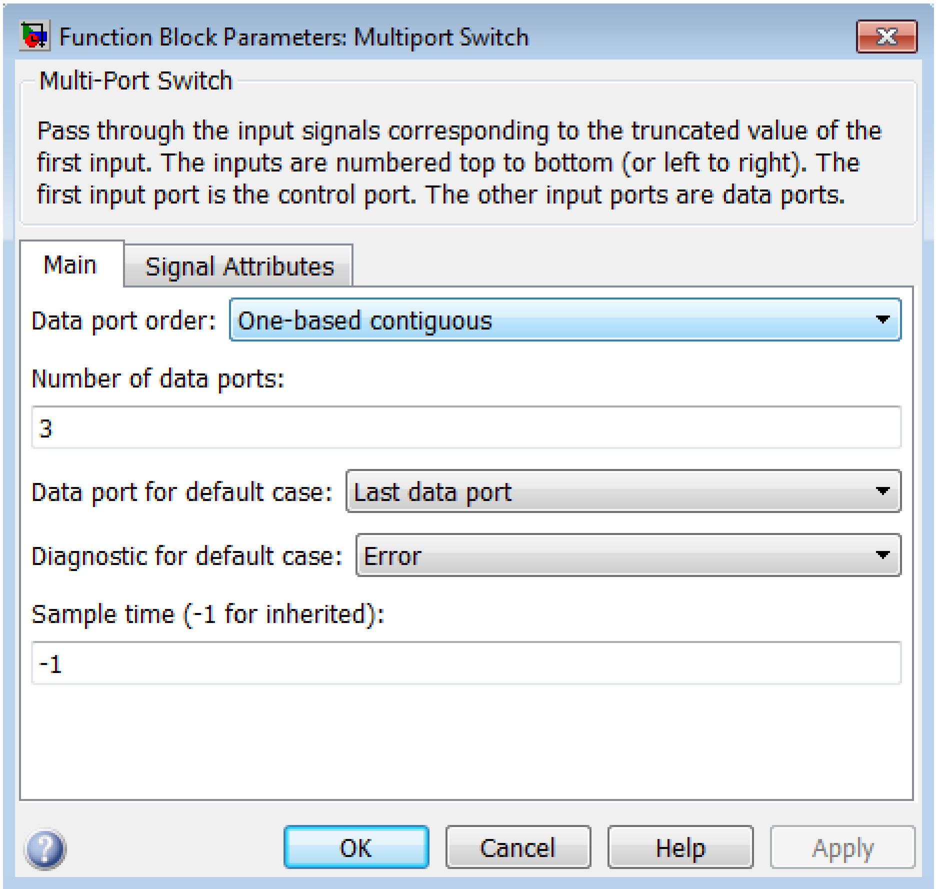
The control signal can be of any data type that Simulink supports, including fixed-point and enumerated types. If the control signal is numeric, it cannot be complex. If the control signal is an enumerated signal, the block uses the value of the underlying integer to select a data port. If the underlying integer does not correspond to a data port, an error occurs.

The data signals can be of any data type that Simulink supports. If any data signal is of an enumerated type, all others must be of the same enumerated type.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Multiport Switch block dialog box appears as follows:



Function Block Parameters: Multiport Switch

Multi-Port Switch

Pass through the input signals corresponding to the truncated value of the first input. The inputs are numbered top to bottom (or left to right). The first input port is the control port. The other input ports are data ports.

Data port order: One-based contiguous

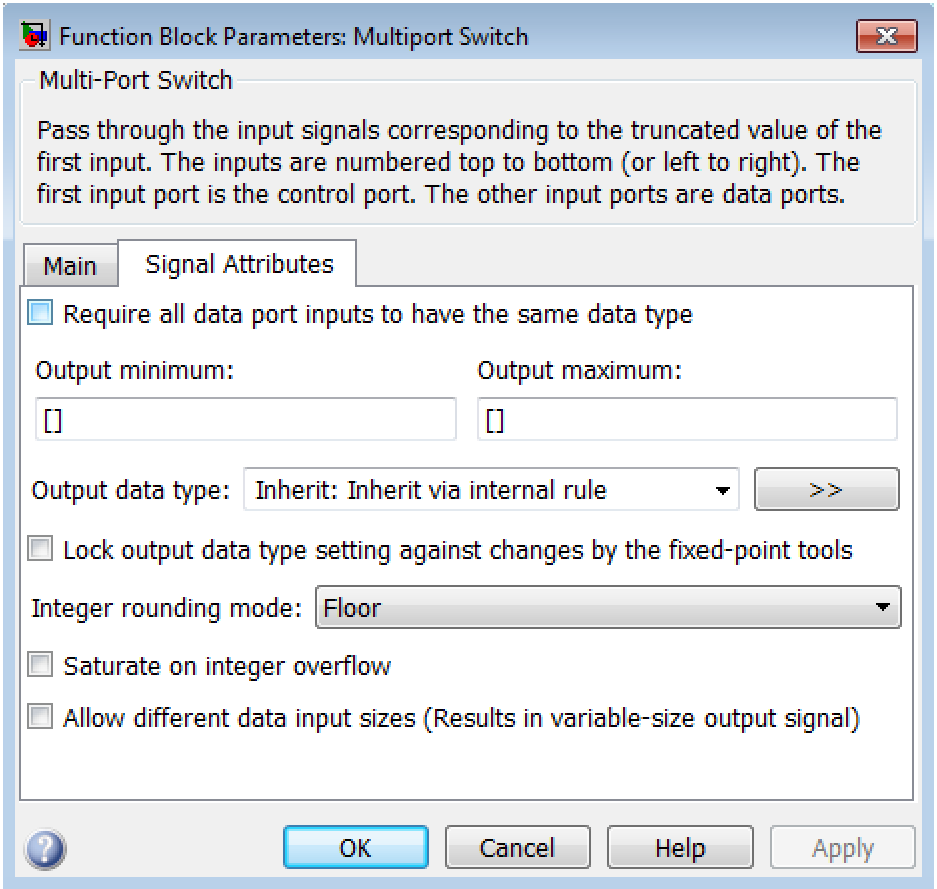
Number of data ports: 3

Data port for default case: Last data port

Diagnostic for default case: Error

Sample time (-1 for inherited): -1

The **Signal Attributes** pane of the Multiport Switch block dialog box appears as follows:



Function Block Parameters: Multiport Switch

Multi-Port Switch

Pass through the input signals corresponding to the truncated value of the first input. The inputs are numbered top to bottom (or left to right). The first input port is the control port. The other input ports are data ports.

☐ Require all data port inputs to have the same data type

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate on integer overflow

☐ Allow different data input sizes (Results in variable-size output signal)

- "Data port order" on page 1-1033
- "Number of data ports" on page 1-1035
- "Data port indices" on page 1-1036
- "Data port for default case" on page 1-1038
- "Diagnostic for default case" on page 1-1039

- “Sample time (-1 for inherited)” on page 1-1974
- “Require all data port inputs to have the same data type” on page 1-1041
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1721
- “Integer rounding mode” on page 1-1722
- “Saturate on integer overflow” on page 1-1724
- “Allow different data input sizes” on page 1-1047
- “Output minimum” on page 1-1048
- “Output maximum” on page 1-1049
- “Output data type” on page 1-1050
- “Mode” on page 1-1052
- “Data type override” on page 1-1926
- “Signedness” on page 1-1055
- “Word length” on page 1-1056
- “Scaling” on page 1-1736
- “Fraction length” on page 1-1058
- “Slope” on page 1-1059
- “Bias” on page 1-1060

Data port order

Specify the type of ordering for your data input ports.

Settings

Default: `One-based contiguous` (for Multiport Switch block), `Zero-based contiguous` (for Index Vector block)

`Zero-based contiguous`
: Block uses zero-based indexing for ordering contiguous data ports.

`One-based contiguous`
: Block uses one-based indexing for ordering contiguous data ports.

`Specify indices`
: Block uses noncontiguous indexing for ordering data ports.

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 `slupdate` on your model to replace each Multiport Switch block of this configuration with a block that explicitly specifies data port indices.
- Avoid situations where the block contains unused data ports for simulation or code generation. When the control port is of fixed-point or built-in data type, verify that all data port indices are representable with that type. Otherwise, the following block behavior occurs:

If the block has unused data ports and data port order is...	You get...
`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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Number of data ports

Specify the number of data input ports to the block.

Settings

Default: `3` (for Multiport Switch block), `1` (for Index Vector block)

The block icon changes to match the number of data input ports you specify.

Dependency

Selecting `Zero-based contiguous` or `One-based contiguous` for **Data port order** enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Data port indices

Specify an array of indices for your data ports.

Settings

Default: `{1,2,3}`

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.

Dependency

Selecting `Specify indices` for **Data port order** enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Data port for default case

Specify whether to use the last data port for out-of-range inputs, or to use an additional port.

Settings

Default: `Last data port`

`Last data port`

Block uses the last data port for output when the control port value does not match any data port indices.

`Additional data port`

Block uses an additional data port for output when the control port value does not match any data port indices.

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Diagnostic for default case

Specify the diagnostic action to take when the control port value does not match any data port indices.

Settings

Default: `Error`

`None`
: Do not show any warning or error message.

`Warning`
: Show a warning message in the MATLAB Command Window and continue the simulation.

`Error`
: Show an error message in the Simulation Diagnostics Viewer and stop the simulation. 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 1-1025.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Require all data port inputs to have the same data type

Specify allowed data types.

Settings

Default: Off

☑ On

Requires all data port inputs to have the same data type.

☐ Off

Allows data port inputs to have different data types.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
: Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`
: Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`
: Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`
: Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`
: Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`
: Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`
: Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see “Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Allow different data input sizes

Select this check box to allow input signals with different sizes.

Settings

Default: Off

☑ On

Allows input signals with different sizes, and propagate the input signal size to the output signal.

☐ Off

Requires that input signals be the same size.

Command-Line Information

Parameter: `AllowDiffInputSize`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Output minimum

Specify the minimum value that the block outputs.

Settings

Default: `[]`

The default value is `[]` (unspecified).

Simulink uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Tip

This number must be a finite real double scalar value.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Output maximum

Specify the maximum value the block outputs.

Settings

Default: `[]`

The default value is `[]` (unspecified).

Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Tip

This number must be a finite real double scalar value.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Inherit via internal rule`

`Inherit:  Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit:  Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit:  Inherit via back propagation`

Uses the data type of the driving block.

`double`

Specifies output data type `double`.

`single`

Specifies output data type `single`.

`int8`

Specifies output data type `int8`.

`uint8`
: Specifies output data type `uint8`.

`int16`
: Specifies output data type `int16`.

`uint16`
: Specifies output data type `uint16`.

`int32`
: Specifies output data type `int32`.

`uint32`
: Specifies output data type `uint32`.

`fixdt(1,16,0)`
: Specifies output data type fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
: Specifies output data type fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
: Uses a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit via internal rule` (default)
- `Inherit via back propagation`

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Specifies fixed-point data types.

`Expression`

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specifies the fixed-point data as signed.

`Unsigned`
: Specifies the fixed-point data as unsigned.

Dependency

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Dependency

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`

`Binary point`
Specify binary point location.

`Slope and bias`
Enter slope and bias.

`Best precision`
Specify best-precision values.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependency

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependency

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type".

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependency

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type”.

Bus Support

The Multiport Switch block is a bus-capable block. The data inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- All the buses must be equivalent (same hierarchy with identical names and attributes for all elements).
- All signals in a nonvirtual bus input to a Multiport Switch block must have the same sample time. This requirement holds even when the elements of the associated bus object specify inherited sample times.

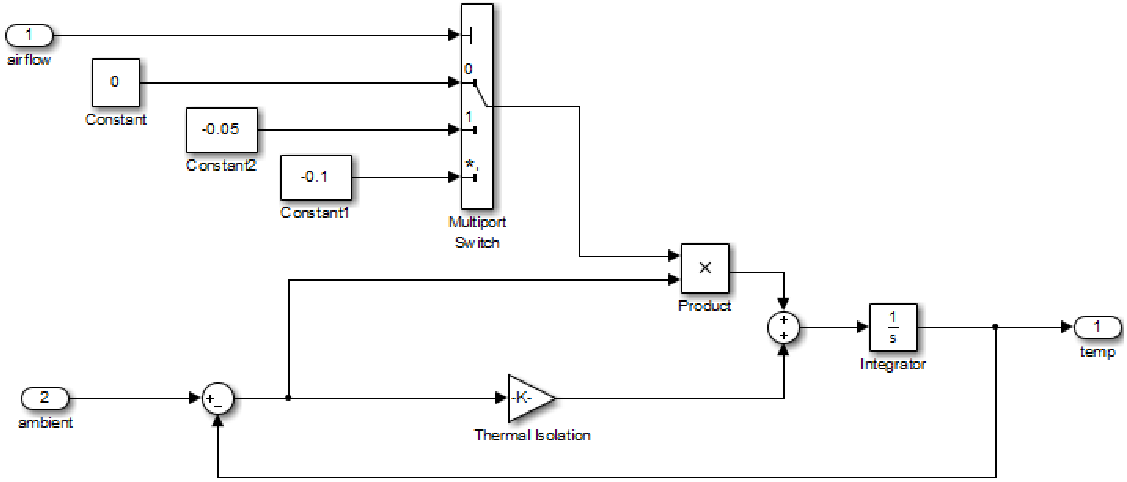
You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “About Composite Signals” and “Bus-Capable Blocks” for more information.

You can use an array of buses as an input signal to a Multiport Switch block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”. When you use an array of buses with a Multiport Switch block, set **Number of data ports** to a value of 2 or greater.

Examples

Zero-Based Indexing for Data Ports

The `sf_aircontrol` model uses a Multiport Switch block in the `Physical Plant` subsystem. This block uses zero-based indexing for contiguous ordering of three data ports.

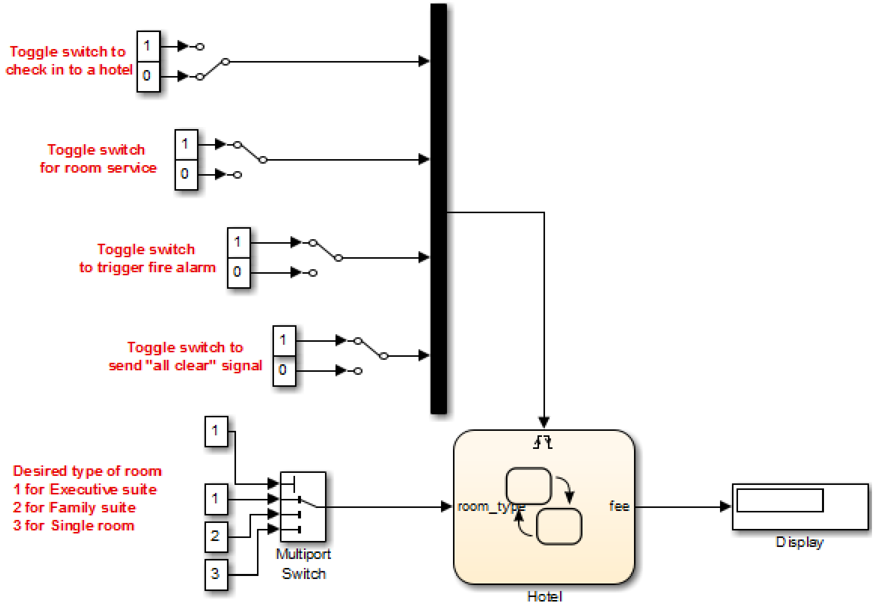

1
airflow
0
Constant
-0.05
Constant2
-0.1
Constant1
0
1
*,
Multiport
Switch
×
Product
+
+
1
s
Integrator
1
temp
2
ambient
+
_
-K-
Thermal Isolation

The indices are visible on the data port labels. You do not have to open the block dialog box to determine whether the data ports use zero-based or one-based indexing.

When you set **Data port for default case** to `Last data port`, the last data port includes a `*` on the label. The comma after the `*` indicates that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices 0, 1, or 2. In this case, the Multiport Switch block outputs a value of –0.1.

One-Based Indexing for Data Ports

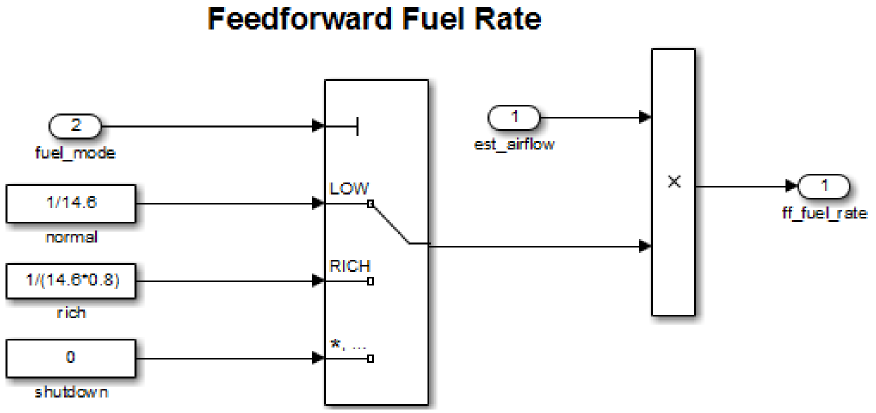
The `sf_semantics_hotel_checkin` model uses a Multiport Switch block. This block uses one-based indexing for contiguous ordering of three data ports.


Toggle switch to check in to a hotel
1
0
Toggle switch for room service
1
0
Toggle switch to trigger fire alarm
1
0
Toggle switch to send "all clear" signal
1
0
Desired type of room
1 for Executive suite
2 for Family suite
3 for Single room
1
1
2
3
Multiport Switch
room_type
fee
Hotel
Display

If you increase the size of the block icon, the indices are visible on the data port labels. You do not have to open the block dialog box to determine whether the data ports use zero-based or one-based indexing.

Enumerated Names for Data Port Indices

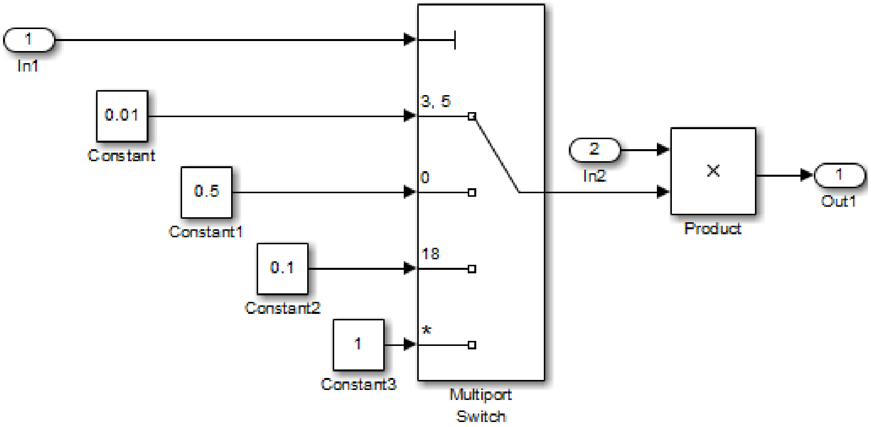
The `sldemo_fuelsys` model uses a Multiport Switch block in the `fuel_rate_control/fuel_calc/feedforward_fuel_rate` subsystem. This block uses the enumerated type `sld_FuelModes` to specify three data port indices: LOW, RICH, and DISABLED.


Feedforward Fuel Rate
2
fuel_mode
1/14.6
normal
1/(14.6*0.8)
rich
0
shutdown
LOW
RICH
*, ...
1
est_airflow
×
1
ff_fuel_rate

When you set **Data port for default case** to `Last data port`, the last data port includes a `*` on the label. The comma and ellipsis after the `*` indicate that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices LOW, RICH, or DISABLED. In this case, the Multiport Switch block outputs a value of 0.

Noncontiguous Values for Data Port Indices

The following model uses a Multiport Switch block that specifies noncontiguous integer values for data ports.

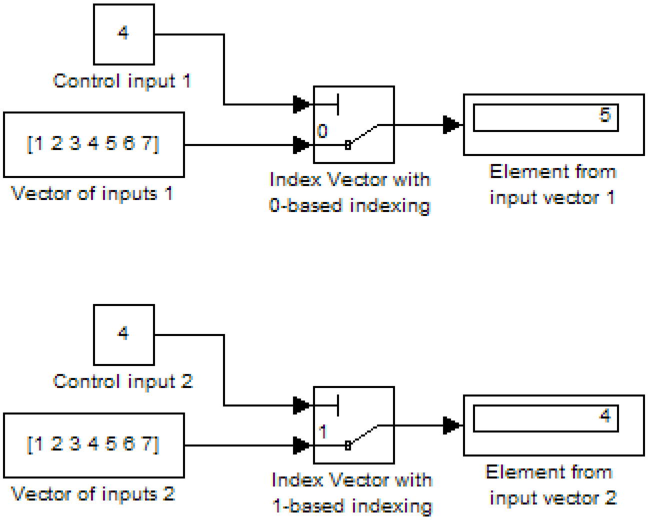

1
In1
0.01
Constant
0.5
Constant1
0.1
Constant2
1
Constant3
3, 5
0
18
*
Multiport Switch
2
In2
×
Product
1
Out1

The values of the indices are visible on the data port labels. You do not have to open the block dialog box to determine which value maps to each data port.

When you set **Data port for default case** to `Additional data port`, an extra port with a `*` label appears. This port corresponds to the default case, which applies when the control input does not match the data port indices 3, 5, 0, or 18. In this case, the Multiport Switch block outputs a value of 1.

Behavior for an Index Vector Block

The following model uses two Index Vector blocks that specify zero-based and one-based indexing, respectively.


4
Control input 1
[1 2 3 4 5 6 7]
Vector of inputs 1
0
Index Vector with
0-based indexing
5
Element from
input vector 1
4
Control input 2
[1 2 3 4 5 6 7]
Vector of inputs 2
1
Index Vector with
1-based indexing
4
Element from
input vector 2

If the value of the control input is not an integer, the block truncates the value by rounding to floor. For more information, see “How the Block Interprets the Control Input” on page 1-1024.

Characteristics

Bus-capable	Yes, with restrictions
Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes

Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also Switch

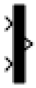
Purpose

Combine several input signals into vector

Library

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. The elements of the vector output signal take their order from the top to bottom, or left to right, input port signals. See "How to Rotate a Block" for a description of the port order for various block orientations. To avoid adding clutter to a model, Simulink hides the name of a Mux block when you copy it from the Simulink library to a model. See "Mux Signals" for information about creating and decomposing vectors.

Note The Mux block allows you to connect signals of differing data and numeric types and matrix signals to its inputs. In this case, the Mux block acts like a Bus Creator block and outputs a bus signal rather than a vector. MathWorks discourages using Mux blocks to create bus signals, and might not support this practice in future releases. See "Avoid Mux/Bus Mixtures" for more information.

Use the **Number of inputs** parameter to specify input signal names and sizes as well as the number of inputs. You can use one of the following 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 will have two inputs, named `position` and `velocity`.

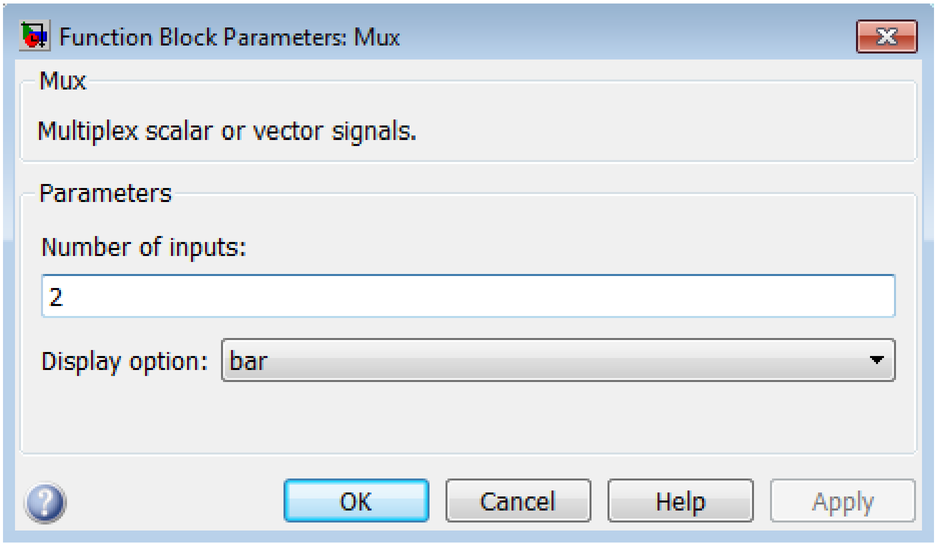
MathWorks encourages using Vector Concatenate blocks rather than Mux blocks to combine vectors. The primary exception is the creation of a vector of function calls, which requires a Mux block. In future releases, Mux blocks might have no unique capabilities and might be deprecated.

To create a composite signal, in which the constituent signals retain their identities and can have different data types, use a Bus Creator block rather than a Mux block. Although you can use a Mux block to create a composite signal, MathWorks discourages this practice. See “Avoid Mux/Bus Mixtures” for more information.

Data Type Support

The Mux block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Mux

Mux

Multiplex scalar or vector signals.

Parameters

Number of inputs: 2

Display option: bar

OK
Cancel
Help
Apply

Parameters and Dialog Box

Number of inputs

Specify number and size of inputs.

Settings

Default: `2`

You can enter a comma-separated list of signal names for this parameter field.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Display option

Specify the appearance of the block in the model.

Settings

Default: `bar`

`bar`
: Displays the block in a solid foreground color

`none`
: `Mux` appears inside the block

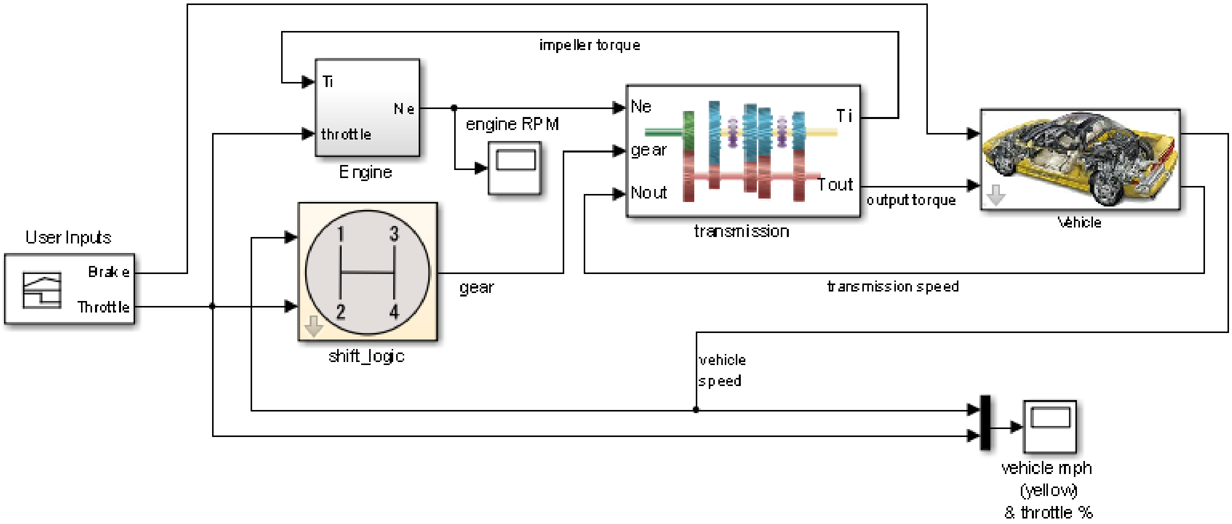
`signals`
: Displays signal names next to each port

Command-Line Information

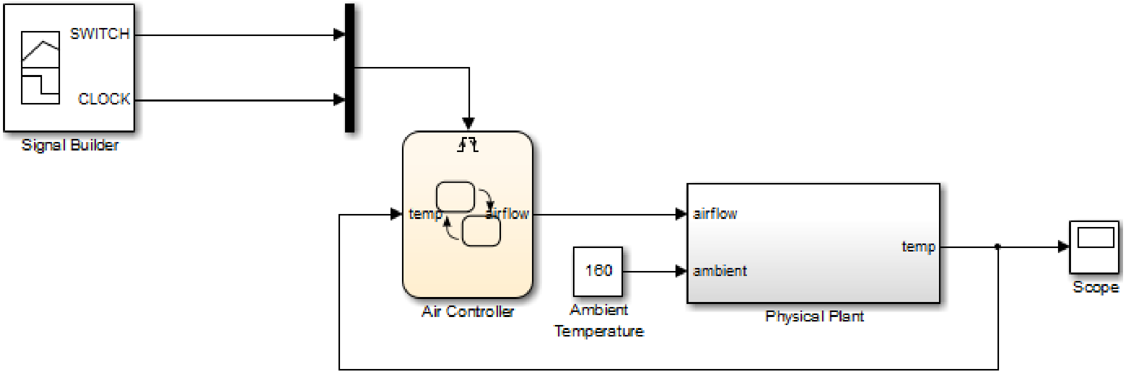
See “Block-Specific Parameters” on page 6-109 for the command-line information.

Examples

The `sf_car` model uses a Mux block to combine two signals for input to a Scope block:


impeller torque
Ti
Ne
throttle
Engine
engine RPM
Ne
gear
Nout
Ti
Tout
transmission
output torque
Vehicle
User Inputs
Brake
Throttle
1
3
2
4
shift_logic
gear
transmission speed
vehicle
speed
vehicle mph
(yellow)
& throttle %

The `sf_aircontrol` model uses a Mux block to combine two signals for input to a Stateflow chart:


SWITCH
CLOCK
Signal Builder
temp
airflow
Air Controller
160
Ambient
Temperature
airflow
ambient
temp
Physical Plant
Scope

The following models also show how to use the Mux block:

- `sldemo_auto_climatecontrol`
- `sldemo_suspn`
- `sldemo_zeroxing`
- `penddemo`

Characteristics

Virtual	Yes For more information, see “Virtual Blocks” in the Simulink documentation.

See Also

Demux

Outport

Purpose

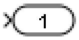
Create output port for subsystem or external output

Library

Ports & Subsystems, Sinks

Description

Outport blocks are the links from a system to a destination outside the system.



Simulink software assigns Outport block port numbers according to these rules:

- It automatically numbers the Outport blocks within a top-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 on a Subsystem block is the block whose **Port number** parameter matches the relative position of the output port on the Subsystem block. For example, the 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, although the block continues to send the signal to the same block outside the subsystem.

When you create a subsystem by selecting existing blocks, if more than one Outport block is included in the grouped blocks, Simulink software automatically renumbers the ports on the blocks.

The Outport block name appears in the Subsystem icon as a port label. To suppress display of the label, click the Outport block and select **Format > Hide Name**.

Outport Blocks in Conditionally Executed Contexts

When an Outport block is in a conditionally executed subsystem or a Model block, use one of the following approaches:

- Specify an initial output value for the Outport block.
- Use the value inherited from input signal of the Outport block.

Specifying Initial Conditions

To explicitly specify an initial output value:

1. Select `Dialog` in the **Source of initial output value** drop-down list.
2. Specify the **Initial output** parameter.

If you select `Dialog`, you can also specify what happens to the output when the subsystem is disabled, using the **Output when disabled** drop-down menu. Select either:

- `reset` – The output value is reset to the **Initial output** value when the subsystem is disabled.
- `held` – The output value remains at its most recent value when the subsystem is disabled.

Note If you are connecting the output of the conditionally executed subsystem to a Merge block, set **Output when disabled** to `held` to ensure consistent simulation results.

If you are using simplified initialization mode, you must select `held` when connecting a conditionally executed subsystem to a Merge block. However, you do not need to specify an Initial Condition (IC) for the Outport block of a conditionally executed subsystem that directly drives a Merge block. (This rule applies only when Outport and Merge blocks are in the same model.) For more information, see “Underspecified initialization detection”.

Inheriting Initial Conditions

The initial output value of the outport block can be inherited from the following sources:

- Output port of another conditionally executed subsystem
- Merge block (with Initial output specified)
- Function-Call Model Reference block
- Constant block (simplified initialization mode only)
- IC block (simplified initialization mode only)

The procedure you use to inherit the initial conditions of the Outport block differs depending on whether you are using classic initialization mode or simplified initialization mode.

To inherit initial conditions in classic initialization mode:

1. Select `Dialog` in the **Source of initial output value** drop-down list.
2. Set the **Initial output** parameter to `[]` (empty matrix).
3. Click **OK**.

Note For all other driving blocks, specify an explicit initial output value.

To inherit initial conditions in simplified initialization mode:

1. Select `Input signal` in the **Source of initial output value** drop-down list.
2. Click **OK**.

   The **Initial output** and **Output when disabled** parameters are disabled, and the values are both inherited from the input signal.

For more information on classic and simplified initialization mode, see "Underspecified initialization detection".

Outport Blocks in a Top-Level System

Outport blocks in a top-level system have two uses: to supply external outputs to the workspace, which you can do by using either the **Model Configuration Parameters** dialog box or the `sim` command, and to provide a means for analysis functions to obtain output from the system.

- To supply external outputs to the workspace, use the Configuration Parameters dialog box (see Exporting Output Data to the MATLAB Workspace) or the `sim` command (see `sim`). 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`, you could specify this command:

  ```
  [t,x,speed,dist] = sim(...);
  ```

- To provide a means for the `linmod` and `trim` analysis functions to obtain output from the system (see "Linearizing Models")

Connecting Buses to Root Level 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, you must set the Outport block **Data type** parameter to use a bus object name for the `Bus: <object name>` or `<data type expression>` option, to define the type of bus that the Outport produces. If the bus signal is virtual, it will be converted to nonvirtual, as described in “Automatic Bus Conversion”. See “Bus Objects” more information.

Data Type Support

The Outport block accepts real or complex signals of any data type that Simulink supports. An Outport block can also accept fixed-point and enumerated data types when the block is not a root-level output port. The complexity and data type of the block output are the same as those of its input. The Outport block also accepts a bus object as a data type.

Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

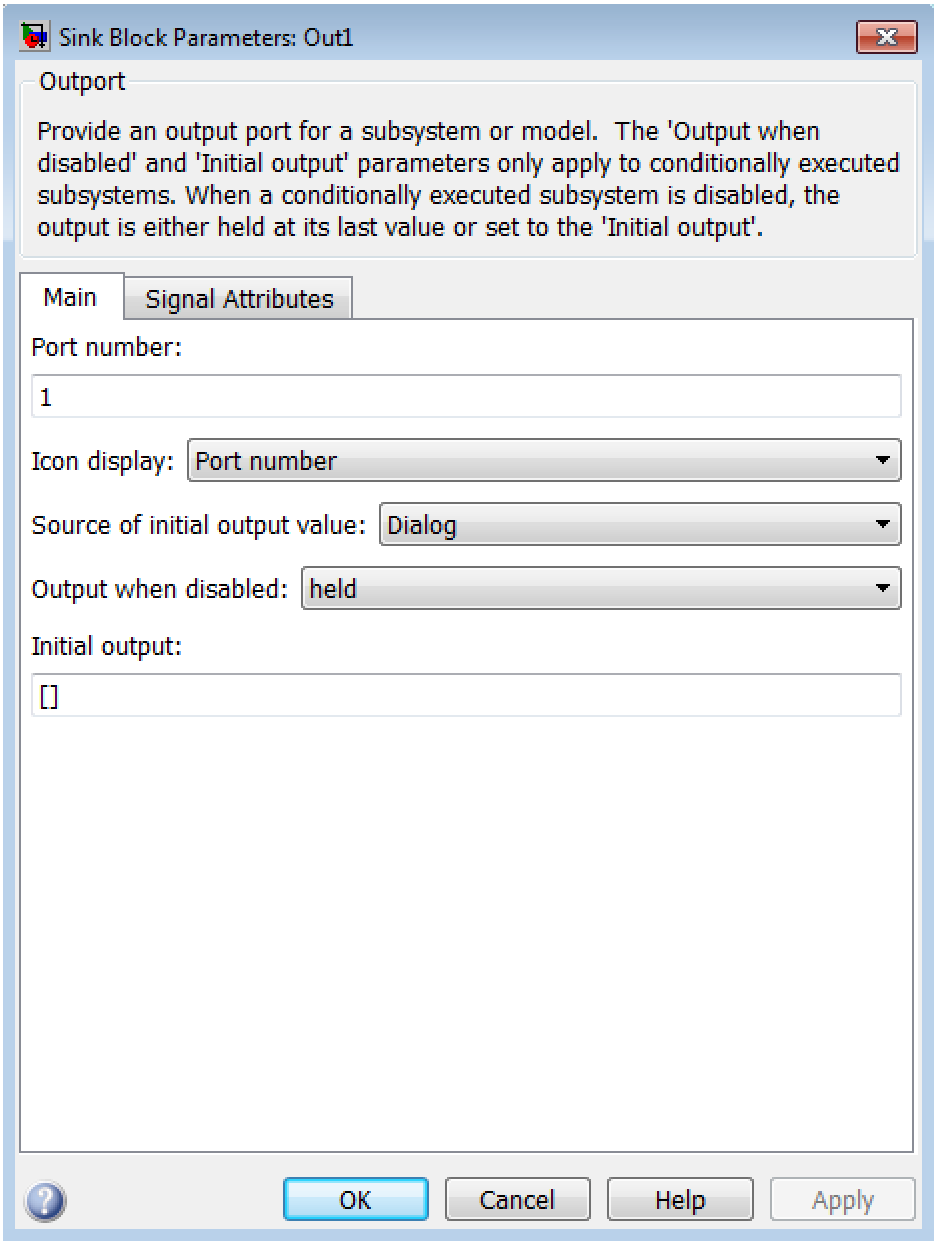
For more information, see “Data Types Supported by Simulink”.

The elements of a signal array connected to an Outport block can be of differing complexity and data types except in the following circumstance: If the output port is in a conditionally executed subsystem and the initial output is specified, all elements of an input array must be of the same complexity and data types.

Typical Simulink data type conversion rules apply to an output port’s **Initial output** parameter. If the initial output value is in the range of the block’s output data type, Simulink software converts the initial output to the output data type. If the specified initial output is out of the range of the output data type, Simulink software halts the simulation and signals an error.

Parameters and Dialog Box

The **Main** pane of the Outport block dialog box appears as follows:



Sink Block Parameters: Out1

Outport

Provide an output port for a subsystem or model. The 'Output when disabled' and 'Initial output' parameters only apply to conditionally executed subsystems. When a conditionally executed subsystem is disabled, the output is either held at its last value or set to the 'Initial output'.

Port number: 1

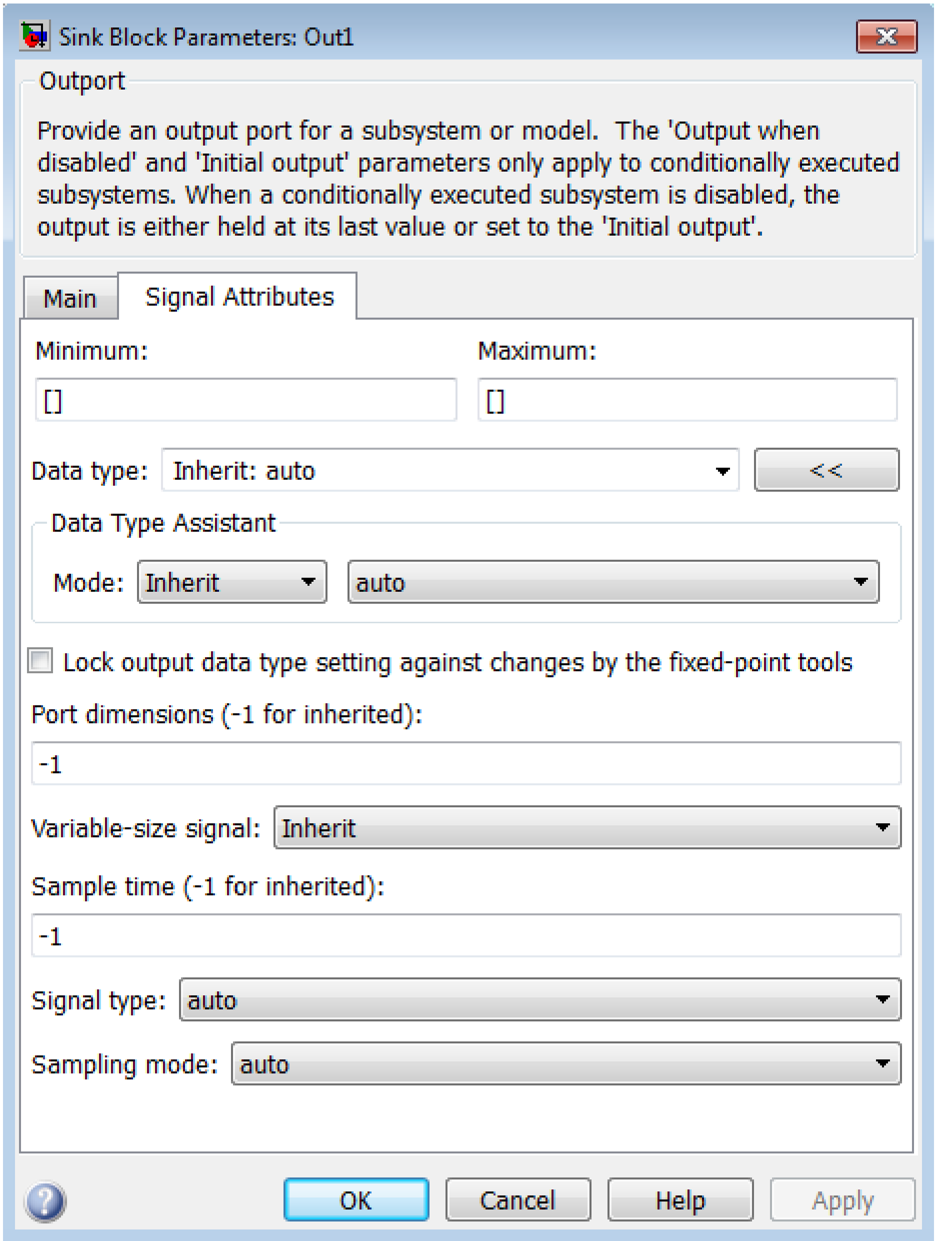
Icon display: Port number

Source of initial output value: Dialog

Output when disabled: held

Initial output: []

The **Signal Attributes** pane of the Outport block dialog box appears as follows:



Sink Block Parameters: Out1

Outport

Provide an output port for a subsystem or model. The 'Output when disabled' and 'Initial output' parameters only apply to conditionally executed subsystems. When a conditionally executed subsystem is disabled, the output is either held at its last value or set to the 'Initial output'.

Minimum: []

Maximum: []

Data type: Inherit: auto

Data Type Assistant

Mode: Inherit auto

☐ Lock output data type setting against changes by the fixed-point tools

Port dimensions (-1 for inherited): -1

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Sampling mode: auto

- “Package” on page 1-1977
- “Icon display” on page 1-1086
- “Source of initial output value” on page 1-1087
- “Output when disabled” on page 1-1088
- “Initial output” on page 1-1089
- “Minimum” on page 1-1920
- “Maximum” on page 1-1921
- “Data type” on page 1-1091
- “Show data type assistant” on page 1-1672
- “Mode” on page 1-1094
- “Data type override” on page 1-1926
- “Signedness” on page 1-1927
- “Word length” on page 1-1930
- “Scaling” on page 1-1736
- “Fraction length” on page 1-1931
- “Slope” on page 1-1932
- “Bias” on page 1-1932
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1721
- “Output as nonvirtual bus in parent model” on page 1-1102
- “Port dimensions (-1 for inherited)” on page 1-1104
- “Variable-size signal” on page 1-1105
- “Sample time (-1 for inherited)” on page 1-1974
- “Signal type” on page 1-1108
- “Sampling mode” on page 1-1109

Port number

Specify the port number of the block.

Settings

Default: 1

This parameter controls the order in which the port that corresponds to the block appears on the parent subsystem or model block.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Icon display

Specify the information to be displayed on the icon of this input port.

Settings

Default: `Port number`

`Signal name`
: Display the name of the signal connected to this port (or signals if the input is a bus).

`Port number`
: Display port number of this port.

`Port number and signal name`
: Display both the port number and the names of the signals connected to this port.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Source of initial output value

Select the source of the initial output value of the Outport block.

Settings

Default: `Dialog`

`Dialog`
The initial output value is specified by the **Initial output** parameter on the dialog.

`Input signal`
The initial output value is inherited from the input signal.

Tips

- If you are using classic initialization mode, you *must* select `Dialog`. Selecting `Input signal` will cause an error.
- If you are using classic initialization mode and want to inherit the initial output value, set this parameter to `Dialog`, and then specify `[]` (empty matrix) for the **Initial output** value. For more information, see Inheriting Initial Conditions on page 1077.

Dependencies

This parameter applies only if the Outport resides in an Enabled Subsystem.

Selecting `Dialog` enables the following parameters:

- **Output when disabled**
- **Initial output**

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output when disabled

Specify what happens to the block output when the subsystem is disabled.

Settings

Default: `held`

`held`
: Output is held when the subsystem is disabled.

`reset`
: Output is reset when the subsystem is disabled.

Tips

- If you are connecting the output of a conditionally executed subsystem to a Merge block, set **Output when disabled** to `held` to ensure consistent simulation results.
- If you are using simplified initialization mode, you must select `held` when connecting a conditionally executed subsystem to a Merge block.

Dependencies

- Selecting `Dialog` in **Source of initial output value** enables this parameter.
- This parameter applies only if the Outport resides in an Enabled Subsystem.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Initial output

For conditionally executed subsystems, specify the block output before the subsystem executes and while it is disabled.

Settings

Default: `[ ]`

Simulink software does not allow the initial output of this block to be `inf` or `NaN`.

Tips

- If you are using classic initialization mode, specify `[]` (empty matrix) to inherit the initial output value from the input signal. For more information, see Inheriting Initial Conditions on page 1077.
- You can also specify `[]` if you are using classic initialization mode, and your model does not depend on the initial output of the conditionally executed subsystem.
- If you are using simplified initialization mode, you cannot specify `[]`, you must specify an explicit value. If you do not want to specify an initial output value for this block, set **Source of initial output value** to `Input signal`.
- For information about specifying an initial condition structure, see "Specify Initial Conditions for Bus Signals"

Dependencies

- Selecting `Dialog` in **Source of initial output value** enables this parameter.
- This parameter applies only if the Outport resides in an Enabled Subsystem.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Minimum

Specify the minimum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Maximum

Specify the maximum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Data type

Specify the output data type of the external input.

Settings

Default: `Inherit:  auto`

`Inherit:  auto`
A rule that inherits a data type

`double`
Data type is `double`.

`single`
Data type is `single`.

`int8`
Data type is `int8`.

`uint8`
Data type is `uint8`.

`int16`
Data type is `int16`.

`uint16`
Data type is `uint16`.

`int32`
Data type is `int32`.

`uint32`
Data type is `uint32`.

`boolean`
Data type is `boolean`.

`fixdt(1,16,0)`
Data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
Data type is fixed point `fixdt(1,16,2^0,0)`.

`Enum: <class name>`
Data type is enumerated, for example, `Enum: BasicColors`.

`Bus: <object name>`
Data type is a bus object.

`<data type expression>`
The name of a data type object, for example `Simulink.NumericType`

Do not specify a bus object as the expression.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`
: Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right.

`Built in`
: Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

  - `double` (default)
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`
  - `uint32`
  - `boolean`

`Fixed point`
: Fixed-point data types.

`Enumerated`
: Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

`Bus`
: Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or

change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details about the Bus Editor, see “Manage Bus Objects with the Bus Editor”.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specify the fixed-point data as signed.

`Unsigned`
: Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`

`Binary point`
Specify binary point location.

`Slope and bias`
Enter slope and bias.

`Best precision`
Specify best-precision values.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On
  Locks the output data type setting for this block.

☐ Off
  Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Use Lock Output Data Type Setting".

Output as nonvirtual bus in parent model

Select this parameter if you want the bus emerging in the parent model to be nonvirtual. The bus that is input to the port can be virtual or nonvirtual, regardless of the setting of **Output as nonvirtual bus in parent model**.

Settings

Default: Off

☑ On
  Select this parameter if you want the bus emerging in the parent model to be nonvirtual.

☐ Off
  Clear this parameter if you want the bus emerging in the parent model to be virtual.

Tips

All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. Therefore, if you select this option all signals in the bus must have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus, to allow the signal or bus to be included in a nonvirtual bus.

Dependency

via bus object enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Port dimensions (-1 for inherited)

Specify the dimensions that a signal must have in order to be connected to this Outport block.

Settings

Default: `-1`

Valid values are:

`-1`	A signal of any dimensions can be connected to this port.
`N`	The signal connected to this port must be a vector of size `N`.
`[R C]`	The signal connected to this port must be a matrix having `R` rows and `C` columns.

Dependency

Clearing **via bus object** enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Variable-size signal

Specify the type of signals allowed out of this port.

Settings

Default: `Inherit`

`Inherit`
  Allow variable-size and fixed-size signals.

`No`
  Do not allow variable-size signals.

`Yes`
  Allow only variable-size signals.

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: string
Value: `'Inherit’| 'No' | 'Yes'`
Default: `'Inherit'`

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Signal type

Specify the numeric type of the signal output by this block.

Settings

Default: `auto`

`auto`
: Output the numeric type of the signal that is connected to its input.

`real`
: Output a real-valued signal. The signal connected to this block must be real. If it is not, Simulink software displays an error if you try to update the diagram or simulate the model that contains this block.

`complex`
: Output a complex signal. The signal connected to this block must be complex. If it is not, Simulink software displays an error if you try to update the diagram or simulate the model that contains this block.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sampling mode

Specify the sampling mode (`Sample based` or `Frame based`) that the input signal must match.

Settings

Default: `auto`

`auto`
  Accept any sampling mode.

`Sample based`
  The output signal is sample-based.

`Frame based`
  The output signal is frame-based.

Dependency

Frame-based processing requires a DSP System Toolbox license.

For more information, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Sample Time	Inherited from the driving block
Dimensionalized	Yes
Multidimensionalized	Yes
Virtual	Yes, when the block resides in a subsystem block and not at the root level of a model For more information, see "Virtual Blocks" in the Simulink documentation.
Zero-Crossing Detection	No

See Also

Inport

Purpose

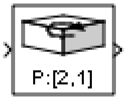
Rearrange dimensions of multidimensional array dimensions

Library

Math Operations

Description

The block reorders the elements of the input signal so that they are in the order you specify in the **Order** parameter.

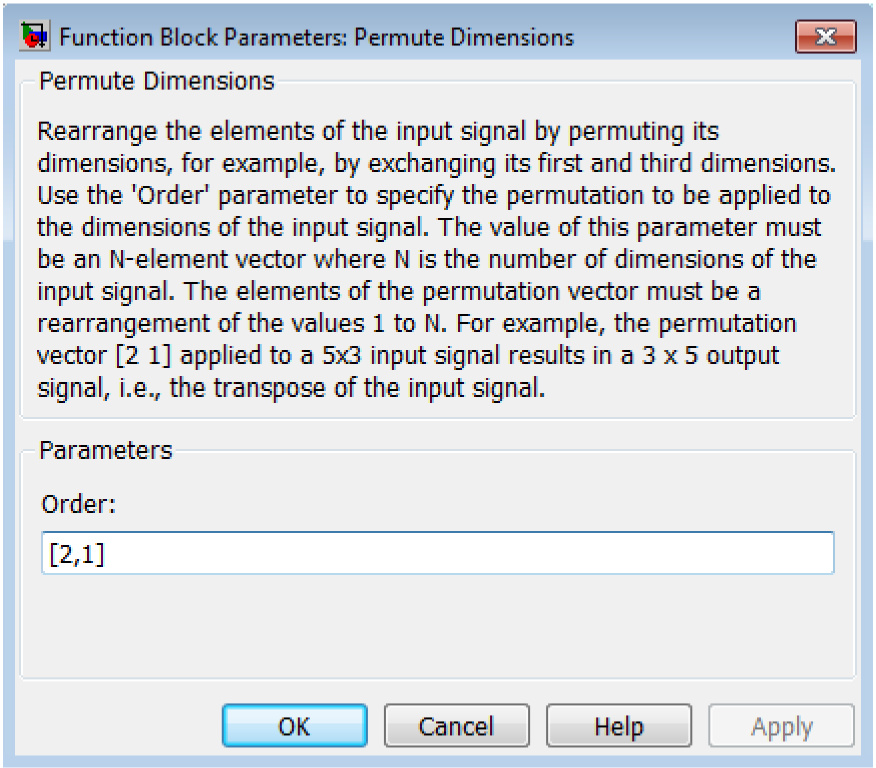


Data Type Support

This block accepts signals of any data type that Simulink supports, including fixed-point and enumerated data types. Output must be the same data type as the input.

You can use an array of buses as an input signal to a Permute Dimensions block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Rearrange the elements of the input signal by permuting its dimensions, for example, by exchanging its first and third dimensions. Use the 'Order' parameter to specify the permutation to be applied 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 1 to N. For example, the permutation vector [2 1] applied to a 5x3 input signal results in a 3 x 5 output signal, i.e., the transpose of the input signal.

Order: [2,1]

Parameters and Dialog Box

Order

Specify the permutation order to apply to the dimensions of the input signal. This parameter is a vector of elements, where the number of elements in the vector is the number of dimensions of the input signal.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No

Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Math Function (`transpose`), `permute` (in the MATLAB reference documentation)

PID Controller

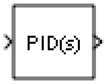
Purpose

Simulate continuous- or discrete-time PID controllers

Library

Continuous, Discrete

Description



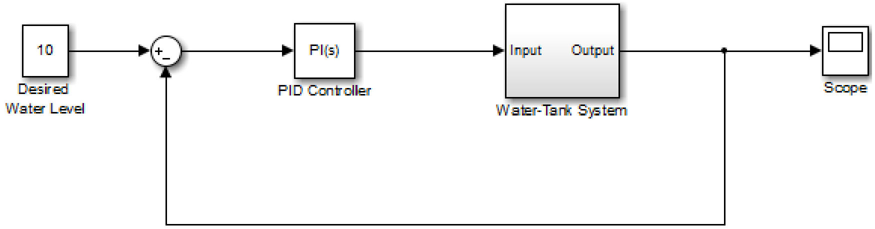
Implement a continuous- or discrete-time controller (PID, PI, PD, P, or I) in your Simulink model. PID controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design™ software (PID Tuner or SISO Design Tool).

The PID Controller block output is a weighted sum of the input signal, the integral of the input signal, and the derivative of the input signal. The weights are the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action.

Configurable options in the PID Controller block include:

- Controller type (PID, PI, PD, P, or I)
- Controller form (Parallel or Ideal)
- Time domain (continuous or discrete)
- Initial conditions and reset trigger
- Output saturation limits and built-in anti-windup mechanism
- Signal tracking for bumpless control transfer and multiloop control

In one common implementation, the PID Controller block operates in the feedforward path of the feedback loop:



The input of the block is typically an error signal, which is the difference between a reference signal and the system output. For a two-input block that permits setpoint weighting, see the PID Controller (2 DOF) block reference page.

You can generate code to implement your controller using any Simulink data type, including fixed-point data types. (Code generation requires Simulink Coder software; fixed-point implementation requires the Fixed-Point Designer product.)

For examples illustrating some applications of the PID Controller block, see the following Simulink examples:

- Anti-Windup Control Using a PID Controller
- Bumpless Control Transfer Between Manual and PID Control

Data Type Support

The PID Controller block accepts real signals of any numeric data type that Simulink software supports, including fixed-point data types. See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Parameters

The following table summarizes the PID Controller block parameters, accessible on the block parameter dialog box.

Task	Parameters
Choose controller form and type.	• **Controller Form** in Main tab • **Controller**
Choose discrete or continuous time.	• **Time-domain** • **Sample time**
Choose an integration method (discrete time).	• **Integrator method** • **Filter method**

Task	Parameters
Set and tune controller gains.	• **Proportional (P)** in Main tab • **Integral (I)** in Main tab • **Derivative (D)** in Main tab • **Filter coefficient (N)** in Main tab
Set integrator and filter initial conditions.	• **Initial conditions Source** in Main tab • **Integrator Initial condition** in Main tab • **Filter Initial condition** in Main tab • **External reset** in Main tab • **Ignore reset when linearizing** in Main tab
Limit block output.	• **Limit output** in PID Advanced tab • **Lower saturation limit** in PID Advanced tab • **Upper saturation limit** in PID Advanced tab • **Ignore saturation when linearizing** in PID Advanced tab
Configure anti-windup mechanism (when you limit block output).	• **Anti-windup method** in PID Advanced tab • **Back-calculation gain (Kb)** in PID Advanced tab
Enable signal tracking.	• **Enable tracking mode** in PID Advanced tab • **Tracking gain (Kt)** in PID Advanced tab

Task	Parameters
Configure data types.	• **Parameter data type** in Data Type Attributes tab • **Product output data type** in Data Type Attributes tab • **Summation output data type** in Data Type Attributes tab • **Accumulator data type** in Data Type Attributes tab • **Integrator output data type** in Data Type Attributes tab • **Filter output data type** in Data Type Attributes tab • **Saturation output data type** in Data Type Attributes tab • **Lock output data type setting against changes by the fixed-point tools** in Data Type Attributes tab • **Saturate on integer overflow** in Data Type Attributes tab • **Integer rounding mode** in Data Type Attributes tab
Configure block for code generation.	• **State name** in State Attributes tab • **State name must resolve to Simulink signal object** in State Attributes tab • **Code generation storage class** in State Attributes tab • **Code generation storage type qualifier** in State Attributes tab

Controller form

Select the controller form.

Settings

`Parallel` (Default)

Selects a controller form in which the output is the sum of the proportional, integral, and derivative actions, weighted according to the independent gain parameters **P**, **I**, and **D**. The filter coefficient **N** sets the location of the pole in the derivative filter. For a continuous-time parallel PID controller, the transfer function is:

$$C_{par}(s) = \left[P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right) \right]$$

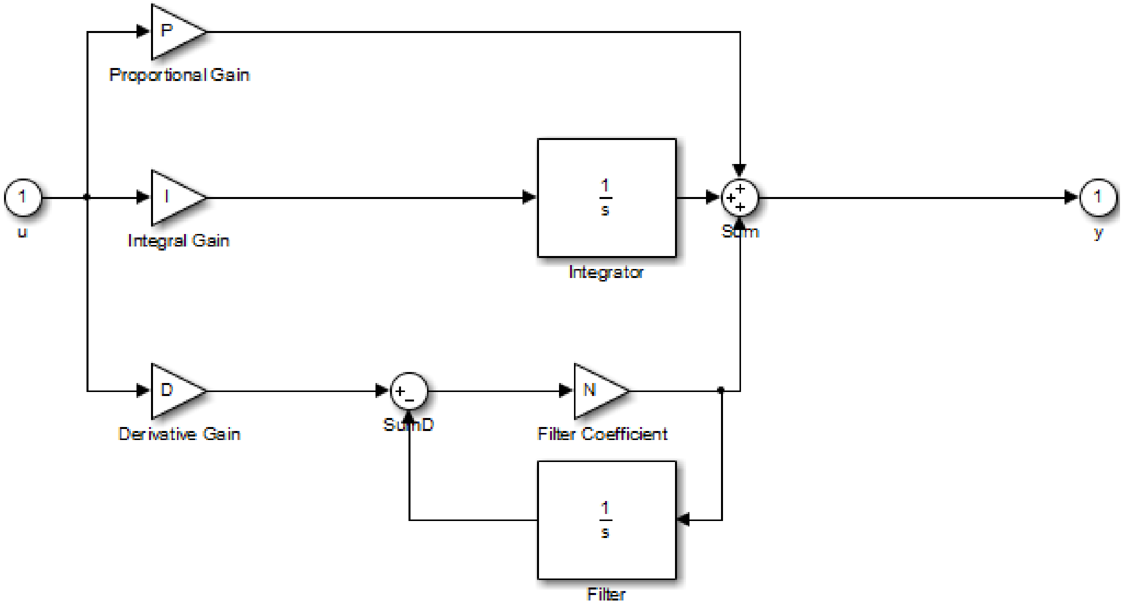
For a discrete-time parallel PID controller, the transfer function takes the form:

$$C_{par}(z) = P + Ia(z) + D\left[\frac{N}{1+Nb(z)}\right]$$

where the **Integrator method** determines `a(z)` and the **Filter method** determines `b(z)` (for sampling time $\mathtt{T}_{s}$):

	Forward Euler method	**Backward Euler method**	**Trapezoidal method**
$a(z)$ (determined by Integrator method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2}\frac{z+1}{z-1}$
$b(z)$ (determined by Filter method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2}\frac{z+1}{z-1}$

The controller transfer function for the current settings is displayed in the block dialog box.


P
Proportional Gain
1
u
I
Integral Gain
1
s
Integrator
+
+
+
Sum
1
y
D
Derivative Gain
+
–
SumD
N
Filter Coefficient
1
s
Filter

Parallel PID Controller

`Ideal`

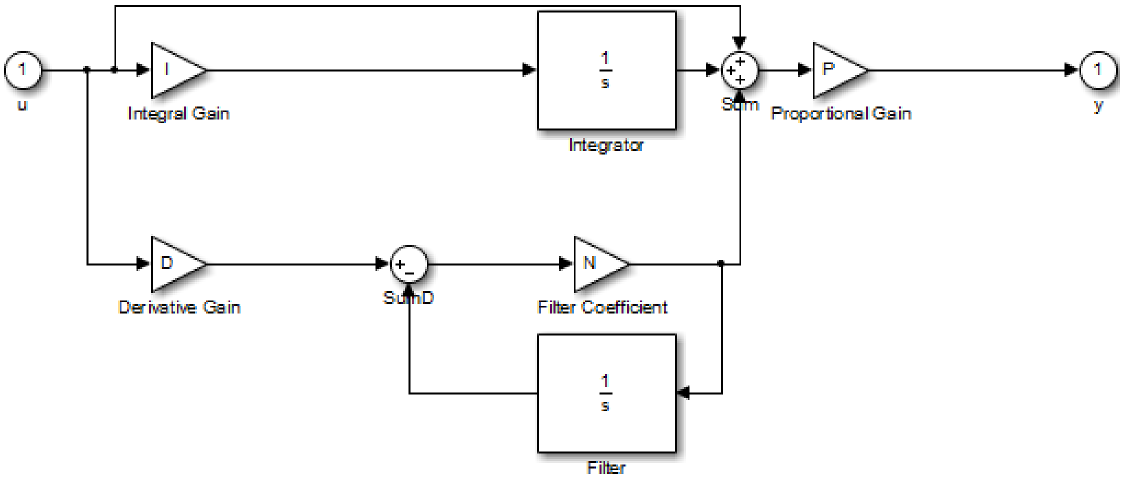
Selects a controller form in which the proportional gain `P` acts on the sum of all actions. The transfer functions are the same as for the parallel form, except that `P` multiplies all terms. For a continuous-time ideal PID controller, the transfer function is:

$$C_{id}(s) = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right)\right]$$

For a discrete-time ideal PID controller the transfer function is:

$$C_{id}(z) = P\left[1 + Ia(z) + D\frac{N}{1 + Nb(z)}\right]$$

where the **Integrator method** determines `a(z)` and the **Filter method** determines `b(z)` as described previously for the parallel controller form.


1
u
I
Integral Gain
1/s
Integrator
Sum
P
Proportional Gain
1
y
D
Derivative Gain
SumD
N
Filter Coefficient
1/s
Filter

Ideal PID Controller

Controller

Specify the controller type.

Settings

`PID` (Default)

Implements a controller with proportional, integral, and derivative action.

`PI`

Implements a controller with proportional and integral action.

`PD`

Implements a controller with proportional and derivative action.

`P`

Implements a controller with proportional action.

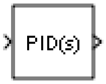
`I`

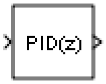
Implements a controller with integral action.

The controller transfer function for the current settings is displayed in the block dialog box.

Time-domain

Select continuous or discrete time domain. The appearance of the block changes to reflect your selection.





Settings

`Continuous-time` (Default)
: Selects the continuous-time representation.

`Discrete-time`
: Selects the discrete-time representation. Selecting `Discrete-time` also allows you to specify the:

  - **Sample time**, which is the discrete interval between samples.
  - Discrete integration methods for the integrator and the derivative filter using the **Integrator method** and **Filter method** menus.

Integrator method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the integrator output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Settings

`Forward Euler` (Default)

Selects the Forward Rectangular (left-hand) approximation.

- This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the `Forward Euler` method can result in instability, even when discretizing a system that is stable in continuous time.

`Backward Euler`

Selects the Backward Rectangular (right-hand) approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product and you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.
- An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

`Trapezoidal`

Selects the Bilinear approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product and you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see "Algebraic Loops" in the Simulink documentation.

- 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.

Filter method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the derivative filter output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Settings

`Forward Euler` (Default)

Selects the Forward Rectangular (left-hand) approximation.

- This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the `Forward Euler` method can result in instability, even when discretizing a system that is stable in continuous time.

`Backward Euler`

Selects the Backward Rectangular (right-hand) approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product, this filter method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.
- An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method.

`Trapezoidal`

Selects the Bilinear approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product, this filter method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

- An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method. Of all available filter methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Sample time (-1 for inherited)

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the discrete interval between samples.

Settings

Default: 1

By default, the block uses a discrete sample time of 1. To specify a different sample time, enter another discrete value, such as 0.1.

If you specify a value of -1, the PID Controller block inherits the sample time from the upstream block. Do not enter a value of 0; to implement a continuous-time controller, select the **Time-domain** `Continuous-time`.

See "Specify Sample Time" in the online documentation for more information.

Proportional (P)

(Available for PID, PD, PI, and P controllers.) Specify the proportional gain `P`.

Default: 1

Enter a finite, real gain value into the **Proportional (P)** field. Use either scalar or vector gain values. For a `Parallel` PID **Controller form**, the proportional action is independent of the integral and derivative actions. For an `Ideal` PID **Controller form**, the proportional action acts on the integral and derivative actions. See "Controller form" on page 1-1118 for more information about the role of `P` in the controller transfer function.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See "Choosing a Compensator Design Approach".

Integral (I)

(Available for PID, PI, and I controllers.) Specify the integral gain `I`.

Default: 1

Enter a finite, real gain value into the **Integral (I)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See “Choosing a Compensator Design Approach”.

Derivative (D)

(Available for PID and PD controllers.) Specify the derivative gain `D`.

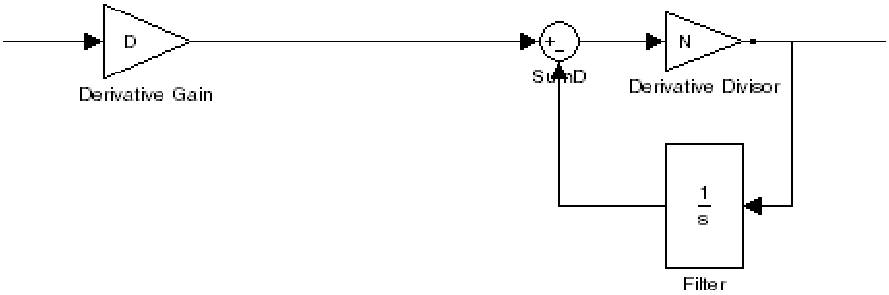
Default: 0

Enter a finite, real gain value into the **Derivative (D)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See “Choosing a Compensator Design Approach”.

Filter coefficient (N)

(Available for PID and PD controllers.) Specify the filter coefficient `N`, which determines the pole location of the filter in the derivative action:


D
Derivative Gain
+
_
SumD
N
Derivative Divisor
1
s
Filter

The filter pole falls at `s = -N` in the `Continuous-time` **Time-domain**. For `Discrete-time`, the location of the pole depends on which **Filter method** you select (for sampling time `T`$_s$):

- `Forward Euler`:

  $$z_{pole} = 1 - NT_s$$

- `Backward Euler`:

  $$z_{pole} = \frac{1}{1 + NT_s}$$

- `Trapezoidal`:

  $$z_{pole} = \frac{1 - NT_s/2}{1 + NT_s/2}$$

Default: 100.

Enter a finite, real gain value into the **Filter Coefficient (N)** field. Use either scalar or vector gain values. Note that the PID controller block does not support `N = inf` (ideal unfiltered derivative).

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See "Choosing a Compensator Design Approach". Automatic tuning requires `N > 0`.

Initial conditions Source

(Only available for controllers with integral or derivative action.) Select the source of the integrator and filter initial conditions. Simulink uses initial conditions to initialize the integrator and filter output at the start of a simulation or at a specified trigger event (See "External reset" on page 1-1137). The integrator and filter initial conditions in turn determine the initial block output.

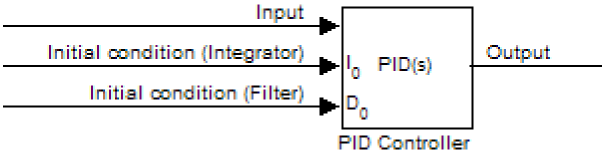
Settings

`internal` (Default)

Specifies the integrator and filter initial conditions explicitly using the **Integrator Initial condition** and **Filter Initial condition** parameters.

`external`

Specifies the integrator and filter initial conditions externally. An additional input port appears under the block input for each initial condition: $\mathtt{I}_0$ for the integrator and $\mathtt{D}_0$ for the filter:


Input
Initial condition (Integrator)
Initial condition (Filter)
I_0
PID(s)
D_0
Output
PID Controller

Integrator Initial condition

(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the integrator initial value. Simulink uses the initial condition to initialize the integrator output at the start of a simulation or at a specified trigger event (see “External reset” on page 1-1137). The integrator initial condition, together with the filter initial condition, determines the initial output of the PID controller block.

Default: 0

Simulink does not permit the integrator initial condition to be `inf` or `NaN`.

Filter Initial condition

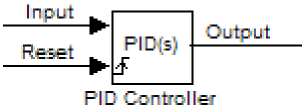
(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the filter initial value. Simulink uses the initial condition to initialize the filter output at the start of a simulation or at a specified trigger event (see “External reset” on page 1-1137). The filter initial condition, together with the integrator initial condition, determines the initial output of the PID controller block.

Default: 0

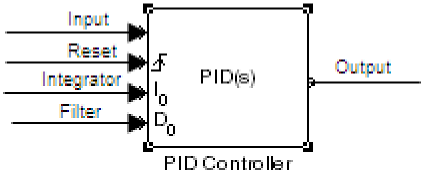
Simulink does not permit the filter initial condition to be `inf` or `NaN`.

External reset

Select the trigger event that resets the integrator and filter outputs to the initial conditions you specify in the **Integrator Initial condition** and **Filter Initial condition** fields. Selecting any option other than `none` enables a reset input on the block for the external reset signal, as shown:


Input
Reset
PID(s)
Output
PID Controller

Or, if the **Initial conditions Source** is `External`,


Input
Reset
Integrator
Filter
I_0
D_0
PID(s)
Output
PID Controller

Settings

`none` (Default)

Does not reset the integrator and filter outputs to initial conditions.

`rising`

Resets the outputs when the reset signal has a rising edge.

`falling`

Resets the outputs when the reset signal has a falling edge.

`either`

Resets the outputs when the reset signal either rises or falls.

`level`

Resets and holds the outputs to the initial conditions while the reset signal is nonzero.

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.

Ignore reset when linearizing

Force Simulink linearization commands to ignore any reset mechanism that you have chosen with the **External reset** menu. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the PID Controller block to reset.

Settings

☐ `Off` (Default)

Simulink linearization commands do not ignore states corresponding to the reset mechanism.

☑ `On`

Simulink linearization commands ignore states corresponding to the reset mechanism.

Enable zero-crossing detection

Enable zero-crossing detection in continuous-time models upon reset and upon entering or leaving a saturation state.

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate an **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Settings

☑ `On` (Default)

Uses zero-crossing detection at any of the following events: reset; entering or leaving an upper saturation state; and entering or leaving a lower saturation state.

☐ `Off`

Does not use zero-crossing detection.

Enabling zero-crossing detection for the PID Controller block also enables zero-crossing detection for all under-mask blocks that include the zero-crossing detection feature.

Limit output

Limit the block output to values you specify as the **Lower saturation limit** and **Upper saturation limit** parameters.

Activating this option limits the block output internally to the block, obviating the need for a separate Saturation block after the controller in your Simulink model. It also allows you to activate the block's built-in anti-windup mechanism (see "Anti-windup method" on page 1-1144).

Settings

☐ `Off` (Default)

Does not limit the block output, which equals the weighted sum of the proportional, integral, and derivative actions.

☑ `On`

Limits the block output to the **Lower saturation limit** or the **Upper saturation limit** whenever the weighted sum exceeds those limits. Allows you to select an **Anti-windup method**.

Lower saturation limit

(Available only when you select the **Limit output** check box.) Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

Default: `-inf`

Upper saturation limit

(Available only when you select the **Limit output** check box.) Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

Default: `inf`

Anti-windup method

(Available only when you select the **Limit output** option and the controller includes integral action.) Select an anti-windup mechanism to discharge the integrator when the block is saturated, which occurs when the sum of the block components exceeds the output limits.

When you select the **Limit output** check box and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator wind-up), increasing the difference between the block output and the sum of the block components. Without a mechanism to prevent integrator wind-up, two results are possible:

- If the sign of the input signal never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.
- If the sign of the input signal changes once the weighted sum has grown beyond the output limits, it can take a long time to discharge the integrator and return the weighted sum within the block saturation limit.

In both cases, controller performance can suffer. To combat the effects of wind-up without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. Activating an anti-windup mechanism can improve controller performance.

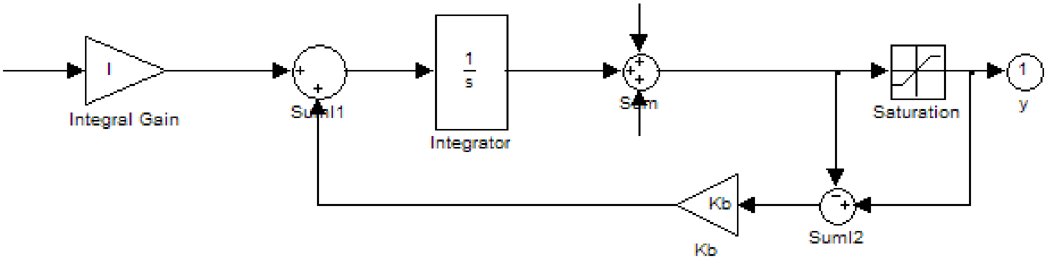
Settings

`none` (Default)

Does not use an anti-windup mechanism. This setting may cause the block's internal signals to be unbounded even if the output appears to be bounded by the saturation limits. This can result in slow recovery from saturation or unexpected overflows.

`back-calculation`

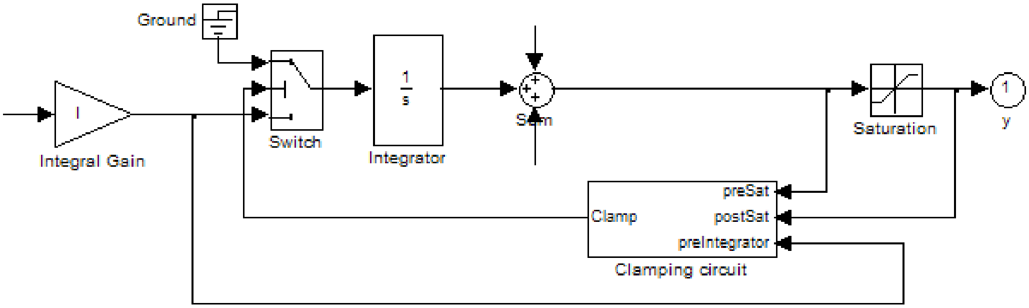
Discharges the integrator when the block output saturates using the integral-gain feedback loop:


I
Integral Gain
SumI1
1
s
Integrator
Sum
Saturation
1
y
Kb
Kb
SumI2

You can also specify a value for the **Back-calculation coefficient (Kb)**.

`clamping`

Stops integration when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Resumes integration when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. The integrator portion of the block is:


Ground
I
Integral Gain
Switch
1
s
Integrator
Sum
Saturation
1
y
Clamp
preSat
postSat
preIntegrator
Clamping circuit

The clamping circuit implements the logic necessary to determine whether integration continues.

Back-calculation gain (Kb)

(Available only when the `back-calculation` **Anti-windup method** is active.) Specify the gain coefficient of the anti-windup feedback loop.

The `back-calculation` anti-windup method discharges the integrator on block saturation using a feedback loop having gain coefficient `Kb`.

Default: 1

Ignore saturation when linearizing

Force Simulink linearization commands ignore PID Controller block output limits. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the PID Controller block to exceed the output limits.

Settings

☑ `On` (Default)

Simulink linearization commands ignore states corresponding to saturation.

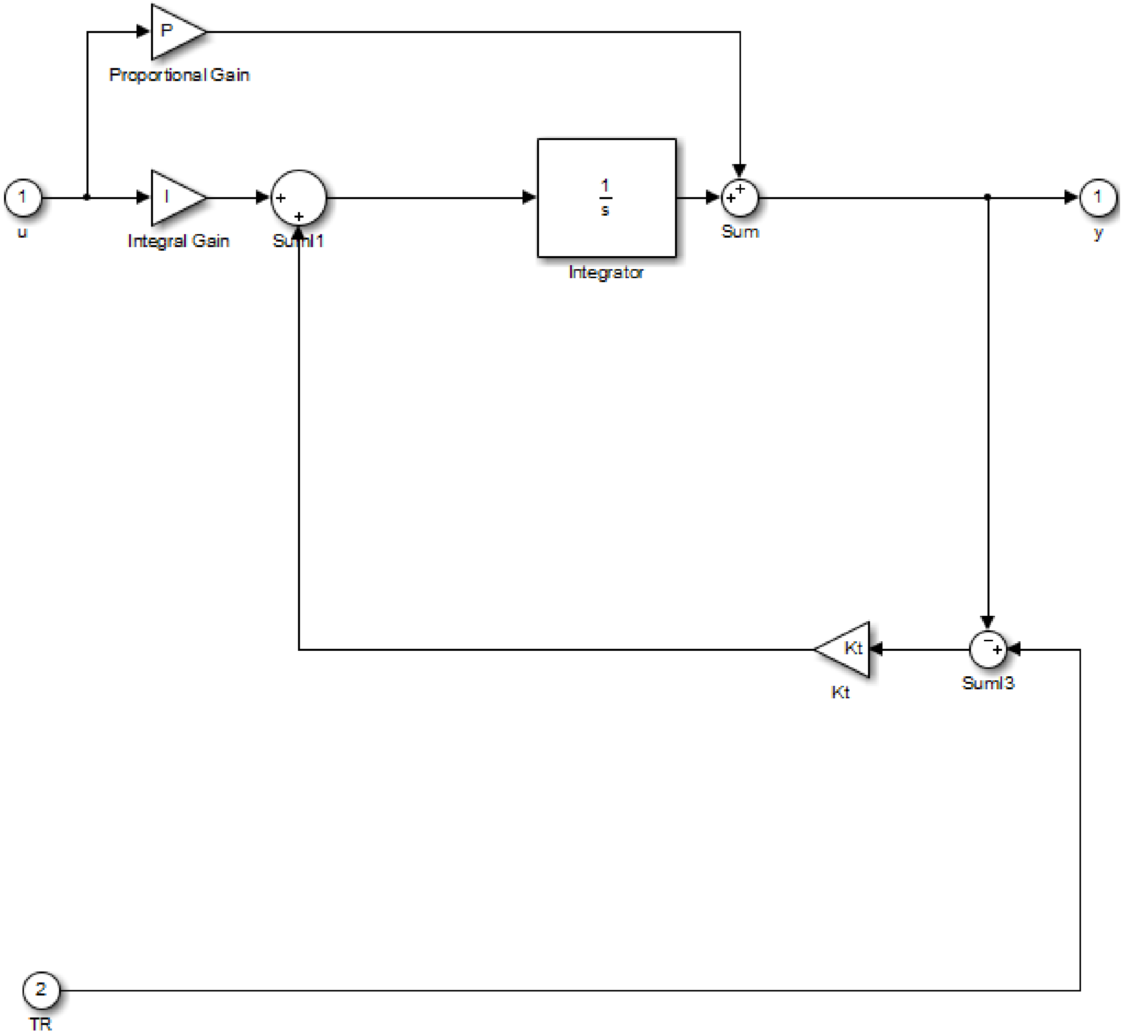
☐ `Off`

Simulink linearization commands do not ignore states corresponding to saturation.

Enable tracking mode

(Available for any controller with integral action.) Activate signal tracking, which lets the output of the PID Controller block follow a tracking signal. Provide the tracking signal to the block at the `TR` port, which becomes active when you select **Enable tracking mode**.

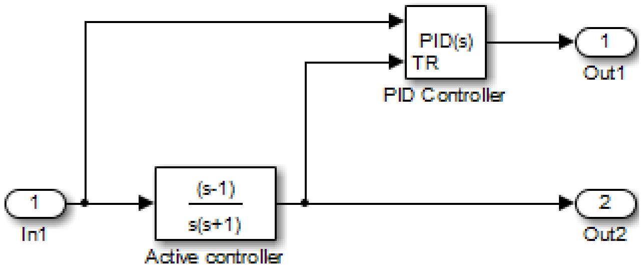
When signal tracking is active, the difference between the tracked signal and the block output is fed back to the integrator input with a gain `Kt`. The structure is illustrated for a PI controller:


P
Proportional Gain
1
u
I
Integral Gain
+
+
SumI1
1
s
Integrator
+
+
Sum
1
y
Kt
Kt
−
+
SumI3
2
TR

You can also specify the **Tracking coefficient (Kt)**.

Bumpless control transfer

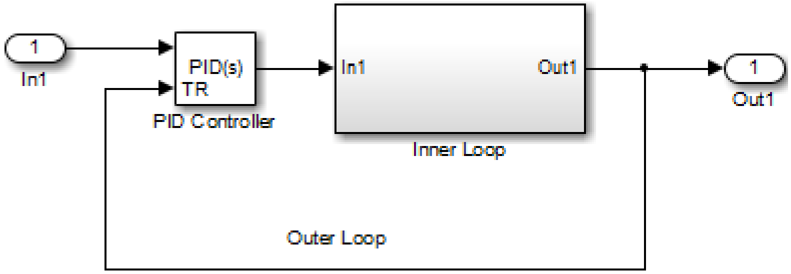
Use signal tracking, for example, to achieve bumpless control transfer in systems that switch between two controllers. You can make one controller track the output of the other controller by connecting the `TR` port to the signal you want to track. For example:


PID(s)
TR
PID Controller
1
Out1
1
In1
(s-1)
s(s+1)
Active controller
2
Out2

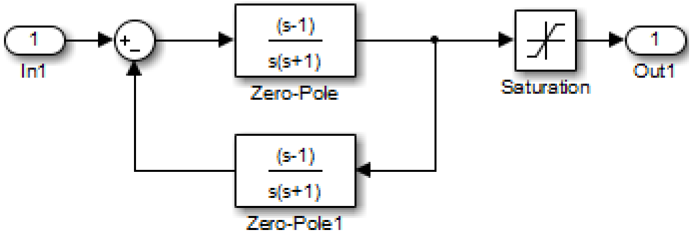
In this example, the outputs `Out1` and `Out2` can drive a controlled system (not shown) through a switch that transfers control between the "Active controller" block and the PID Controller block. The signal tracking feature of the PID Controller block provides smooth operation upon transfer of control from one controller to another, ensuring that the two controllers have the same output at the time of transfer.

Multiloop control

Use signal tracking to prevent block wind-up in multiloop control approaches, as this example illustrates:


1
In1
PID(s)
TR
PID Controller
In1
Out1
Inner Loop
1
Out1
Outer Loop

The inner-loop subsystem contains the following blocks:


1
In1
+
_
(s-1)
s(s+1)
Zero-Pole
Saturation
1
Out1
(s-1)
s(s+1)
Zero-Pole1

In this example, the inner loop has an effective gain of 1 when it does not saturate. Without signal tracking, the inner loop winds up in saturation. Signal tracking ensures that the PID Controller output does not exceed the saturated output of the inner loop.

Settings

☐ `Off` (Default)

Disables signal tracking and removes `TR` block input.

☑ `On`

Enables signal tracking and activates `TR` input.

Tracking gain (Kt)

(Available only when you select **Enable tracking mode**.) Specify `Kt`, which is the gain of the signal tracking feedback loop.

Default: 1

Parameter data type

Select the data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt**.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit:  Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit:  Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`Inherit:  Same as input`

Use data type of input signal.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
: Name of a data type object. For example, `Simulink.NumericType`.

Product output data type

Select the product output data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt** .

See "Data Types Supported by Simulink" in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit:  Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit:  Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`Inherit:  Same as input`

Use data type of input signal.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Summation output data type

Select the summation output data type of the sums **Sum**, **Sum D**, **Sum I1** , **SumI2** ,and **SumI3**, which are sums computed internally within the block. To see where Simulink computes each of these sums , right-click the PID Controller block in your model and select `Look Under Mask`:

- **Sum** is the weighted sum of the proportional, derivative, and integral signals.
- **SumD** is the sum in the derivative filter feedback loop.
- **SumI1** is the sum of the block input signal (weighted by the integral gain `I`) and **SumI2**. **SumI1** is computed only when **Limit output** and **Anti-windup method** `back-calculation` are active.
- **SumI2** is the difference between the weighted sum **Sum** and the limited block output. **SumI2** is computed only when **Limit output** and **Anti-windup method** `back-calculation` are active.
- **SumI3** is the difference between the block output and the signal at the block's tracking input. **SumI3** is computed only when you select the **Enable tracking mode** box.

See "Data Types Supported by Simulink" in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit:  Same as first input`.

- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

Note The accumulator internal rule favors greater numerical accuracy, possibly at the cost of less efficient generated code. To get the same accuracy for the output, set the output data type to `Inherit:  Same as accumulator`.

`Inherit:  Inherit via back propagation`
Use data type of the driving block.

`Inherit:  Same as first input`
Use data type of first input signal.

`Inherit:  Same as accumulator`
Use the same data type as the corresponding accumulator.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Accumulator data type

Specify the accumulator data type.

Settings

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`
Use internal rule to determine accumulator data type.

`Inherit: Same as first input`
Use data type of first input signal.

`double`
Accumulator data type is `double`.

`single`
Accumulator data type is `single`.

`int8`
Accumulator data type is `int8`.

`uint8`
Accumulator data type is `uint8`.

`int16`
Accumulator data type is `int16`.

`uint16`
Accumulator data type is `uint16`.

`int32`
Accumulator data type is `int32`.

`uint32`
Accumulator data type is `uint32`.

`fixdt(1,16,0)`
Accumulator data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
Accumulator data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
: The name of a data type object, for example `Simulink.NumericType`

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specify Data Types Using Data Type Assistant”.

Integrator output data type

Select the data type of the integrator output.

See "Data Types Supported by Simulink" in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use `Inherit:  Inherit via back propagation`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`

Name of a data type object. For example, `Simulink.NumericType`.

Filter output data type

Select the data type of the filter output.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use `Inherit:  Inherit via back propagation`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
: Name of a data type object. For example, `Simulink.NumericType`.

Saturation output data type

Select the saturation output data type.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit:  Same as input` (Default)
    Use data type of input signal.

`Inherit:  Inherit via back propagation`
    Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
    Name of a data type object. For example, `Simulink.NumericType`.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule` (default)
- `Inherit via back propagation`
- `Same as first input`
- `Same as accumulator`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`
: Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`
: Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

  - `Inherit via back propagation`
  - `Same as input` (default)

`Built in`
: Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

  - `double` (default)
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`
  - `uint32`

`Fixed point`
: Fixed-point data types.

`Expression`
: Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of accumulator data to specify

Settings

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit via internal rule` (default)
- `Same as first input`

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Specifies fixed-point data types.

`Expression`

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Dependency

Clicking the **Show data type assistant** button for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specify the fixed-point data as signed.

`Unsigned`
: Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Signedness

Specify whether you want the fixed-point data to be signed or unsigned.

Settings

Default: `Signed`

`Signed`
  Specify the fixed-point data to be signed.

`Unsigned`
  Specify the fixed-point data to be unsigned.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`, `Binary point`, `Integer`

`Binary point`
    Specify binary point location.

`Slope and bias`
    Enter slope and bias.

`Best precision`
    Specify best-precision values. This option appears for some blocks.

`Integer`
    Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0. This option appears for some blocks.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Binary point`

`Binary point`
    Specify binary point location.

`Slope and bias`
    Enter slope and bias.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Word length

Specify the bit size of the word that will hold the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see “Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

State name

Assign unique name to each state. The state names apply only to the selected block.

To assign a name to a single state, enter the name between quotes; for example, `'velocity'`.

To assign names to multiple states, enter a comma-delimited list surrounded by braces; for example, `{'a', 'b', 'c'}`. Each name must be unique. To assign state names with a variable that has been defined in the MATLAB workspace, enter the variable without quotes. The variable can be a string, cell, or structure.

Settings

Default: `' '` (no name)

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Settings

Default: Off

☑ On

Require that state name resolve to Simulink signal object.

☐ Off

Do not require that state name resolve to Simulink signal object.

Dependencies

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Code generation storage class

Select state storage class.

Settings

Default: `Auto`

`Auto`
  Auto is the appropriate storage class for states that you do not need to interface to external code.

`ExportedGlobal`
  State is stored in a global variable

`ImportedExtern`
  `model_private.h` declares the state as an extern variable.

`ImportedExternPointer`
  `model_private.h` declares the state as an extern pointer.

Dependencies

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

"State Storage Classes" in the Simulink Coder documentation.

Code generation storage type qualifier

Specify the Simulink Coder storage type qualifier.

Settings

Default: `' '`

If left blank, no qualifier is assigned.

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Direct Feedthrough	The following ports support direct feedthrough: • Reset port • Integrator and filter initial condition port • Input port, for every integration method except Forward Euler
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Supported for gain parameters **P**, **I**, and **D** and for filter coefficient **N**
States	Inherited from driving block and parameters
Dimensionalized	Yes
Zero-Crossing Detection	Yes (in continuous-time domain)

See Also

PID Controller (2 DOF), Gain, Integrator, Discrete-Time Integrator, Derivative, Discrete Derivative.

Purpose

Simulate continuous- or discrete-time two-degree-of-freedom PID controllers

Library

Continuous, Discrete

Description

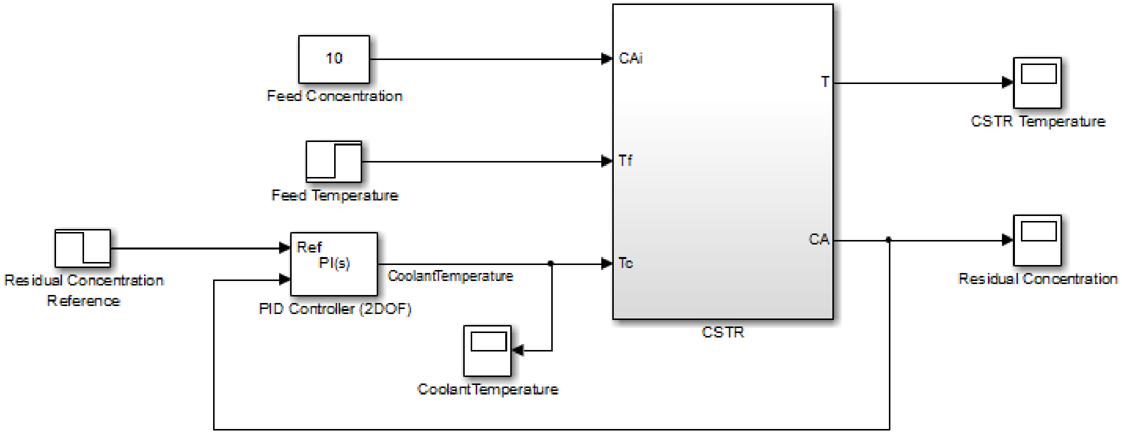
Implement a continuous- or discrete-time two-degree-of-freedom controller (PID, PI, or PD) in your Simulink model. The PID Controller (2DOF) block allows you to implement setpoint weighting in your controller to achieve both smooth setpoint tracking and good disturbance rejection.

The PID Controller (2DOF) block generates an output signal based on the difference between a reference signal and a measured system output. The block computes a weighted difference signal for each of the proportional, integral, and derivative actions according to the setpoint weights you specify. The block output is the sum of the proportional, integral, and derivative actions on the respective difference signals, where each action is weighted according to the gain parameters. A first-order pole filters the derivative action. Controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software (PID Tuner or SISO Design Tool).

Configurable options in the PID Controller (2DOF) block include:

- Controller type (PID, PI, or PD)
- Controller form (Parallel or Ideal)
- Time domain (continuous or discrete)
- Initial conditions and reset trigger
- Output saturation limits and built-in anti-windup mechanism
- Signal tracking for bumpless control transfer and multiloop control

In one common implementation, the PID Controller (2DOF) block operates in the feedforward path of the feedback loop. The block receives a reference signal at the `Ref` input and a measured system output at the other input. For example:


10
Feed Concentration
Feed Temperature
Residual Concentration
Reference
Ref
PI(s)
PID Controller (2DOF)
CoolantTemperature
CoolantTemperature
CAi
Tf
Tc
T
CA
CSTR
CSTR Temperature
Residual Concentration

For a single-input block that accepts an error signal (a difference between a setpoint and a system output), see the Discrete PID Controller block reference page.

You can generate code to implement your controller using any Simulink data type, including fixed-point data types. (Code generation requires Simulink Coder software; fixed-point implementation requires the Fixed-Point Designer product.)

For an example illustrating an application of the PID Controller (2 DOF) block, see the Simulink example Two Degree-of-Freedom PID Control for Setpoint Tracking.

Data Type Support

The PID Controller (2DOF) block accepts real signals of any numeric data type that Simulink software supports, including fixed-point data types. See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Parameters

The following table summarizes the PID Controller (2DOF)block parameters, accessible via the block parameter dialog box.

Task	Parameters
Choose controller form and type.	• **Controller Form** in Main tab • **Controller**
Choose discrete or continuous time.	• **Time-domain** • **Sample time**
Choose an integration method (discrete time).	• **Integrator method** • **Filter method**
Set and tune controller gains.	• **Proportional (P)** in Main tab • **Integral (I)** in Main tab • **Derivative (D)** in Main tab • **Filter coefficient (N)** in Main tab • **Setpoint weight (b)** in Main tab • **Setpoint weight (c)** in Main tab
Set integrator and filter initial conditions.	• **Initial conditions Source** in Main tab • **Integrator Initial condition** in Main tab • **Filter Initial condition** in Main tab • **External reset** in Main tab • **Ignore reset when linearizing** in Main tab

Task	Parameters
Limit block output.	• **Limit output** in PID Advanced tab • **Lower saturation limit** in PID Advanced tab • **Upper saturation limit** in PID Advanced tab • **Ignore saturation when linearizing** in PID Advanced tab
Configure anti-windup mechanism (when you limit block output).	• **Anti-windup method** in PID Advanced tab • **Back-calculation gain (Kb)** in PID Advanced tab
Enable signal tracking.	• **Enable tracking mode** in PID Advanced tab • **Tracking gain (Kt)** in PID Advanced tab
Configure data types.	• **Parameter data type** in Data Type Attributes tab • **Product output data type** in Data Type Attributes tab • **Summation output data type** in Data Type Attributes tab • **Accumulator data type** in Data Type Attributes tab • **Integrator output data type** in Data Type Attributes tab • **Filter output data type** in Data Type Attributes tab

Task	Parameters
	- **Saturation output data type** in Data Type Attributes tab - **Lock output data type setting against changes by the fixed-point tools** in Data Type Attributes tab - **Saturate on integer overflow** in Data Type Attributes tab - **Integer rounding mode** in Data Type Attributes tab
Configure block for code generation.	- **State name** in State Attributes tab - **State name must resolve to Simulink signal object** in State Attributes tab - **Code generation storage class** in State Attributes tab - **Code generation storage type qualifier** in State Attributes tab

Controller form

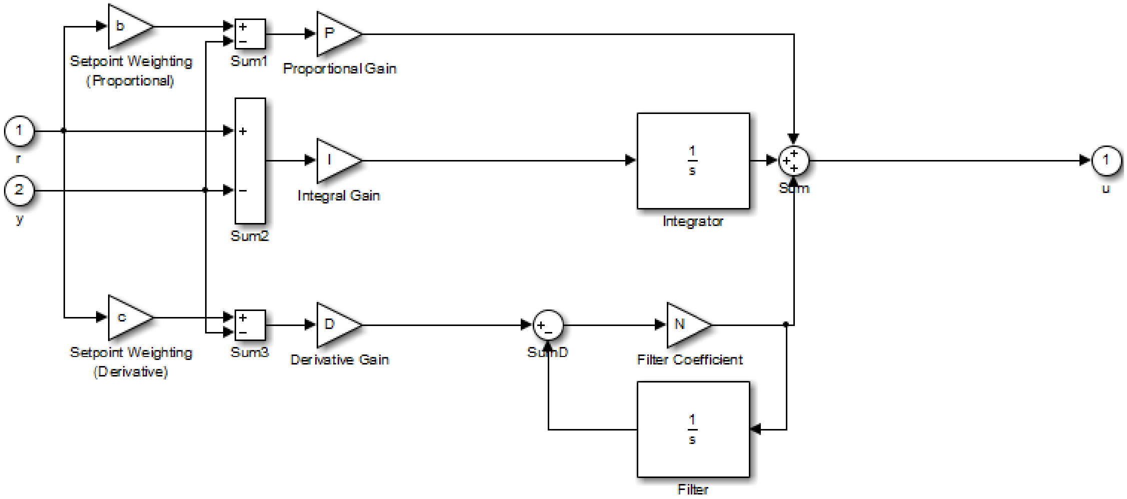
Select the controller form.

Settings

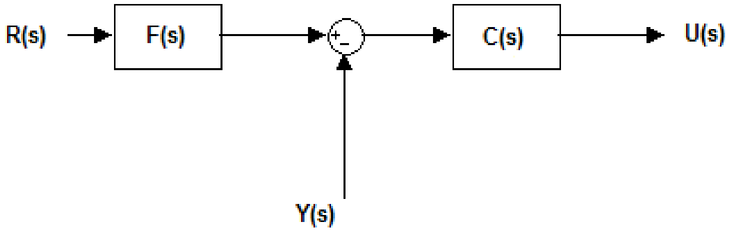
`Parallel` (Default)

Selects a controller form in which the proportional, integral, and derivative gains **P**, **I**, and **D** operate independently. The filter coefficient **N** sets the location of the pole in the derivative filter.

Parallel two-degree-of-freedom PID controller, where input 1 receives a reference signal and input 2 receives feedback from the measured system output:


b
Setpoint Weighting (Proportional)
Sum1
P
Proportional Gain
1
r
2
y
Sum2
I
Integral Gain
1/s
Integrator
Sum
1
u
c
Setpoint Weighting (Derivative)
Sum3
D
Derivative Gain
SumD
N
Filter Coefficient
1/s
Filter

The parallel two-degree-of-freedom PID controller can be equivalently modeled by the following block diagram:



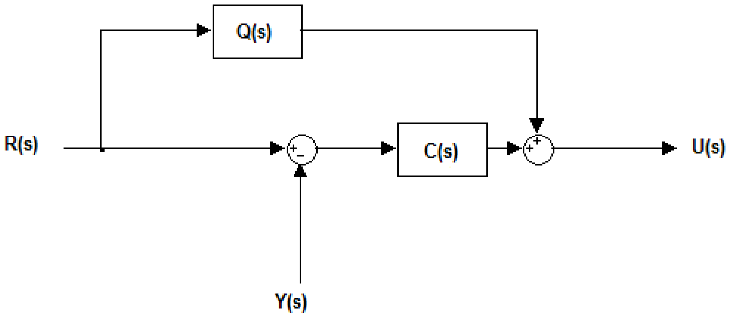
`R(s)` represents the reference signal and `Y(s)` represents the feedback from measured system output. In this model, `C(s)` is a single degree-of-freedom controller, and `F(s)` acts as a prefilter on the reference signal. For a parallel two-degree-of-freedom PID controller in the `Continuous-time` **Time-domain**, the transfer functions `F(s)` and `C(s)` are:

$$F_{par}(s) = \frac{(bP + cDN)s^2 + (bPN + I)s + IN}{(P + DN)s^2 + (PN + I)s + IN}$$

$$C_{par}(s) = \frac{(P + DN)s^2 + (PN + I)s + IN}{s(s + N)}$$

where `b` and `c` are the **Setpoint weight** parameters.

Alternatively, the parallel two-degree-of-freedom PID controller can be modeled by the following block diagram:



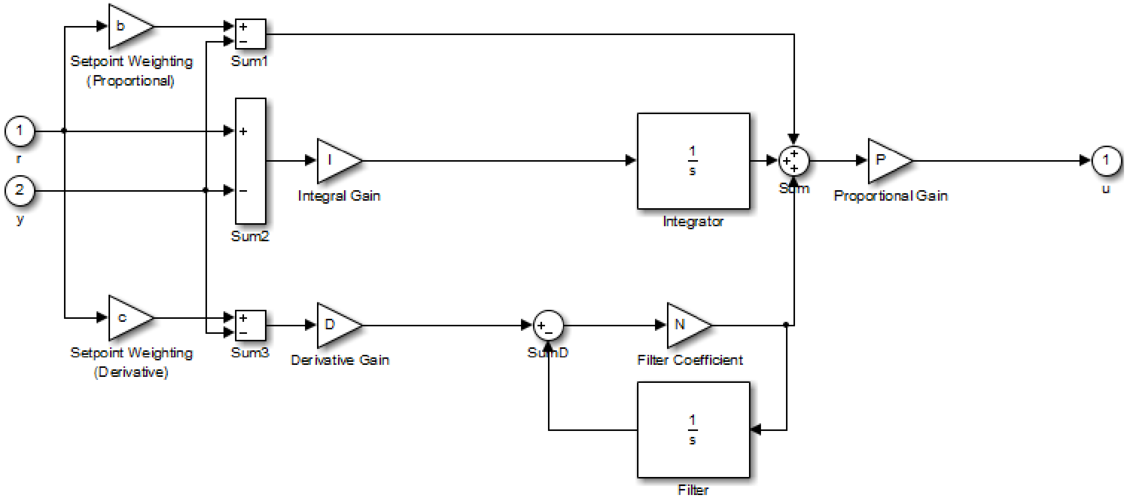
`R(s)`, `Y(s)`, and `C(s)` are as discussed previously. In this realization, `Q(s)` acts as feed-forward conditioning on the reference signal `R(s)`. For a parallel PID controller in the `Continuous-time` **Time-domain**, the transfer function `Q(s)` is:

$$Q_{par}(s) = \frac{\big((b-1)P + (c-1)DN\big)s + (b-1)PN}{s+N}$$

`Ideal`

Selects a controller form in which the proportional gain `P` acts on the sum of all actions.

Ideal two-degree-of-freedom PID controller, where input 1 receives a reference signal and input 2 receives feedback from the measured system output:


b
Setpoint Weighting (Proportional)
Sum1
1
r
2
y
Sum2
I
Integral Gain
1/s
Integrator
Sum
P
Proportional Gain
1
u
c
Setpoint Weighting (Derivative)
Sum3
D
Derivative Gain
SumD
N
Filter Coefficient
1/s
Filter

Similarly to the parallel controller form discussed previously, the ideal two-degree-of-freedom PID controller can be modeled as a single degree-of-freedom controller `C(s)` with a prefilter `F(s)`. For an ideal two-degree-of-freedom PID controller in the `Continuous-time` **Time-domain**, the transfer functions `F(s)` and `C(s)` are:

$$F_{id}(s) = \frac{(b + cDN)s^2 + (bN + I)s + IN}{(1 + DN)s^2 + (N + I)s + IN}$$

$$C_{id}(s) = P\frac{(1 + DN)s^2 + (N + I)s + IN}{s(s + N)}$$

where `b` and `c` are the **Setpoint weight** parameters.

Alternatively, modeling the ideal two-degree-of-freedom PID controller as a one-degree-of-freedom controller `C(s)` with feed-forward conditioning `Q(s)` on the reference signal gives, in continuous-time:

$$Q_{id}(s) = P\frac{\left((b-1)+(c-1)DN\right)s+(b-1)N}{s+N}$$

The controller transfer function for the current settings is displayed in the block dialog box.

Controller

Specify the controller type.

Settings

`PID` (Default)

Implements a controller with proportional, integral, and derivative action.

`PI`

Implements a controller with proportional and integral action.

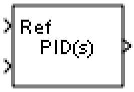
`PD`

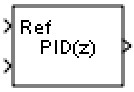
Implements a controller with proportional and derivative action.

The controller transfer function for the current settings is displayed in the block dialog box.

Time-domain

Select continuous or discrete time domain. The appearance of the block changes to reflect your selection.





Settings

`Continuous-time` (Default)
: Selects the continuous-time representation.

`Discrete-time`
: Selects the discrete-time representation. Selecting `Discrete-time` also allows you to specify the:

  - **Sample time**, which is the discrete interval between samples.
  - Discrete integration methods for the integrator and the derivative filter using the **Integrator method** and **Filter method** menus.

Integrator method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the integrator output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Settings

`Forward Euler` (Default)

Selects the Forward Rectangular (left-hand) approximation.

- This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the `Forward Euler` method can result in instability, even when discretizing a system that is stable in continuous time.

`Backward Euler`

Selects the Backward Rectangular (right-hand) approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product and you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.
- An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

`Trapezoidal`

Selects the Bilinear approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product and you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

- 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.

Filter method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the derivative filter output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Settings

`Forward Euler` (Default)

Selects the Forward Rectangular (left-hand) approximation.

- This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the `Forward Euler` method can result in instability, even when discretizing a system that is stable in continuous time.

`Backward Euler`

Selects the Backward Rectangular (right-hand) approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product, this filter method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see "Algebraic Loops" in the Simulink documentation.
- An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method.

`Trapezoidal`

Selects the Bilinear approximation.

- If you are generating code using Simulink Coder software or the Fixed-Point Designer product, this filter method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see "Algebraic Loops" in the Simulink documentation.

- An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method. Of all available filter methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Sample time (-1 for inherited)

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the discrete interval between samples.

Settings

Default: 1

By default, the block uses a discrete sample time of 1. To specify a different sample time, enter another discrete value, such as 0.1.

If you specify a value of –1, the PID Controller (2DOF) block inherits the sample time from upstream blocks. Do not enter a value of 0; to implement a continuous-time controller, select the **Time-domain** `Continuous-time`.

See “Specify Sample Time” in the online documentation for more information.

Proportional (P)

Specify the proportional gain `P`.

Default: 1

Enter a finite, real gain value into the **Proportional (P)** field. Use either scalar or vector gain values. For a `parallel` PID **Controller form**, the proportional action is independent of the integral and derivative actions. For an `ideal` PID **Controller form**, the proportional action acts on the integral and derivative actions. See "Controller form" on page 1-1200 for more information about the role of `P` in the controller transfer function.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See "Choosing a Compensator Design Approach".

Integral (I)

(Available for PID and PI controllers.) Specify the integral gain `I`.

Default: 1

Enter a finite, real gain value into the **Integral (I)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See “Choosing a Compensator Design Approach”.

Derivative (D)

(Available for PID and PD controllers.) Specify the derivative gain `D`.

Default: 0

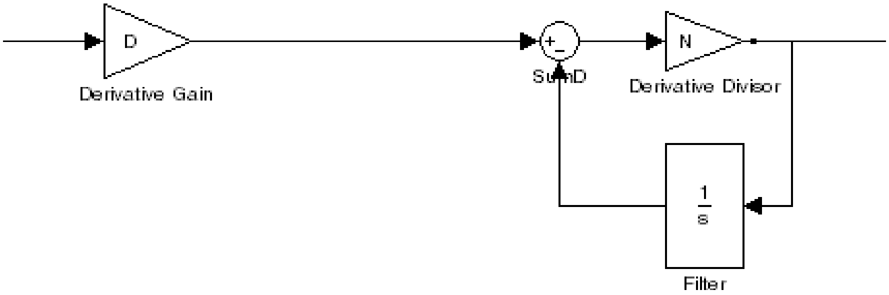
Enter a finite, real gain value into the **Derivative (D)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See “Choosing a Compensator Design Approach”.

Filter coefficient (N)

Specifies the filter coefficient of the controller.

(Available for PID and PD controllers.) Specify the filter coefficient `N`, which determines the pole location of the filter in the derivative action:



The filter pole falls at `s = -N` in the `Continuous-time` **Time-domain**. For `Discrete-time`, the location of the pole depends on which **Filter method** you select (for sampling time `T`$_s$):

- `Forward Euler`:

  $$z_{pole} = 1 - NT_s$$

- `Backward Euler`:

  $$z_{pole} = \frac{1}{1 + NT_s}$$

- `Trapezoidal`:

  $$z_{pole} = \frac{1 - NT_s/2}{1 + NT_s/2}$$

Default: 100.

Enter a finite, real gain value into the **Filter Coefficient (N)** field. Use either scalar or vector gain values. Note that the PID controller (2DOF) block does not support `N = inf` (ideal unfiltered derivative).

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See "Choosing a Compensator Design Approach". Automatic tuning requires `N > 0`.

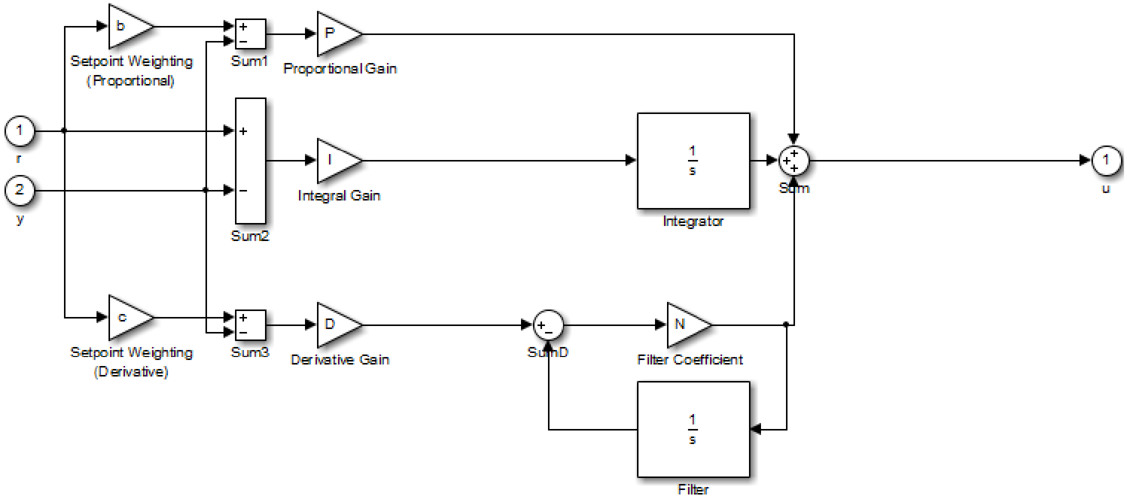
Setpoint weight (b)

Specify the proportional setpoint weight `b`.

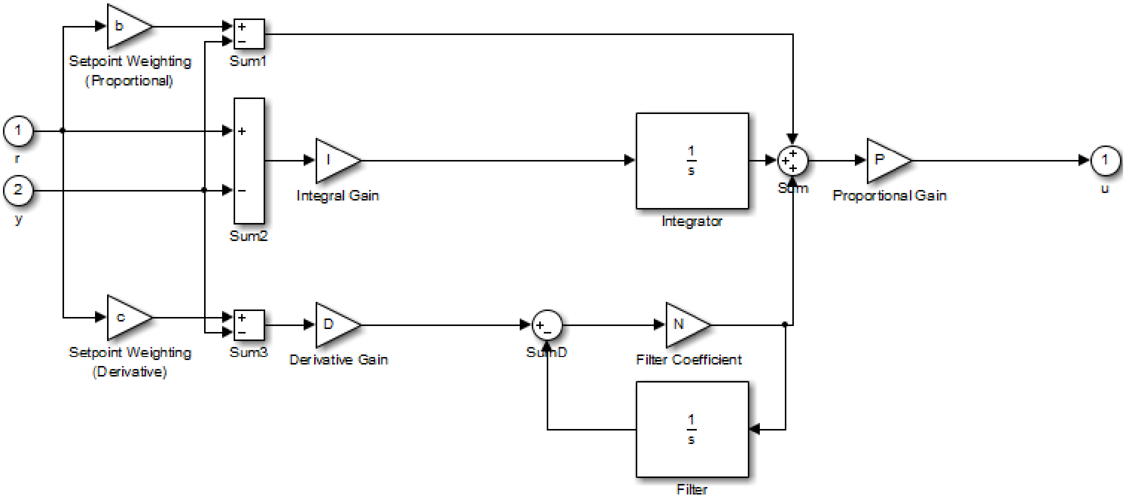
Default: 1

Enter the proportional setpoint weight value into the **Setpoint weight (b)** field. Setting `b = 0` eliminates the proportional action on the reference signal, which can reduce overshoot in the system response to step changes in the setpoint.

The following diagrams show the role of **Setpoint weight (b)** in `Parallel` and `Ideal` PID controllers. See "Controller form" on page 1-1200 for a discussion of the corresponding transfer functions.


b
Setpoint Weighting
(Proportional)
Sum1
P
Proportional Gain
1
r
2
y
Sum2
I
Integral Gain
1/s
Integrator
Sum
1
u
c
Setpoint Weighting
(Derivative)
Sum3
D
Derivative Gain
SumD
N
Filter Coefficient
1/s
Filter

Parallel Two-Degree-of-Freedom PID Controller


b
Setpoint Weighting
(Proportional)
Sum1
1
r
2
y
Sum2
I
Integral Gain
1/s
Integrator
Sum
P
Proportional Gain
1
u
c
Setpoint Weighting
(Derivative)
Sum3
D
Derivative Gain
SumD
N
Filter Coefficient
1/s
Filter

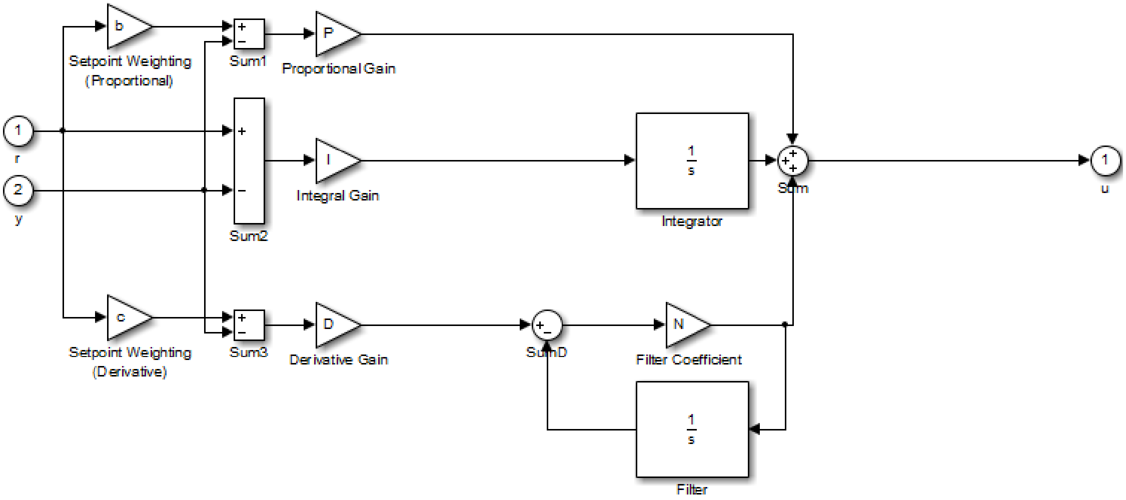
Ideal Two-Degree-of-Freedom PID Controller

Setpoint weight (c)

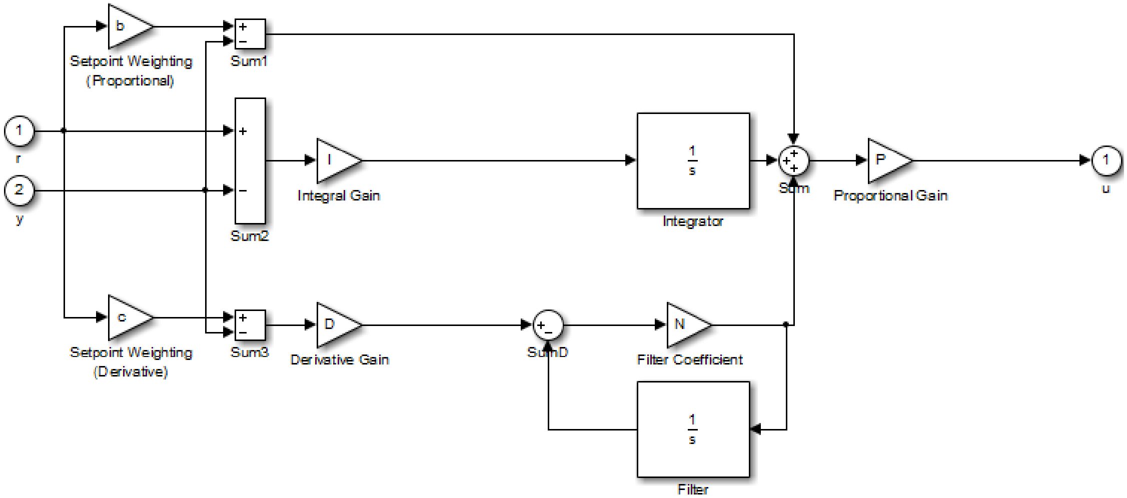
(Available for PID and PD controllers.) Specify the derivative setpoint weight `c`.

Enter the derivative setpoint weight value into the **Setpoint weight (c)** field. To implement a controller that achieves both effective disturbance rejection and smooth setpoint tracking without excessive transient response, set `c = 0`. Setting `c = 0` yields a controller with derivative action on the measured system response but not on the reference input.

The following diagrams show the role of **Setpoint weight (c)** in `Parallel` and `Ideal` PID controllers. See "Controller form" on page 1-1200 for a discussion of the corresponding transfer functions.


b
Setpoint Weighting
(Proportional)
Sum1
P
Proportional Gain
1
r
2
y
Sum2
I
Integral Gain
1/s
Integrator
Sum
1
u
c
Setpoint Weighting
(Derivative)
Sum3
D
Derivative Gain
SumD
N
Filter Coefficient
1/s
Filter

Parallel Two-Degree-of-Freedom PID Controller


b
Setpoint Weighting
(Proportional)
+
−
Sum1
1
r
2
y
+
−
Sum2
I
Integral Gain
1
s
Integrator
+
+
+
Sum
P
Proportional Gain
1
u
c
Setpoint Weighting
(Derivative)
+
−
Sum3
D
Derivative Gain
+
−
SumD
N
Filter Coefficient
1
s
Filter

Ideal Two-Degree-of-Freedom PID Controller

Initial conditions Source

Select the source of the integrator and filter initial conditions. Simulink uses initial conditions to initialize the integrator and filter output at the start of a simulation or at a specified trigger event (see "External reset" on page 1-1224). The integrator and filter initial conditions in turn determine the initial block output.

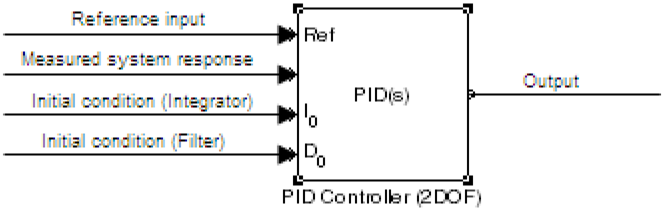
Settings

`internal` (Default)

Specifies the integrator and filter initial conditions explicitly using the **Integrator Initial condition** and **Filter Initial condition** parameters.

`external`

Specifies the integrator and filter initial conditions externally. An additional input port appears under the block inputs for each initial condition: `I`$_0$ for the integrator and `D`$_0$ for the filter:


Reference input
Measured system response
Initial condition (Integrator)
Initial condition (Filter)
Ref
I_0
D_0
PID(s)
Output
PID Controller (2DOF)

Integrator Initial condition

(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the integrator initial value. Simulink uses the initial condition to initialize the integrator output at the start of a simulation or at a specified trigger event (see "External reset" on page 1-1224). The integrator initial condition, together with the filter initial condition, determines the initial output of the PID Controller (2DOF) block.

Default: 0

Simulink does not permit the integrator initial condition to be `inf` or `NaN`.

Filter Initial condition

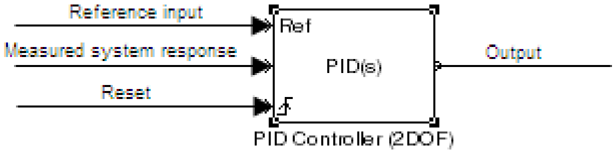
(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the filter initial value. Simulink uses the initial condition to initialize the filter output at the start of a simulation or at a specified trigger event (see "External reset" on page 1-1224). The filter initial condition, together with the integrator initial condition, determines the initial output of the PID Controller (2DOF) block.

Default: 0

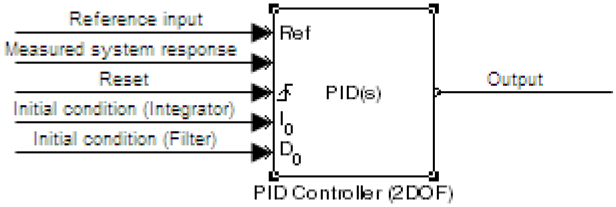
Simulink does not permit the filter initial condition to be `inf` or `NaN`.

External reset

Select the trigger event that resets the integrator and filter outputs to the initial conditions you specify in the **Integrator Initial condition** and **Filter Initial condition** fields. Selecting any option other than `none` enables a reset input on the block for the external reset signal, as shown:


Reference input
Measured system response
Reset
Ref
PID(s)
Output
PID Controller (2DOF)

Or, if the **Initial conditions Source** is `External`:


Reference input
Measured system response
Reset
Initial condition (Integrator)
Initial condition (Filter)
Ref
PID(s)
I_0
D_0
Output
PID Controller (2DOF)

Settings

`none` (Default)

Does not reset the integrator and filter outputs to initial conditions.

`rising`

Resets the outputs when the reset signal has a rising edge.

`falling`

Resets the outputs when the reset signal has a falling edge.

`either`
: Resets the outputs when the reset signal either rises or falls.

`level`
: Resets and holds the outputs to the initial conditions while the reset signal is nonzero.

Note To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller (2DOF) block.

Ignore reset when linearizing

Force Simulink linearization commands to ignore any reset mechanism that you have chosen with the **External reset** menu. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the PID Controller (2DOF) block to reset.

Settings

☐ `Off` (Default)

Simulink linearization commands do not ignore states corresponding to the reset mechanism.

☑ `On`

Simulink linearization commands ignore states corresponding to the reset mechanism.

Enable zero-crossing detection

Enable zero-crossing detection in continuous-time models upon reset and upon entering or leaving a saturation state.

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate an **External reset** in your PID Controller (2DOF) block, activating zero-crossing detection can reduce computation time in your simulation. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Settings

☑ `On` (Default)

Uses zero-crossing detection at any of the following events: reset; entering or leaving an upper saturation state; and entering or leaving a lower saturation state.

☐ `Off`

Does not use zero-crossing detection.

Enabling zero-crossing detection for the PID Controller (2DOF) block also enables zero-crossing detection for all under-mask blocks that include the zero-crossing detection feature.

Limit output

Limit the block output to values you specify as the **Lower saturation limit** and **Upper saturation limit** parameters.

Activating this option limits the block output internally to the block, obviating the need for a separate Saturation block after the controller in your Simulink model. It also allows you to activate the built-in anti-windup mechanism (see "Anti-windup method" on page 1-1231).

Settings

☐ `Off` (Default)

Does not limit the block output, which is the weighted sum of the proportional, integral, and derivative actions.

☑ `On`

Limits the block output to the **Lower saturation limit** or the **Upper saturation limit** whenever the weighted sum exceeds those limits. Allows you to select an **Anti-windup method**.

Lower saturation limit

(Available only when you select the **Limit Output** box.) Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

Default: `-inf`

Upper saturation limit

(Available only when you select the **Limit Output** box.) Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

Default: `inf`

Anti-windup method

(Available only when you select the **Limit Output** option and the controller includes integral action.) Select an anti-windup mechanism to discharge the integrator when the block is saturated, which occurs when the sum of the block components exceeds the output limits.

When you select the **Limit output** check box and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator wind-up), increasing the difference between the block output and the sum of the block components. Without a mechanism to prevent integrator wind-up, two results are possible:

- If the sign of the input signal never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.
- If the sign of the input signal changes once the weighted sum has grown beyond the output limits, it can take a long time to discharge the integrator and return the weighted sum within the block saturation limit.

In both cases, controller performance can suffer. To combat the effects of wind-up without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. Activating an anti-windup mechanism can improve controller performance.

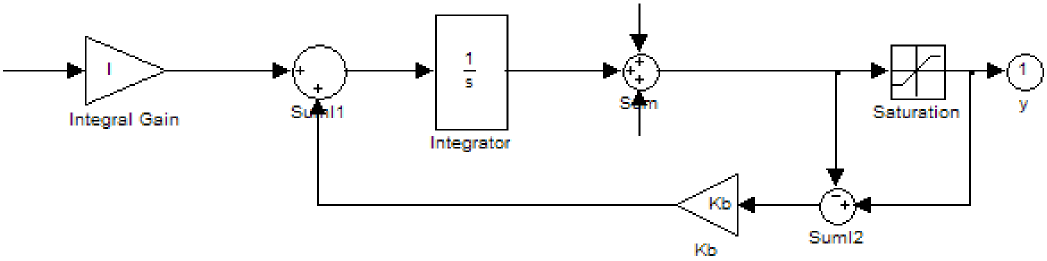
Settings

`none` (Default)

Does not use an anti-windup mechanism. This setting can cause the block's internal signals to be unbounded even if the output appears to be bounded by the saturation limits. This can result in slow recovery from saturation or unexpected overflows.

`back-calculation`

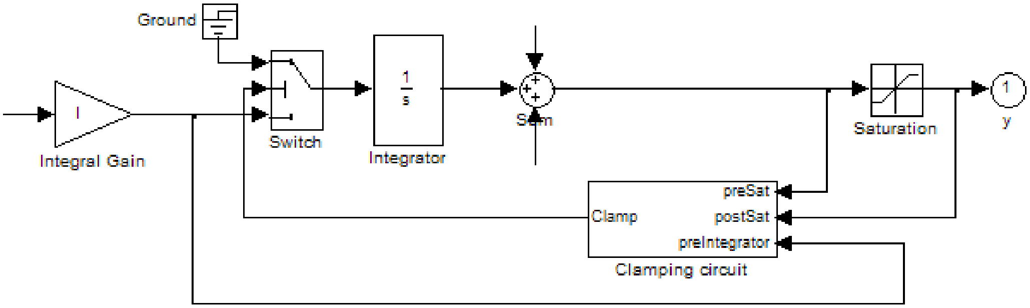
Discharges the integrator when the block output saturates using the integral-gain feedback loop:


I
Integral Gain
SumI1
1
s
Integrator
Sum
Saturation
1
y
Kb
Kb
SumI2

You can also specify a value for the **Back-calculation coefficient (Kb)**.

`clamping`

Stops integration when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Resumes integration when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. The integrator portion of the block is:


Ground
I
Integral Gain
Switch
1
s
Integrator
Sum
Saturation
1
y
Clamp
preSat
postSat
preIntegrator
Clamping circuit

The clamping circuit implements the logic necessary to determine whether integration continues.

Back-calculation gain (Kb)

(Available only when the `back-calculation` **Anti-windup method** is active.) Specify the gain coefficient of the anti-windup feedback loop.

The `back-calculation` anti-windup method discharges the integrator on block saturation using a feedback loop having gain coefficient `Kb`.

Default: 1

Ignore saturation when linearizing

Force Simulink linearization commands ignore PID Controller (2DOF) block output limits. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the PID Controller (2DOF) block to exceed the output limits.

Settings

☑ `On` (Default)

Simulink linearization commands ignore states corresponding to saturation.

☐ `Off`

Simulink linearization commands do not ignore states corresponding to saturation.

Enable tracking mode

(Available for any controller with integral action.) Activate signal tracking, which lets the output of the PID Controller (2DOF) block follow a tracking signal. Provide the tracking signal to the block at the `TR` port, which becomes active when you select **Enable tracking mode**.

When signal tracking is active, the difference between the tracked signal and the block output is fed back to the integrator input with a gain `Kt`. You can also specify the value of the **Tracking coefficient (Kt)**.

For information about using tracking mode to implement bumpless control transfer scenarios and multiloop controllers, see "Enable tracking mode" on page 1-1148 in the PID Controller reference page.

Settings

☐ `Off` (Default)
  Disables signal tracking and removes `TR` block input.

☑ `On`
  Enables signal tracking and activates `TR` input.

Tracking gain (Kt)

(Available only when you select **Enable tracking mode**.) Specify `Kt`, which is the gain of the signal tracking feedback loop.

Default: 1

Parameter data type

Select the data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt** and the setpoint weighting parameters **b** and **c**.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit:  Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit:  Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`Inherit:  Same as input`

Use data type of input signal.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Product output data type

Select the product output data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt** and the setpoint weighting parameters **b** and **c** .

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit: Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as input`

Use data type of input signal.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Summation output data type

Select the summation output data type of the sums **Sum**, **Sum1**, **Sum2**, **Sum3**, **Sum D**, **Sum I1** , **SumI2** ,and **SumI3**, which are sums computed internally within the block. To see where Simulink computes each of these sums , right-click the PID Controller (2DOF) block in your model and select `Look Under Mask`:

- **Sum** is the weighted sum of the proportional, derivative, and integral signals.
- **Sum1** is the difference between the reference input weighted by **b** and the measured system response.
- **Sum2** is the difference between the reference input weighted by **c** and the measured system response.
- **Sum3** is the difference between the unweighted reference input and the measured system response.
- **SumD** is the sum in the derivative filter feedback loop.
- **SumI1** is the sum of the block input signal (weighted by the integral gain `I`) and **SumI2**. **SumI1** is computed only when **Limit output** and **Anti-windup method** `back-calculation` are active.
- **SumI2** is the difference between the weighted sum **Sum** and the limited block output. **SumI2** is computed only when **Limit output** and **Anti-windup method** `back-calculation` are active.
- **SumI3** is the difference between the block output and the signal at the block's tracking input. **SumI3** is computed only when you select the **Enable tracking mode** box.

See "Data Types Supported by Simulink" in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit:  Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

Note The accumulator internal rule favors greater numerical accuracy, possibly at the cost of less efficient generated code. To get the same accuracy for the output, set the output data type to `Inherit:  Same as accumulator`.

`Inherit:  Inherit via back propagation`
Use data type of the driving block.

`Inherit:  Same as first input`
Use data type of first input signal.

`Inherit:  Same as accumulator`
Use the same data type as the corresponding accumulator.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Accumulator data type

Specify the accumulator data type.

Settings

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`
Use internal rule to determine accumulator data type.

`Inherit: Same as first input`
Use data type of first input signal.

`double`
Accumulator data type is `double`.

`single`
Accumulator data type is `single`.

`int8`
Accumulator data type is `int8`.

`uint8`
Accumulator data type is `uint8`.

`int16`
Accumulator data type is `int16`.

`uint16`
Accumulator data type is `uint16`.

`int32`
Accumulator data type is `int32`.

`uint32`
Accumulator data type is `uint32`.

`fixdt(1,16,0)`
Accumulator data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
Accumulator data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
: The name of a data type object, for example `Simulink.NumericType`

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specify Data Types Using Data Type Assistant".

Integrator output data type

Select the data type of the integrator output.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use `Inherit:  Inherit via back propagation`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Filter output data type

Select the data type of the filter output.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit:  Inherit via internal rule` (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use `Inherit:  Inherit via back propagation`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Saturation output data type

Select the saturation output data type.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Settings

`Inherit: Same as input` (Default)
Use data type of input signal.

`Inherit: Inherit via back propagation`
Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule` (default)
- `Inherit via back propagation`
- `Same as first input`
- `Same as accumulator`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via back propagation`
- `Same as input` (default)

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of accumulator data to specify

Settings

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit via internal rule` (default)
- `Same as first input`

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Specifies fixed-point data types.

`Expression`

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Dependency

Clicking the **Show data type assistant** button for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Signedness

Specify whether you want the fixed-point data to be signed or unsigned.

Settings

Default: `Signed`

`Signed`
  Specify the fixed-point data to be signed.

`Unsigned`
  Specify the fixed-point data to be unsigned.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`, `Binary point`, `Integer`

`Binary point`
  Specify binary point location.

`Slope and bias`
  Enter slope and bias.

`Best precision`
  Specify best-precision values. This option appears for some blocks.

`Integer`
  Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0. This option appears for some blocks.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Binary point`

`Binary point`
  Specify binary point location.

`Slope and bias`
  Enter slope and bias.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that will hold the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

State name

Assign unique name to each state. The state names apply only to the selected block.

To assign a name to a single state, enter the name between quotes; for example, `'velocity'`.

To assign names to multiple states, enter a comma-delimited list surrounded by braces; for example, `{'a', 'b', 'c'}`. Each name must be unique. To assign state names with a variable that has been defined in the MATLAB workspace, enter the variable without quotes. The variable can be a string, cell, or structure.

Settings

Default: `' '` (no name)

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Settings

Default: Off

☑ On

Require that state name resolve to Simulink signal object.

☐ Off

Do not require that state name resolve to Simulink signal object.

Dependencies

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Code generation storage class

Select state storage class.

Settings

Default: `Auto`

`Auto`
: Auto is the appropriate storage class for states that you do not need to interface to external code.

`ExportedGlobal`
: State is stored in a global variable

`ImportedExtern`
: *`model`*`_private.h` declares the state as an extern variable.

`ImportedExternPointer`
: *`model`*`_private.h` declares the state as an extern pointer.

Dependencies

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“State Storage Classes” in the Simulink Coder documentation.

Code generation storage type qualifier

Specify the Simulink Coder storage type qualifier.

Settings

Default: `'  '`

If left blank, no qualifier is assigned.

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Direct Feedthrough	The following ports support direct feedthrough: • Reset port • Integrator and filter initial condition port • Input port, for every integration method except Forward Euler
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Supported for gain parameters **P**, **I**, and **D** for filter coefficient **N**, and for setpoint weights **b** and **c**
States	Inherited from driving block and parameters
Dimensionalized	Yes
Zero-Crossing Detection	Yes (in continuous-time domain)

See Also

PID Controller, Gain, Integrator, Discrete-Time Integrator, Derivative, Discrete Derivative.

Polynomial

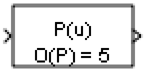
Purpose

Perform evaluation of polynomial coefficients on input values

Library

Math Operations

Description

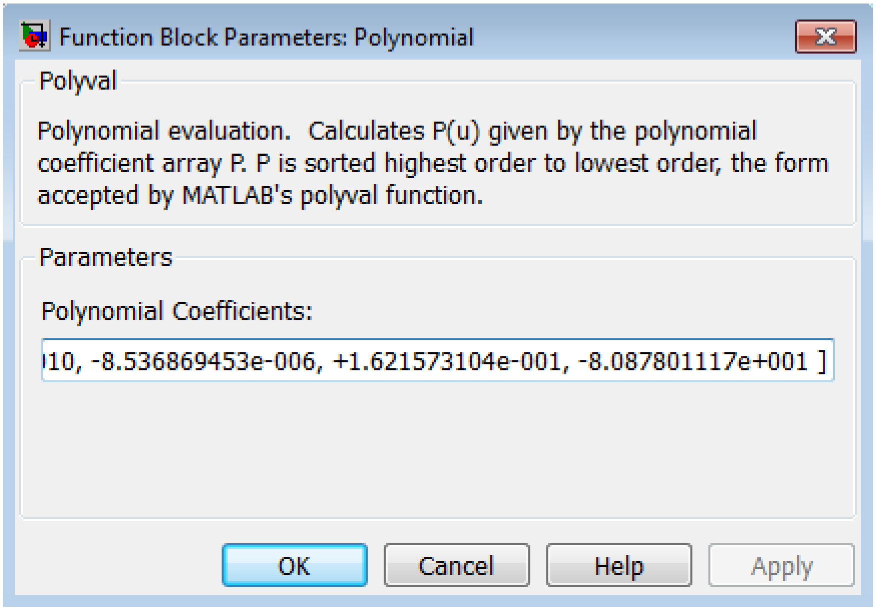


You define a set of polynomial coefficients in the form that the MATLAB `polyval` command accepts. The block evaluates `P(u)` at each time step for the input *u*. The inputs and coefficients must be real.

Data Type Support

The Polynomial block accepts real signals of type `double` or `single`. The **Polynomial coefficients** parameter must be of the same type as the inputs. The output data type is the same as the input data type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Polyval

Polynomial evaluation. Calculates P(u) given by the polynomial coefficient array P. P is sorted highest order to lowest order, the form accepted by MATLAB's polyval function.

Parameters

Polynomial Coefficients:

```
10, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001 ]
```

Parameters and Dialog Box

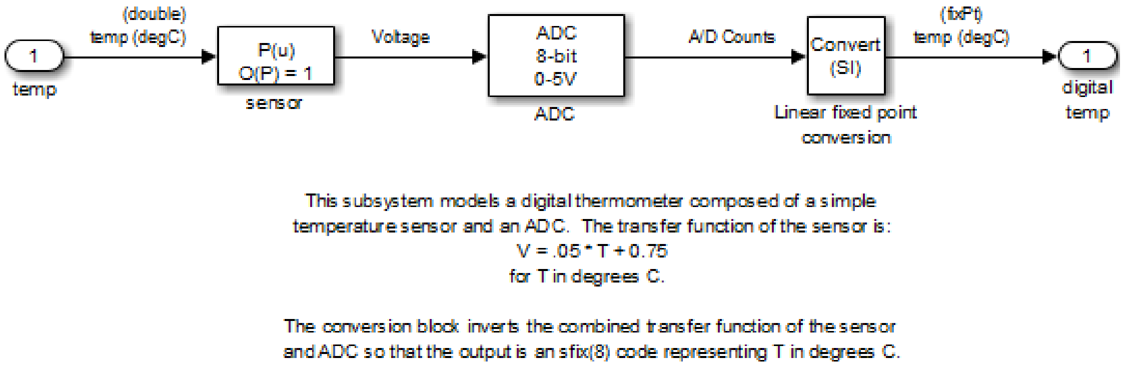
Polynomial coefficients

Specify polynomial coefficients in MATLAB `polyval` form. The first coefficient corresponds to x^N and the remaining coefficients correspond to decreasing orders of x. The last coefficient represents the constant for the polynomial. See `polyval` in the MATLAB documentation for more information.

Examples

The `sldemo_boiler` model shows how to use the Polynomial block.

In the `Boiler Plant model/digital thermometer` subsystem, the Polynomial block models a first-order polynomial using the coefficients `[0.05 0.75]`:


(double)
temp (degC)
1
temp
P(u)
O(P) = 1
sensor
Voltage
ADC
8-bit
0-5V
ADC
A/D Counts
Convert
(SI)
Linear fixed point
conversion
(fixPt)
temp (degC)
1
digital
temp

This subsystem models a digital thermometer composed of a simple temperature sensor and an ADC. The transfer function of the sensor is:
V = .05 * T + 0.75
for T in degrees C.

The conversion block inverts the combined transfer function of the sensor and ADC so that the output is an sfix(8) code representing T in degrees C.

Characteristics

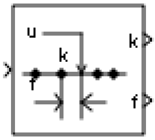
Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	No

Purpose Compute index and fraction for Interpolation Using Prelookup block

Library Lookup Tables

Description

How This Block Works with an Interpolation Using Prelookup Block



The Prelookup block works best with the Interpolation Using Prelookup block. The Prelookup block calculates the index and interval fraction that specify how its input value `u` relates to the breakpoint data set. You feed the resulting index and fraction values into an Interpolation Using Prelookup block to interpolate an *n*-dimensional table. This combination of blocks performs the same operation that a single instance of the n-D Lookup Table block performs. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility that can provide more efficient simulation and code generation. For more information, see "Efficiency of Performance" in the *Simulink User's Guide*.

Supported Block Operations

To use the Prelookup block, you specify a set of breakpoint values directly on the dialog box or feed values into the `bp` input port. Typically, this breakpoint data set corresponds to one dimension of the table data in an Interpolation Using Prelookup block. The Prelookup block generates a pair of outputs for each input value `u` by calculating:

- The index of the breakpoint set element that is less than or equal to `u` and forms an interval containing `u`
- The interval fraction in the range $0 \leq f < 1$, which represents the normalized position of `u` on the breakpoint interval between the index and the next index value for in-range input

For example, if the breakpoint data set is `[ 0 5 10 20 50 100 ]` and the input value `u` is 55, the index is 4 and the fractional value is 0.1. Labels for the index and interval fraction appear as `k` and `f` on the Prelookup block icon. The index value is zero-based.

The interval fraction can be negative or greater than `1` for out-of-range input. See the documentation for the **Extrapolation method** block parameter for more information.

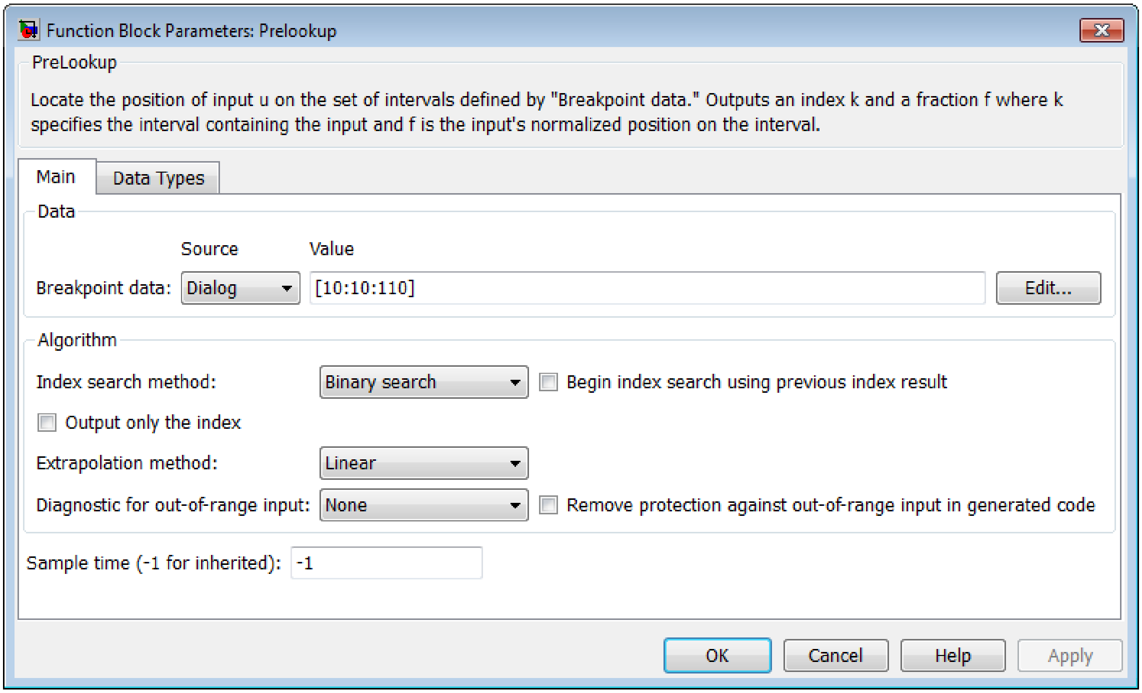
Data Type Support

The Prelookup block accepts real signals of any numeric data type that Simulink supports, except `Boolean`. The Prelookup block supports fixed-point data types for signals and breakpoint data.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Prelookup block dialog box appears as follows:



Function Block Parameters: Prelookup

PreLookup

Locate the position of input u on the set of intervals defined by "Breakpoint data." Outputs an index k and a fraction f where k specifies the interval containing the input and f is the input's normalized position on the interval.

Main | Data Types

Data

Breakpoint data: Source: Dialog; Value: [10:10:110]

Algorithm

Index search method: Binary search

☐ Begin index search using previous index result

☐ Output only the index

Extrapolation method: Linear

Diagnostic for out-of-range input: None

☐ Remove protection against out-of-range input in generated code

Sample time (-1 for inherited): -1

Breakpoint data

Specify whether to enter breakpoint data directly on the dialog box or to inherit the data from an input port.

- If you set **Source** to `Dialog`, enter breakpoint data in the edit field under **Value**. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, you specify breakpoint attributes on the **Data Types** pane.

- If you set **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 from the `bp` input port.

Click the **Edit** button to open the Lookup Table Editor (see "Edit Existing LookupTables" in the Simulink documentation).

Tip To learn how to define evenly spaced breakpoints, see "Formulation of Evenly Spaced Breakpoints" in the Simulink documentation.

Index search method

Select `Evenly spaced points`, `Linear search`, or `Binary search`. Each search method has speed advantages in different situations:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting `Evenly spaced points` to calculate table indices.

  This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input values for `u` do not vary much 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.

Tip The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is `Evenly spaced points`.

Begin index search using previous index result

Select this check box when you want the block to start its search using the index found at the previous time step. For 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.

Output only the index

Select this check box when you want the block to output only the resulting index value, without the interval fraction.

Typical applications 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

Extrapolation method

Specify how to handle out-of-range values for the block input `u`. Options include:

- `Clip`

Block Input	Block Outputs
Less than the first breakpoint	• 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`.

Tip The Prelookup block supports linear extrapolation only when all of these conditions apply:

- The input `u`, breakpoint data, and fraction output use floating-point data types.
- The index uses a built-in integer data type.

Use last breakpoint for input at or above upper limit

Specify how to index input values of `u` that are greater than or equal to the last breakpoint. The index value is zero-based. When input equals the last breakpoint, block outputs differ as follows:

Check Box	Block Outputs
Selected	• Index of the last element in the breakpoint data set • Interval fraction of `0`
Cleared	• Index of the next-to-last breakpoint • Interval fraction of `1`

This check box is visible only when:

- **Output only the index** is cleared.
- **Extrapolation method** is `Clip`.

However, when **Output only the index** is selected and **Extrapolation method** is `Clip`, the block behaves as if this check box is selected even though it is invisible.

Tip When you select **Use last breakpoint for input at or above upper limit** for a Prelookup block, you must also select **Valid index input may reach last index** for the Interpolation Using Prelookup block to which it connects. This action allows the blocks to use the same indexing convention when accessing the last elements of their breakpoint and table data sets.

Diagnostic for out-of-range input

Specify whether to produce a warning or error when the input `u` is out of range. Options include:

- `None` — no warning or error
- `Warning` — display a warning in the MATLAB Command Window and continue the simulation
- `Error` — halt the simulation and display an error in the Simulation Diagnostics Viewer

Remove protection against out-of-range input in generated code

Specify whether or not to include code that checks for out-of-range breakpoint inputs.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range breakpoint inputs.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range breakpoint inputs.	For safety-critical applications

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

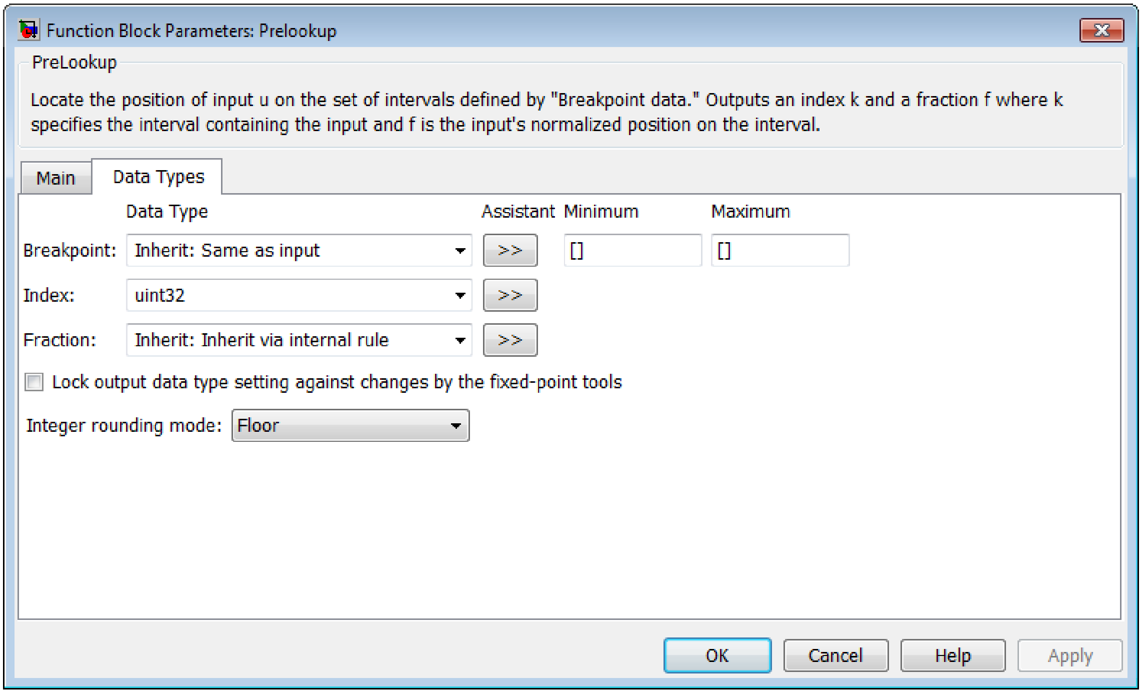
- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Verification and Validation > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Consult the Model Advisor” in the Simulink documentation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the Simulink User’s Guide for more information.

The **Data Types** pane of the Prelookup block dialog box appears as follows:



Function Block Parameters: Prelookup

PreLookup

Locate the position of input u on the set of intervals defined by "Breakpoint data." Outputs an index k and a fraction f where k specifies the interval containing the input and f is the input's normalized position on the interval.

Main
Data Types
Data Type
Assistant
Minimum
Maximum
>>
>>
>>

Breakpoint: Inherit: Same as input; Minimum: []; Maximum: []

Index: uint32

Fraction: Inherit: Inherit via internal rule

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

OK
Cancel
Help
Apply

Note The parameters for breakpoint attributes (data type, minimum, and maximum) are not available when you set **Source** to `Input port`. In this case, the block inherits all breakpoint attributes from the `bp` input port.

Breakpoint > Data Type

Specify the breakpoint data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the breakpoint data type.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Tip Specify a breakpoint data type different from the data type of input `u` for these cases:

- Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal `u`
- Sharing of prescaled breakpoint data between two Prelookup blocks with different data types for input `u`
- Sharing of custom storage breakpoint data in the generated code for blocks with different data types for input `u`

Breakpoint > Minimum

Specify the minimum value that the breakpoint data can have. The default value is `[]` (unspecified).

Breakpoint > Maximum

Specify the maximum value that the breakpoint data can have. The default value is `[]` (unspecified).

Index > Data Type

Specify a data type that can index all elements in the breakpoint data set. You can:

- Select a built-in integer data type from the list.
- Specify an integer data type using a fixed-point representation.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the index data type.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Fraction > Data Type

Specify the data type of the interval fraction. You can:

- Select a built-in data type from the list.
- Specify data type inheritance through an internal rule.
- Specify a fixed-point data type using the [Slope Bias] or binary-point-only scaling representation.
  - If you use the [Slope Bias] representation, the scaling must be trivial — that is, the slope is `1` and the bias is `0`.
  - If you use the binary-point-only representation, the fixed power-of-two exponent must be less than or equal to zero.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the fraction data type.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting”.

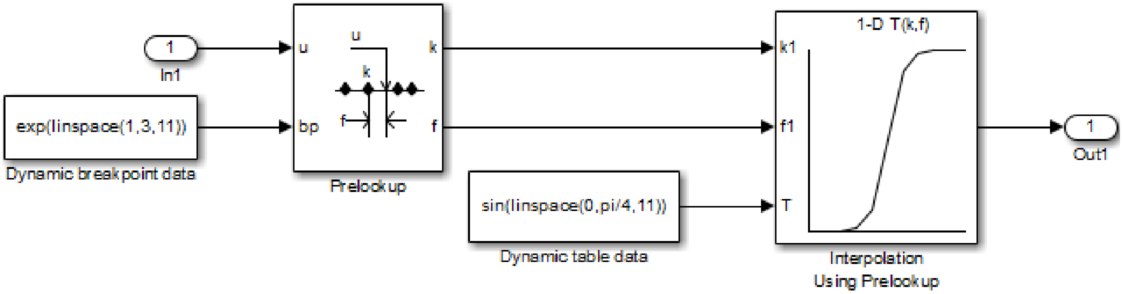
Integer rounding mode

re youSpecify the rounding mode for fixed-point operations. For more information, see “Rounding” in the Simulink Fixed Point documentation.

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Examples

In the following model, a Constant block feeds the breakpoint data set to the `bp` input port of the Prelookup block.



1
In1
exp(linspace(1,3,11))
Dynamic breakpoint data
u
bp
k
f
Prelookup
sin(linspace(0,pi/4,11))
Dynamic table data
1-D T(k,f)
k1
f1
T
Interpolation
Using Prelookup
1
Out1

The Prelookup block inherits the following breakpoint attributes from the `bp` input port:

Breakpoint Attribute	**Value**
Minimum	`–Inf`
Maximum	`Inf`
Data type	`single`

Similarly, a Constant block feeds the table data values to the `T` input port of the Interpolation Using Prelookup block, which inherits the following table attributes:

Table Attribute	**Value**
Minimum	`–Inf`
Maximum	`Inf`
Data type	`single`

Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision, floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.

For other examples, see "Prelookup and Interpolation Blocks" in the Simulink documentation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also

Interpolation Using Prelookup

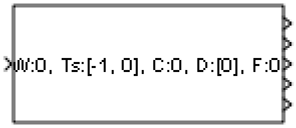
Purpose

Output signal attributes, including width, dimensionality, sample time, and complex signal flag

Library

Signal Attributes

Description



The Probe block outputs selected information about the signal on its input. The block can output the input signal's width, dimensionality, sample time, and a flag indicating whether the input is a complex-valued signal. The block has one input port. The number of output ports depends on the information that you select for probing, that is, signal dimensionality, sample time, and/or complex signal flag. Each probed value is output as a separate signal on a separate output port. The block accepts real or complex-valued signals of any built-in data type. It outputs signals of type `double`. During simulation, the block icon displays the probed data.

Data Type Support

The Probe block accepts signals of the following data types:

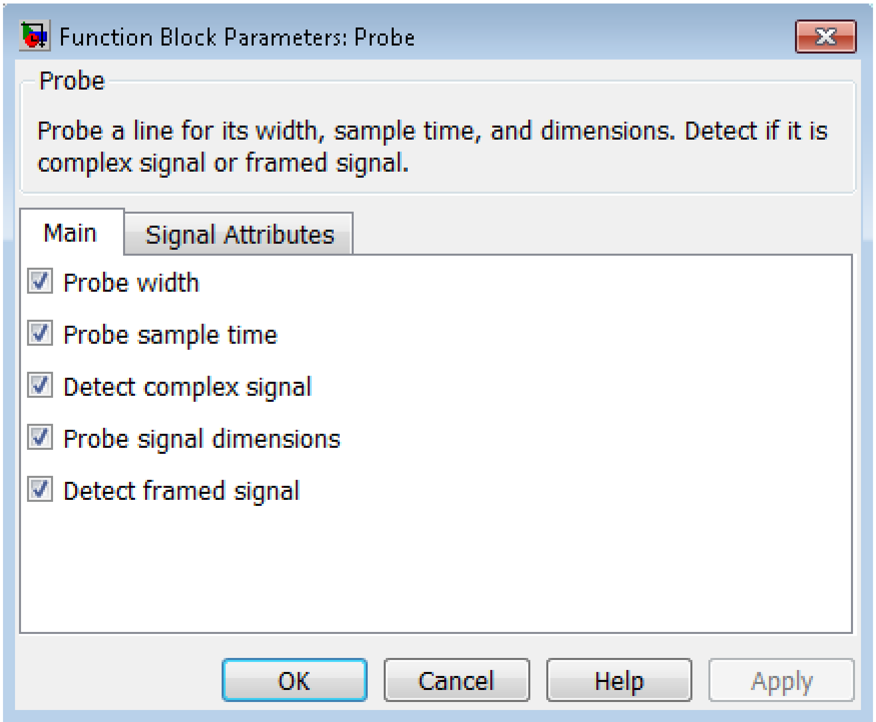
- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated (input only)
- Bus object

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

You can use an array of buses as an input signal to a Probe block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

Parameters and Dialog Box

The **Main** pane of the Probe block dialog box appears as follows:



Function Block Parameters: Probe

Probe

Probe a line for its width, sample time, and dimensions. Detect if it is complex signal or framed signal.

☑ Probe width

☑ Probe sample time

☑ Detect complex signal

☑ Probe signal dimensions

☑ Detect framed signal

Probe width

Select to output the width, or number of elements, of the probed signal.

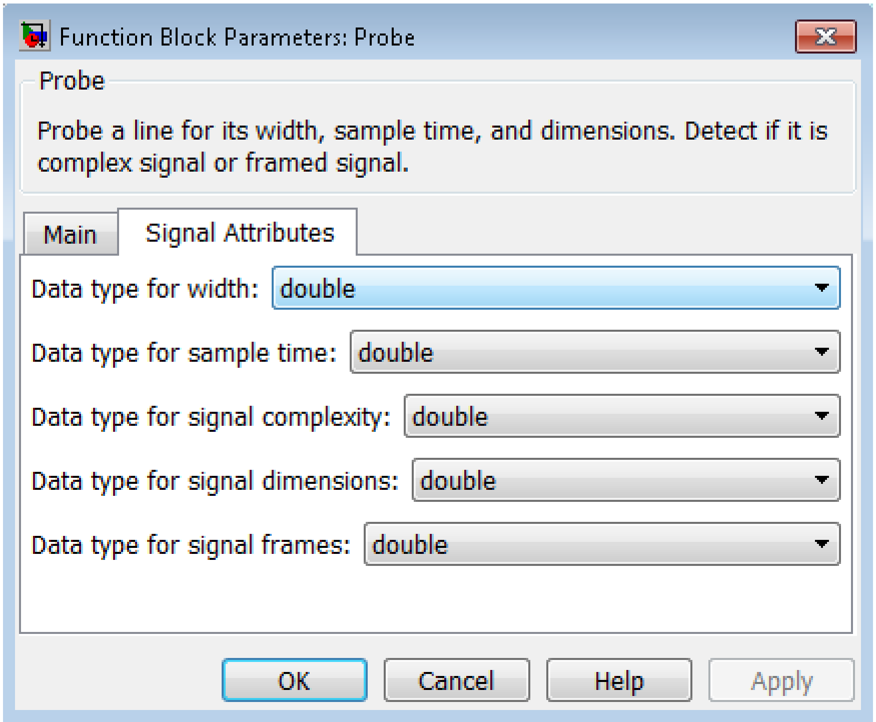
Probe sample time
Select to output the sample time of the probed signal. The output is a two-element vector that specifies the period and offset of the sample time, respectively. See “Specify Sample Time” for more information.

Detect complex signal
Select to output 1 if the probed signal is complex; otherwise, 0.

Probe signal dimensions
Select to output the dimensions of the probed signal.

Detect framed signal
Select to output 1 if the probed signal is framed; otherwise, 0.

The **Signal Attributes** pane of the Probe block dialog box appears as follows:



Function Block Parameters: Probe

Probe

Probe a line for its width, sample time, and dimensions. Detect if it is complex signal or framed signal.

Main
Signal Attributes

Data type for width: double

Data type for sample time: double

Data type for signal complexity: double

Data type for signal dimensions: double

Data type for signal frames: double

OK
Cancel
Help
Apply

Note The Probe block ignores the **Data type override** setting of the Fixed-Point Tool.

Data type for width
Select the output data type for the width information.

Data type for sample time
Select the output data type for the sample time information.

Data type for signal complexity
Select the output data type for the complexity information.

Data type for signal dimensions
Select the output data type for the dimensions information.

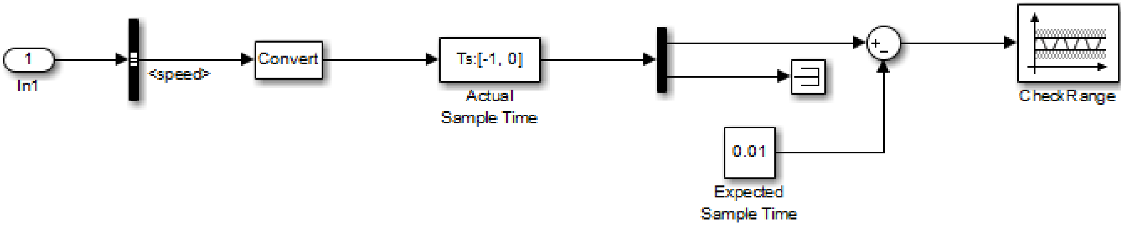
Data type for signal frames
Select the output data type for the frames information.

Note For **Data type for width**, **Data type for sample time**, and **Data type for signal dimensions**, the `Boolean` data type is not supported. Furthermore, if you select `Same as input` in any of these drop-down lists, and the block's input signal data type is `Boolean`, when you simulate your model, you see an error.

Examples

The `sldemo_fuelsys` model shows how you can use the Probe block.

In the `fuel_rate_control/validate_sample_time` subsystem, the Probe block determines the sample time of the input signal to verify that it matches the assumed value of the design:



Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Dimensionalized	Yes

Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

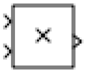
Multiply and divide scalars and nonscalars or multiply and invert matrices

Library

Math Operations

Description

Default Product Block Use



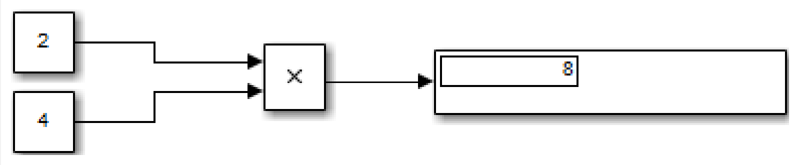
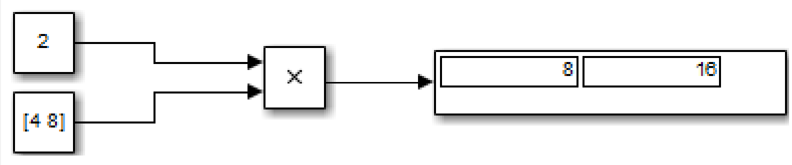
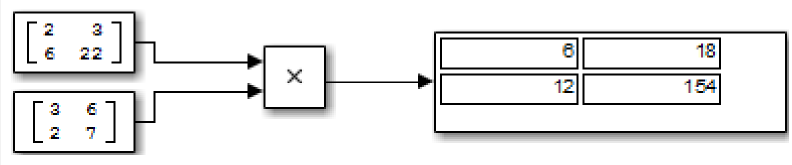
By default, the Product block outputs the result of multiplying two inputs: two scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The default parameter values that specify this behavior are:

- **Multiplication:** `Element-wise(.*)`
- **Number of inputs:** `2`

Setting non-default values for either of these parameters can change a Product block to function as a Divide block or a Product of Elements block. See the documentation of those two blocks for more information.

If all you need is to multiply two inputs to create an output, you can use the Product block with default parameter values. If you need additional capabilities, see "Product Block Capabilities" on page 1-1304.

The following 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.	
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.	

Product Block Capabilities

The Product block, the Divide block, and the Product of Elements block are actually the same underlying block with different default values for the **Number of inputs** parameter. All three blocks can therefore provide the same capabilities. The Product block (or the Divide block or Product of Elements block if appropriately configured) can:

- Numerically multiply and divide any number of scalar, vector, or matrix inputs

- Perform matrix multiplication and division on any number of matrix inputs

The Product block performs scalar or matrix multiplication, depending on the value of the **Multiplication** parameter. The block accepts one or more inputs, depending on the **Number of inputs** parameter. The **Number of inputs** parameter also specifies the operation to perform on each input.

The Product block can input any combination of scalars, vectors, and matrices for which the operation to perform has a mathematically defined result. The block performs the specified operations on the inputs, then outputs the result.

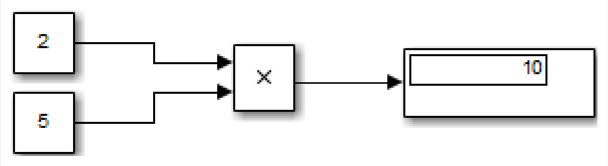
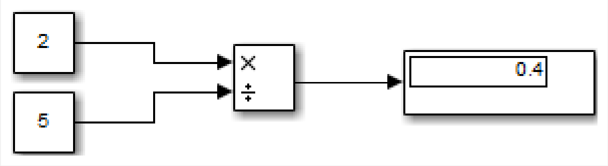
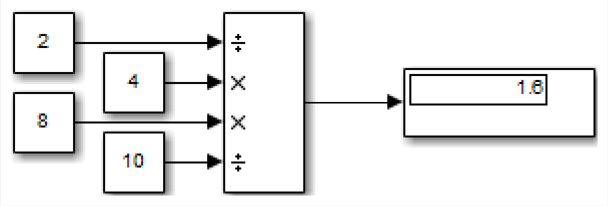
The Product block has two modes: *Element-wise mode*, which processes nonscalar inputs element by element, and *Matrix mode*, which processes nonscalar inputs as matrices. The next two sections describe these two modes.

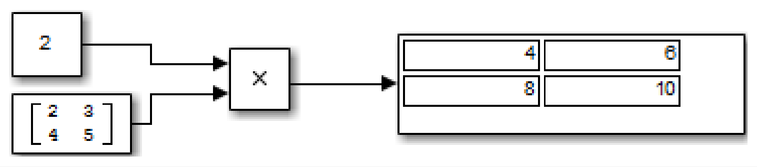
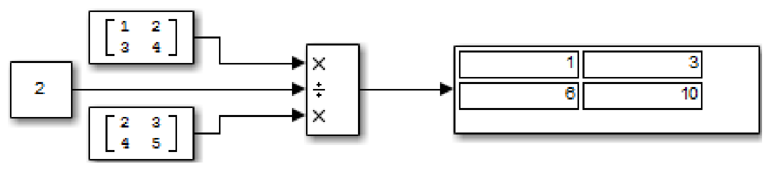
Element-wise Mode

When the value of the **Multiplication** parameter is `Element-wise(.*)`, the Product block is in *Element-wise mode*, in which it operates on the individual numeric elements of any nonscalar inputs. The MATLAB equivalent is the `.*` operator. In element-wise mode, the Product block can perform a variety of multiplication, division, and arithmetic inversion operations.

The value of the **Number of inputs** parameter controls both how many inputs exist and whether each is multiplied or divided to form the output. When the Product block in Element-wise mode has only one input, it is functionally equivalent to a Product of Elements block. When the block has multiple inputs, any nonscalar inputs must have identical dimensions, and the block outputs a nonscalar with those dimensions. To calculate the output, the block first expands any scalar input to a nonscalar that has the same dimensions as the nonscalar inputs.

This table shows the output of the Product block for example inputs, using the indicated values for the **Number of inputs** parameter.

Parameter Values	Examples
Number of inputs: 2	
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 might include extreme values, please manage these cases explicitly.

The generated code might not produce the exact same pattern of `NaN` and `inf` values as simulation when these values are mathematically meaningless. For example, if the simulation output contains a `NaN`, output from the generated code also contains a `NaN`, but not necessarily in the same place.

Data Type Support

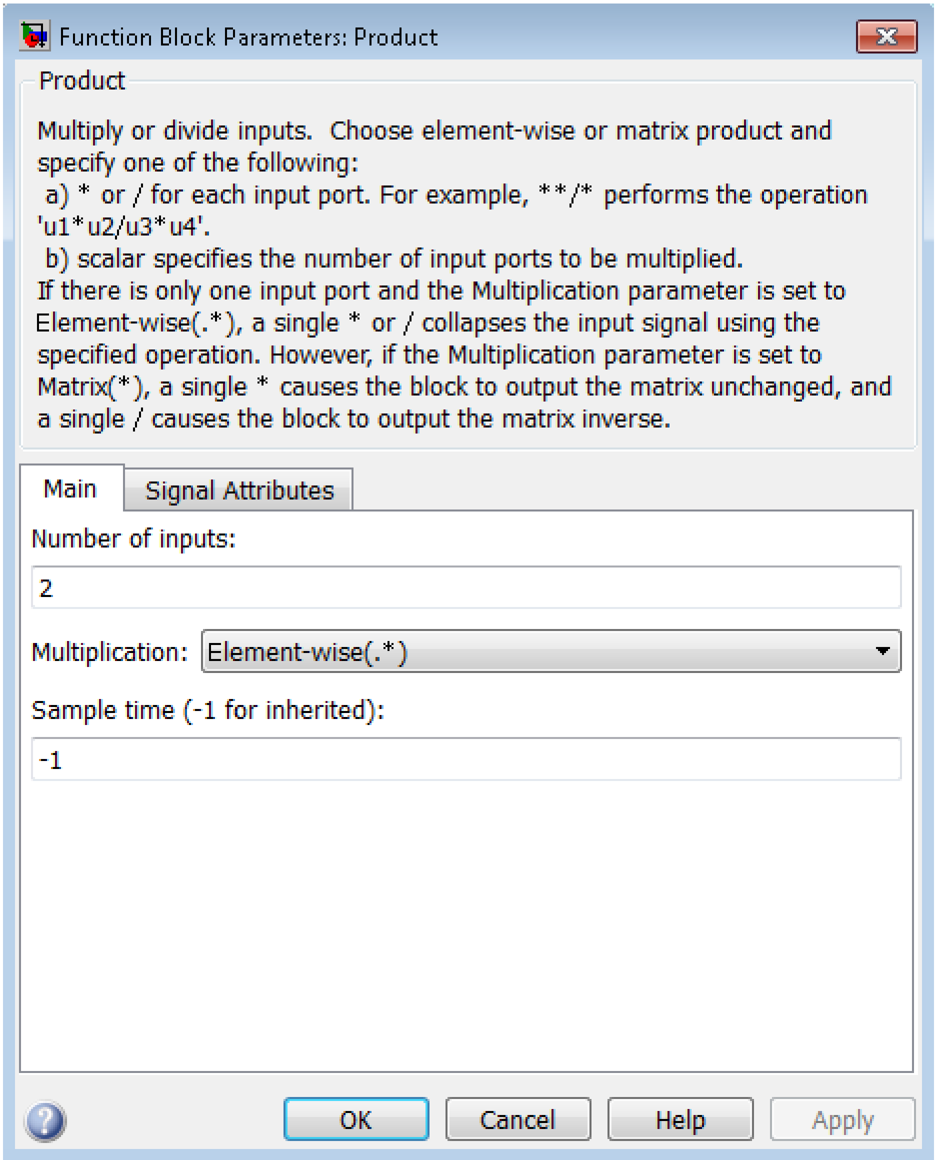
The Product block accepts real or complex signals of any numeric data type that Simulink supports, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

The Product block does not support numeric division for complex signals with `boolean` or fixed-point data types. For other types, the block accepts complex signals as divisors only when the input and output signals all specify the same built-in data type. In this case, however, the block ignores its specified rounding mode.

The Product block accepts multidimensional signals when operating in Element-wise mode, but not when operating in Matrix mode. See “Signal Dimensions”, “Element-wise Mode” on page 1-1305, and “Matrix Mode” on page 1-1307 for more information.

Parameters and Dialog Box

The **Main** pane of the Product block dialog box appears as follows:



Product

Multiply or divide inputs. Choose element-wise or matrix product and specify one of the following:
a) * or / for each input port. For example, **/* performs the operation 'u1*u2/u3*u4'.
b) scalar specifies the number of input ports to be multiplied.
If there is only one input port and the Multiplication parameter is set to Element-wise(.*), a single * or / collapses the input signal using the specified operation. However, if the Multiplication parameter is set to Matrix(*), a single * causes the block to output the matrix unchanged, and a single / causes the block to output the matrix inverse.

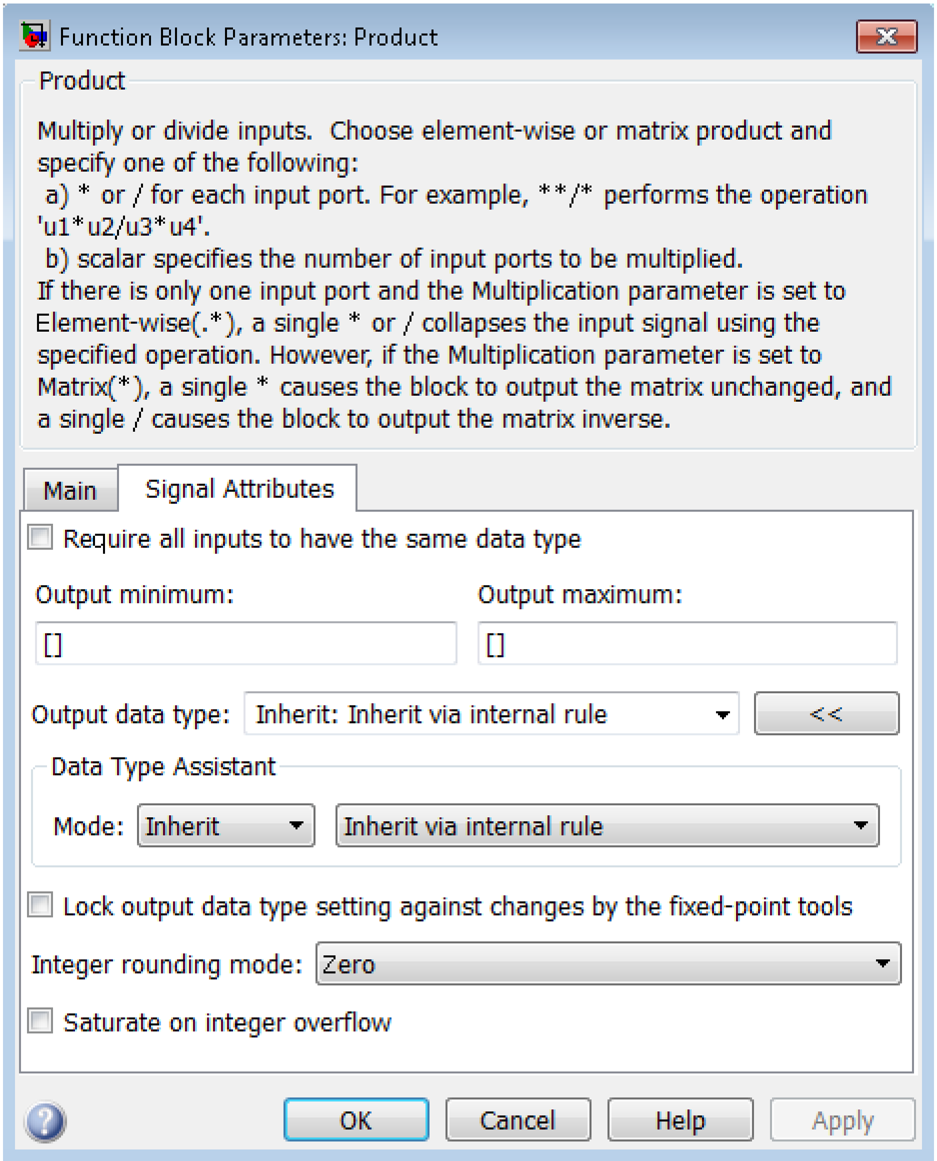
Number of inputs: 2

Multiplication: Element-wise(.*)

Sample time (-1 for inherited): -1

Signal Attributes Pane

The **Signal Attributes** pane of the Product block dialog box appears as follows:



Product

Multiply or divide inputs. Choose element-wise or matrix product and specify one of the following:

a) * or / for each input port. For example, **/* performs the operation 'u1*u2/u3*u4'.

b) scalar specifies the number of input ports to be multiplied.

If there is only one input port and the Multiplication parameter is set to Element-wise(.*), a single * or / collapses the input signal using the specified operation. However, if the Multiplication parameter is set to Matrix(*), a single * causes the block to output the matrix unchanged, and a single / causes the block to output the matrix inverse.

☐ Require all inputs to have the same data type

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

Data Type Assistant

Mode: Inherit | Inherit via internal rule

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Zero

☐ Saturate on integer overflow

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Number of inputs

Control two properties of the Product block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

Settings

Default:`2`

- **`1` or `*` or `/`**

  Has one input. In Element-wise mode, the input is processed 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 1-1305 and "Matrix Mode" on page 1-1307 for more information.

- **An integer value > 1**

  Has 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 1-1305 and "Matrix Mode" on page 1-1307 for more information.

- **An unquoted string of two or more `*` and `/` characters**

  Has the number of inputs given by the length of the string. 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 1-1305 and "Matrix Mode" on page 1-1307 for more information.

Dependency

Setting **Number of inputs** to `*` and selecting `Element-wise(.*)` for **Multiplication** enable the following parameter:

- **Multiply over**

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Multiplication

Specify whether the Product block operates in Element-wise mode or Matrix mode.

Settings

Default: `Element-wise(.*)`

`Element-wise(.*)`
Operate in Element-wise mode.

`Matrix(*)`
Operate in Matrix mode.

Dependency

Selecting `Element-wise(.*)` and setting **Number of inputs** to `*` enable the following parameter:

- **Multiply over**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Multiply over

Affect multiplication on matrix input.

Settings

Default: `All dimensions`

`All dimensions`

Output a scalar that is product of all elements of the matrix, or the product of their inverses, depending on the value of **Number of inputs**.

`Specified dimension`

Output a vector, the composition of which depends on the value of the **Dimension** parameter.

Dependencies

- Enable this parameter by selecting `Element-wise(.*)` for **Multiplication** and setting **Number of inputs** to `*` or `1` or `/`.
- Setting this parameter to `Specified dimension` enables the **Dimension** parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Dimension

Affect multiplication on matrix input.

Settings

Default: `1`

Minimum: `1`

Maximum: `2`

`1`

Output a vector that contains an element for each column of the input matrix.

`2`

Output a vector that contains an element for each row of the input matrix.

Tips

Each element of the output vector contains the product of all elements in the corresponding column or row of the input matrix, or the product of the inverses of those elements, depending on the value of **Number of inputs**:

- `1` or `*`

  Multiply the values of the column or row elements
- `/`

  Multiply the inverses of the column or row elements

Dependency

Enable this parameter by selecting `Specified dimension` for **Multiply over**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Settings

Default: Off

☑ On
Require that all inputs have the same data type.

☐ Off
Do not require that all inputs have the same data type.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Zero`

`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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding”.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the minimum to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Inherit via internal rule`

`Inherit:  Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit:  Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit:  Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`Inherit:  Same as first input`

Use data type of the first input signal.

`double`

Output data type is `double`.

`single`

Output data type is `single`.

`int8`
: Output data type is `int8`.

`uint8`
: Output data type is `uint8`.

`int16`
: Output data type is `int16`.

`uint16`
: Output data type is `uint16`.

`int32`
: Output data type is `int32`.

`uint32`
: Output data type is `uint32`.

`fixdt(1,16,0)`
: Output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
: Output data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
: Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specify Block Output Data Types".

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule` (default)
- `Inherit via back propagation`
- `Same as first input`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant".

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`

`Binary point`
    Specify binary point location.

`Slope and bias`
    Enter slope and bias.

`Best precision`
    Specify best-precision values.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes

Dimensionalized	Yes
Multidimensionalized	Yes, only when the **Multiplication** parameter specifies `Element-wise(.*)`
Zero-Crossing Detection	No

See Also

- Divide
- Dot Product
- Product of Elements

Product of Elements

Purpose

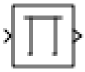
Copy or invert one scalar input, or collapse one nonscalar input

Library

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 by multiplying together the elements of each row or column or taking successive inverses of the elements of each row or column

The Product of Elements block is functionally a Product block that has two preset parameter values:

- **Multiplication**: `Element-wise(.*)`
- **Number of inputs**: `*`

Setting non-default values for either of those parameters can change a Product of Elements block to be functionally equivalent to a Product block or a Divide block. See the documentation of those two blocks for more information.

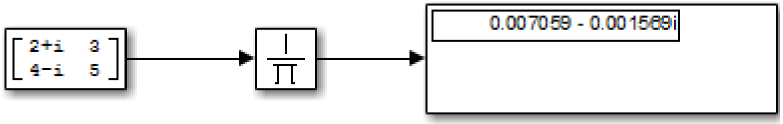
Algorithm

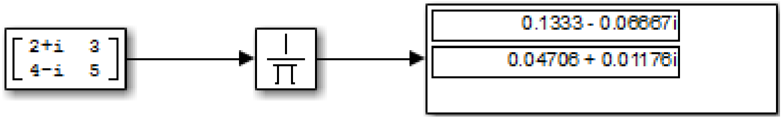
The Product of Elements block uses the following algorithms to perform element-wise operations on inputs of floating-point, built-in integer, and fixed-point types:

Input	Element-Wise Operation	Algorithm
Real scalar, `u`	Multiplication	`y = u`
	Division	`y = 1/u`

Input	**Element-Wise Operation**	**Algorithm**
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. For example, consider the following model:


2+i 3
4-i 5
$\frac{1}{\prod}$
0.007059 - 0.001569i


2+i 3
4-i 5
$\frac{1}{\prod}$
0.1333 - 0.06667i
0.04706 + 0.01176i

The top Product of Elements block collapses the matrix input to a scalar by taking successive inverses of the four elements:

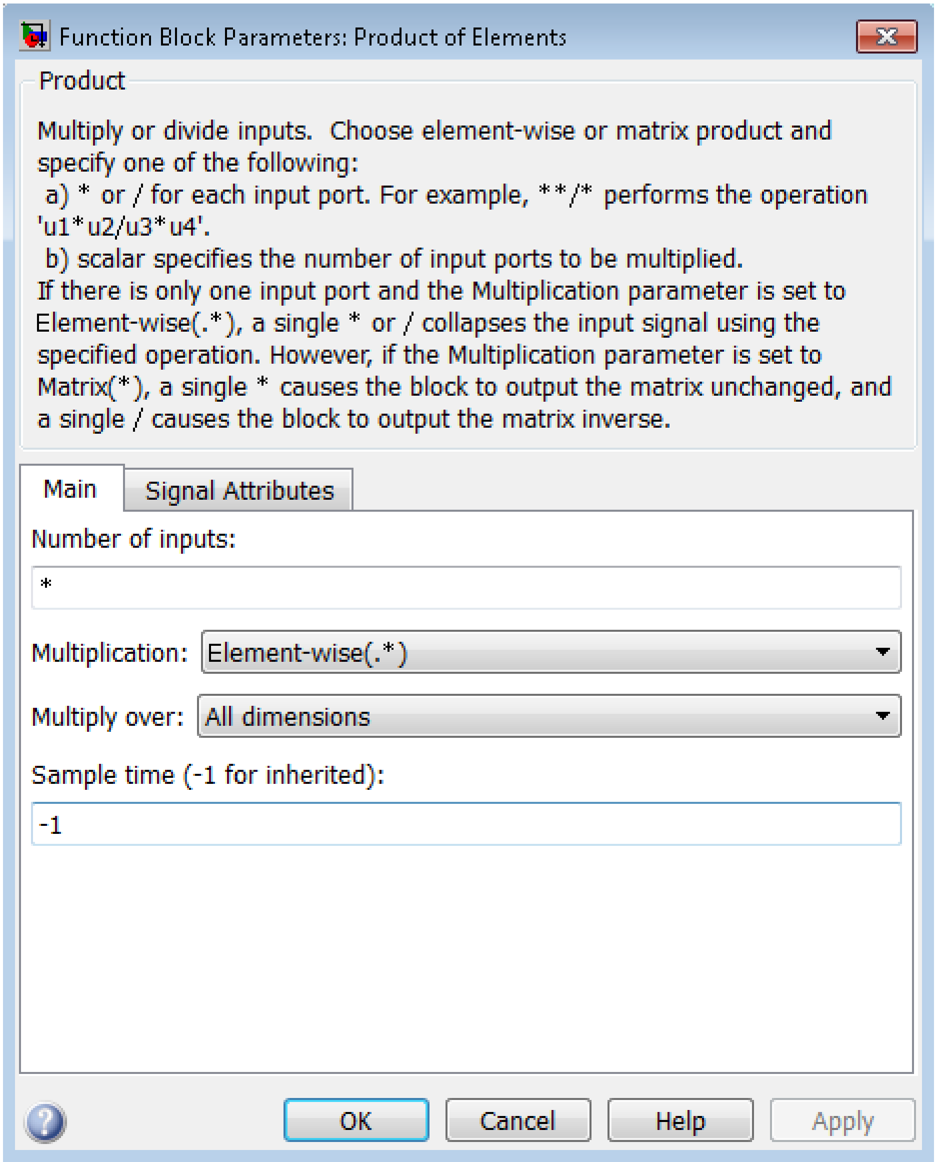
- `y = ((((1/2+i)/3)/4-i)/5)`

The bottom Product of Elements block collapses the matrix input to a vector by taking successive inverses along the second dimension:

- `y(1) = ((1/2+i)/3)`
- `y(2) = ((1/4-i)/5)`

Parameters and Dialog Box

The Product of Elements block has the same parameters and dialog box as the Product block, plus the parameter **Multiply over**, which has the default value `All dimensions`:



Product

Multiply or divide inputs. Choose element-wise or matrix product and specify one of the following:
a) * or / for each input port. For example, **/* performs the operation 'u1*u2/u3*u4'.
b) scalar specifies the number of input ports to be multiplied.
If there is only one input port and the Multiplication parameter is set to Element-wise(.*), a single * or / collapses the input signal using the specified operation. However, if the Multiplication parameter is set to Matrix(*), a single * causes the block to output the matrix unchanged, and a single / causes the block to output the matrix inverse.

Number of inputs: *

Multiplication: Element-wise(.*)

Multiply over: All dimensions

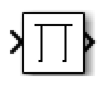
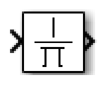
Sample time (-1 for inherited): -1

If you set **Multiply over** to `Specified dimension`, the **Dimension** parameter appears.

To copy, invert, or collapse one input to create an output, you can use the Product of Elements block with default values for all parameters except **Number of inputs**, **Multiply over**, and **Dimension**. These values can require change. For other capabilities, see the Product block documentation, which also describes the “Signal Attributes Pane” on page 1-1310 of the Product of Elements block.

Number of inputs

This parameter is the same as in the Product block, but the value must be `*` (the default), `1`, or `/` to achieve the behavior of a Product of Elements block.

Parameter Value	Block Behavior	Block Icon
`*` or `1`	• Copies a scalar input unchanged • Collapses a vector input to a scalar by multiplying all elements together • Collapses a matrix input to a scalar or vector by multiplying elements together based on the **Multiply over** parameter For more information, see “Algorithm” on page 1-1338.	
`/`	• Outputs the arithmetic inverse of a scalar input • Collapses a vector input to a scalar by taking successive inverses of the elements • Collapses a matrix input to a scalar or vector by taking successive inverses of elements based on the **Multiply over** parameter For more information, see “Algorithm” on page 1-1338.	

Multiply over

This parameter appears only when **Multiplication** is `Element-wise(.*)` and **Number of inputs** is `*`, `1`, or `/`. The parameter affects only a matrix input. The possible values are:

- `All dimensions` — The block outputs a scalar that is the product of all matrix elements, or the result of taking successive inverses of the elements, depending on the value of **Number of inputs**.
- `Specified dimension` — The block outputs a vector, in which the composition depends on the value of the **Dimension** parameter.

Dimension

This parameter appears only when the **Multiply over** parameter appears and is set to `Specified dimension`. The parameter affects only a matrix input, and must be 1 for a scalar or vector input. The possible values are:

- `1` — Output a vector that contains an element for each column of the input matrix.
- `2` — Output a vector that contains an element for each row of the input matrix.

Each element of the output vector contains the product of all elements in the corresponding column or row of the input matrix, or the result of taking successive inverses of those elements. The output vector depends on the value of **Number of inputs**:

- **`*` or `1`**

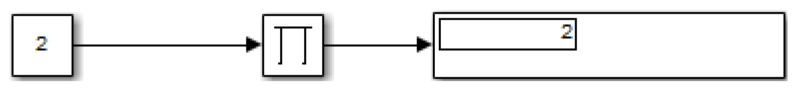
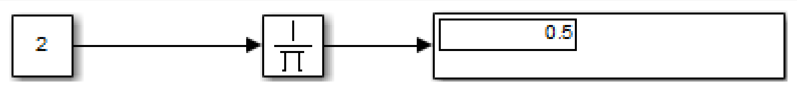
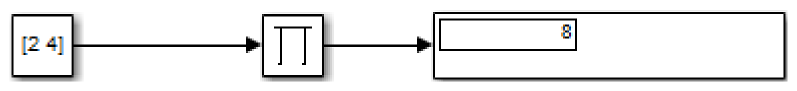
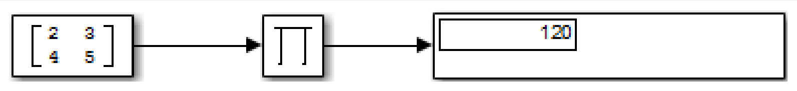
  Multiply the values of the column or row elements.

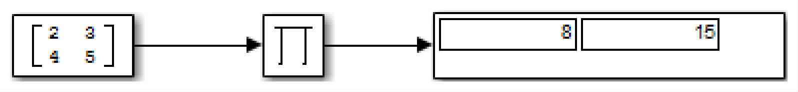
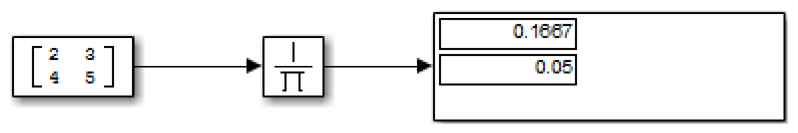
- `/`

  Take successive inverses of the column or row elements. For more information, see "Algorithm" on page 1-1338.

Examples

This table shows the output of the Product of Elements block for example inputs using default block parameter values, except as shown in the table.

Parameter Values	Examples
Multiplication: `Element-wise(.*)` **Number of inputs**: `*`	
Multiplication: `Element-wise(.*)` **Number of inputs**: `/`	
Multiplication: `Element-wise(.*)` **Number of inputs**: `*`	
Multiplication: `Element-wise(.*)` **Number of inputs**: `*` **Multiply over**: `All dimensions`	

Parameter Values	Examples
Multiplication: `Element-wise(.*)` **Number of inputs**: `*` **Multiply over**: `Specified dimension` **Dimension**: `1`	
Multiplication: `Element-wise(.*)` **Number of inputs**: `/` **Multiply over**: `Specified dimension` **Dimension**: `2`	

Pulse Generator

Purpose

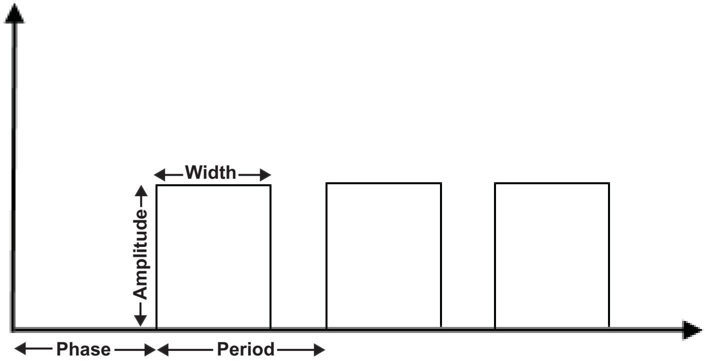
Generate square wave pulses at regular intervals

Library

Sources

Description

The Pulse Generator block generates square wave pulses at regular intervals. The block's waveform parameters, **Amplitude**, **Pulse Width**, **Period**, and **Phase delay**, determine the shape of the output waveform. The following diagram shows how each parameter affects the waveform.



The Pulse Generator can emit scalar, vector, or matrix signals of any real data type. To cause the block to emit a scalar signal, use scalars to specify the waveform parameters. To cause the block to emit a vector or matrix signal, use vectors or matrices, respectively, to specify the waveform parameters. Each element of the waveform parameters affects the corresponding element of the output signal. For example, the first element of a vector amplitude parameter determines the amplitude of the first element of a vector output pulse. All the waveform parameters must have the same dimensions after scalar expansion. The data type of the output is the same as the data type of the **Amplitude** parameter.

Use the **Pulse type** parameter to specify whether the block's output is time-based or sample-based. If you select `sample-based`, the block computes its outputs at fixed intervals that you specify. If you select `time-based`, Simulink software computes the block's outputs only at times when the output actually changes. This choice can result in fewer computations for computing the block's output over the simulation time period.

Depending on the pulse's waveform characteristics, the intervals between changes in the block's output can vary. For this reason, a time-based Pulse Generator block has a variable sample time. Simulink software uses brown as the sample time color of such blocks (see "View Sample Time Information" for more information).

Simulink software 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 software computes a fixed sample time for the time-based pulse generators. Then the time-based pulse generators simulate as sample-based.

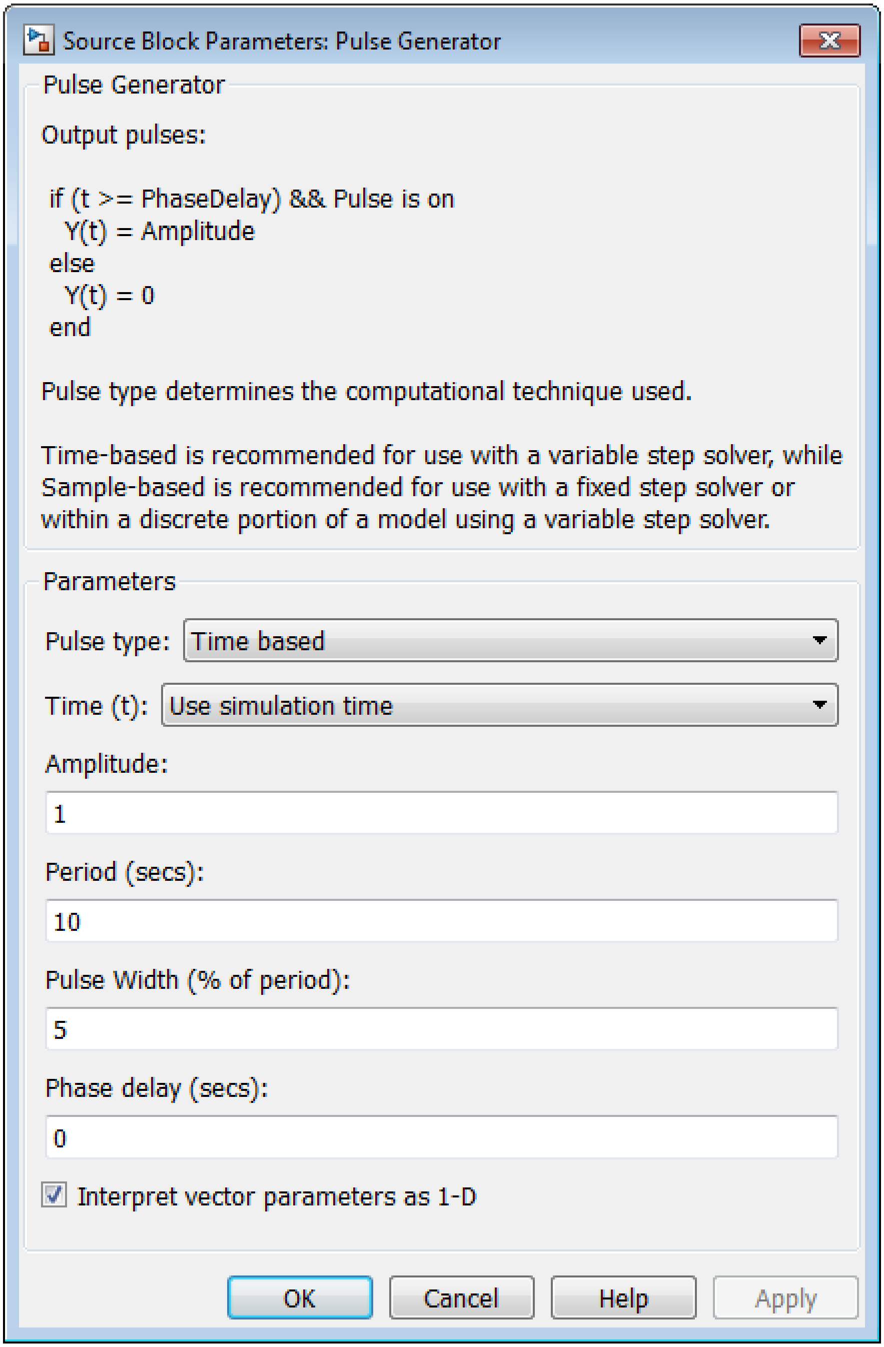
Tip If you use a fixed-step solver and the **Pulse type** is `time-based`, you must choose the step size such that the period, phase delay, and pulse width (in seconds) are integer multiples of the step size. For example, suppose that the period is `4` seconds, the pulse width is `75%` (i.e., `3` s), and the phase delay is `1` s. In this case, the computed sample time is `1` s. Therefore, you must choose a fixed-step size that is `1` or that divides `1` exactly (e.g., `0.25`). You can guarantee this by setting the fixed-step solver's step size to `auto` on the **Solver** pane of the Configuration Parameters dialog box.

If you select time-based as the block's pulse type, you must specify the pulse's phase delay and period in units of seconds. If you specify sample-based, you must specify the block's sample time in seconds, using the **Sample time** parameter, then specify the block's phase delay and period as integer multiples of the sample time. For example, suppose that you specify a sample time of 0.5 second and want the pulse to repeat every two seconds. In this case, you would specify 4 as the value of the block's **Period** parameter.

Data Type Support

The Pulse Generator block outputs real signals of any numeric data type that Simulink supports, including fixed-point data types. The data type of the output signal is the same as that of the **Amplitude** parameter.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Source Block Parameters: Pulse Generator

Pulse Generator

Output pulses:

```
if (t >= PhaseDelay) && Pulse is on
  Y(t) = Amplitude
else
  Y(t) = 0
end
```

Pulse type determines the computational technique used.

Time-based is recommended for use with a variable step solver, while Sample-based is recommended for use with a fixed step solver or within a discrete portion of a model using a variable step solver.

Parameters

Pulse type: Time based

Time (t): Use simulation time

Amplitude: 1

Period (secs): 10

Pulse Width (% of period): 5

Phase delay (secs): 0

☑ Interpret vector parameters as 1-D

Parameters and Dialog Box

Pulse type

The pulse type for this block: time-based or sample-based. The default is time-based.

- If you select `Use simulation time`, the block generates an output pulse where the time variable equals the simulation time.
- If you select `Use external signal`, the block generates an output pulse where the time variable equals the value from the input port, which can differ from the simulation time.

Amplitude

The pulse amplitude. The default is `1`.

Period

The pulse period specified in seconds if the pulse type is time-based or as number of sample times if the pulse type is sample-based. The default is `10` seconds.

Pulse Width

The duty cycle specified as the percentage of the pulse period that the signal is on if time-based or as number of sample times if sample-based. The default is `5` percent.

Phase delay

The delay before the pulse is generated specified in seconds if the pulse type is time-based or as number of sample times if the pulse type is sample-based. The default is `0` seconds.

Sample time

The length of the sample time for this block in seconds. This parameter appears only if the block's pulse type is sample-based. See "Specify Sample Time" in the Simulink User's Guide for more information.

Interpret vector parameters as 1-D

If you select this check box and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise the output dimensionality is the same as that of the other parameters. See "Determining the Output Dimensions of Source Blocks" in the "Working with Signals" chapter of the Simulink User's Guide.

Examples

The following Simulink examples show how to use the Pulse Generator block:

- `sldemo_auto_climatecontrol`
- `sldemo_boiler`

Characteristics

Sample Time	Inherited
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes
Zero-Crossing Detection	No

Quantizer

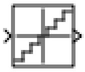
Purpose

Discretize input at specified interval

Library

Discontinuities

Description



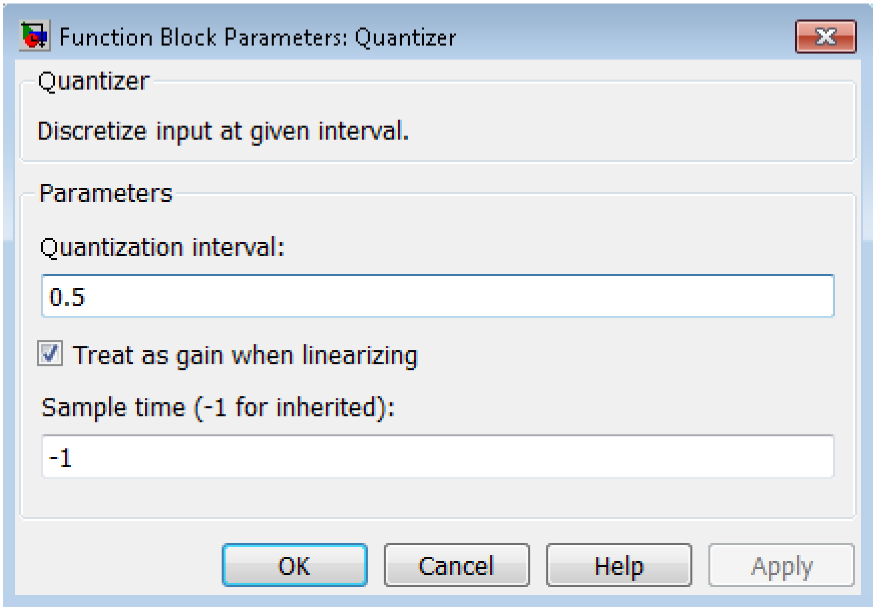
The Quantizer block passes its input signal through a stair-step function so that many neighboring points on the input axis are mapped to one point on the output axis. The effect is to quantize a smooth signal into a stair-step output. The output is computed using the round-to-nearest method, which produces an output that is symmetric about zero.

```
y = q * round(u/q)
```

where `y` is the output, `u` the input, and `q` the **Quantization interval** parameter.

Data Type Support

The Quantizer block accepts and outputs real or complex signals of type `single` or `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Quantizer

Quantizer

Discretize input at given interval.

Parameters

Quantization interval:

0.5

☑ Treat as gain when linearizing

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Quantization interval

The interval around which the output is quantized. Permissible output values for the Quantizer block are `n*q`, where `n` is an integer and `q` the **Quantization interval**. The default is `0.5`.

Treat as gain when linearizing

Simulink software by default treats the Quantizer block as unity gain when linearizing. This setting corresponds to the large-signal linearization case. If you clear this check box, the linearization routines assume the small-signal case and set the gain to zero.

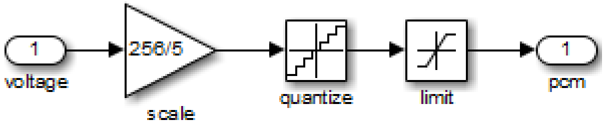
Sample time (-1 for inherited)

Specify the sample time of this Outport block. See “Specify Sample Time” in the online documentation for information on specifying sample times. The output of this block changes at the specified rate to reflect the value of its input.

Examples

The `sldemo_boiler` model shows how you can use the Quantizer block.

The Quantizer block appears in the `Boiler Plant model/digital thermometer/ADC` subsystem.



The ADC subsystem digitizes the input analog voltage by:

- Multiplying the analog voltage by 256/5 with the Gain block
- Rounding the value to integer floor with the Quantizer block
- Limiting the output to a maximum of 255 (the largest unsigned 8-bit integer value) with the Saturation block

For more information, see “Explore the Fixed-Point "Bang-Bang Control" Model” in the Stateflow documentation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes, of parameter
Dimensionalized	Yes
Zero-Crossing Detection	No

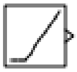
Purpose

Generate constantly increasing or decreasing signal

Library

Sources

Description

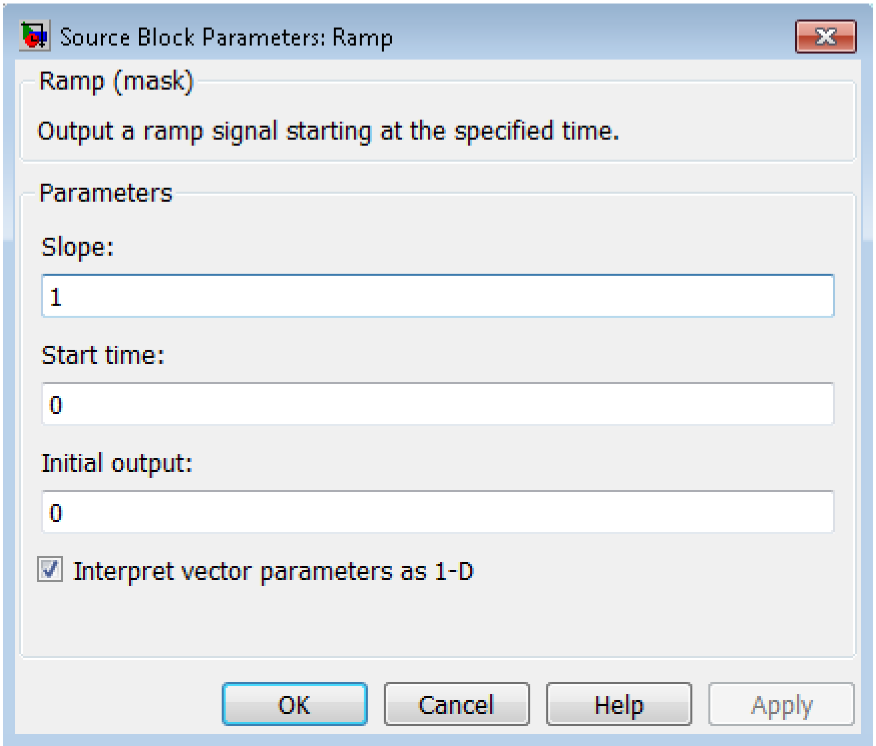


The Ramp block generates a signal that starts at a specified time and value and changes by a specified rate. The block's **Slope**, **Start time**, and **Initial output** parameters determine the characteristics of the output signal. All must have the same dimensions after scalar expansion.

Data Type Support

The Ramp block outputs signals of type `double`. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



Source Block Parameters: Ramp

Ramp (mask)

Output a ramp signal starting at the specified time.

Parameters

Slope: 1

Start time: 0

Initial output: 0

☑ Interpret vector parameters as 1-D

Slope

Specify the rate of change of the generated signal. The default is `1`.

Start time

Specify the time at which the block begins generating the signal. The default is `0`.

Initial output

Specify the initial value of the output signal. The default is `0`.

Interpret vector parameters as 1-D

If you select this option and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise, the output dimensionality is the same as that of the other parameters. See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Ramp block:

- `sldemo_VariableTransportDelay_pipe`

Characteristics

Sample Time	Inherited from driven block
Scalar Expansion	Yes
Dimensionalized	Yes
Zero-Crossing Detection	Yes

Random Number

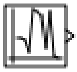
Purpose

Generate normally distributed random numbers

Library

Sources

Description



The Random Number block generates normally distributed random numbers. To generate uniformly distributed random numbers, use the Uniform Random Number block.

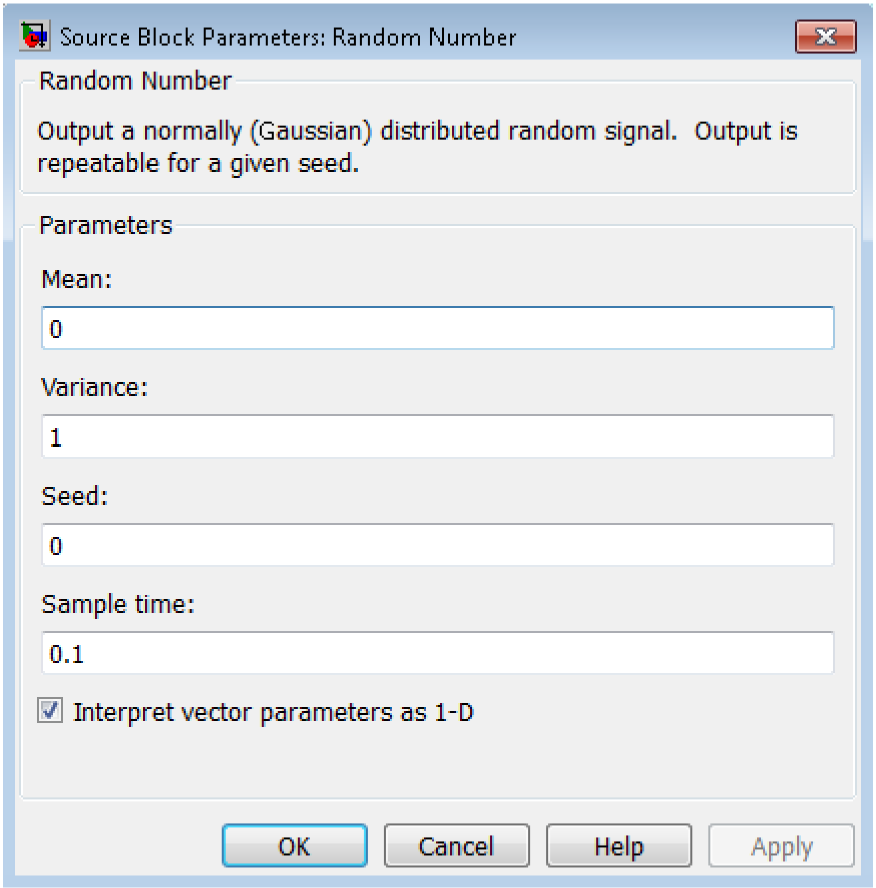
You can generate a repeatable sequence using any Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts. By default, the block produces a sequence that has a mean of 0 and a variance of 1. To generate a vector of random numbers with the same mean and variance, specify the **Seed** parameter as a vector.

Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the **Interpret vector parameters as 1-D** check box and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensionality as the parameters.

Data Type Support

The Random Number block outputs a real signal of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.


Source Block Parameters: Random Number

Random Number

Output a normally (Gaussian) distributed random signal. Output is repeatable for a given seed.

Parameters

Mean:

0

Variance:

1

Seed:

0

Sample time:

0.1

☑ Interpret vector parameters as 1-D

OK
Cancel
Help
Apply

Parameters and Dialog Box

Mean

Specify the mean of the random numbers. The default is `0`.

Variance

Specify the variance of the random numbers. The default is `1`.

Seed

Specify the starting seed for the random number generator. The default is `0`.

The seed must be 0 or a positive integer. Output is repeatable for a given seed.

Sample time

Specify the time interval between samples. The default is `0.1`, which matches the default sample time of the Band-Limited White Noise block. See "Specify Sample Time" in the Simulink documentation for more information.

Interpret vector parameters as 1-D

If you select this check box and the other parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. Otherwise, the block outputs a signal of the same dimensionality as the other parameters. For more information, see "Determining the Output Dimensions of Source Blocks" in the Simulink documentation.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Uniform Random Number

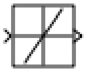
Purpose

Limit rate of change of signal

Library

Discontinuities

Description



The Rate Limiter block limits the first derivative of the signal passing through it. The output changes no faster than the specified limit. The derivative is calculated using this equation:‘

$$rate = \frac{u(i) - y(i-1)}{t(i) - t(i-1)}$$

u(i) and *t(i)* are the current block input and time, and *y(i–1)* and *t(i–1)* are the output and time at the previous step. The output is determined by comparing *rate* to the **Rising slew rate** and **Falling slew rate** parameters:

- If *rate* is greater than the **Rising slew rate** parameter (*R*), the output is calculated as

  $$y(i) = \Delta t \cdot R + y(i-1).$$

- If *rate* is less than the **Falling slew rate** parameter (*F*), the output is calculated as

  $$y(i) = \Delta t \cdot F + y(i-1).$$

- If *rate* is between the bounds of *R* and *F*, the change in output is equal to the change in input:

  $$y(i) = u(i)$$

When the block is running in continuous mode (for example, **Sample time mode** is `inherited` and **Sample time** of the driving block is zero), the **Initial condition** is ignored. The block output at `t = 0` is equal to the initial input:

$$y(0) = u(0)$$

When the block is running in discrete mode (for example, **Sample time mode** is `inherited` and **Sample time** of the driving block is nonzero), the **Initial condition** is preserved:

$$y(-1) = Ic$$

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:

$$rate = \frac{u(0) - y(-1)}{sample\,time}$$

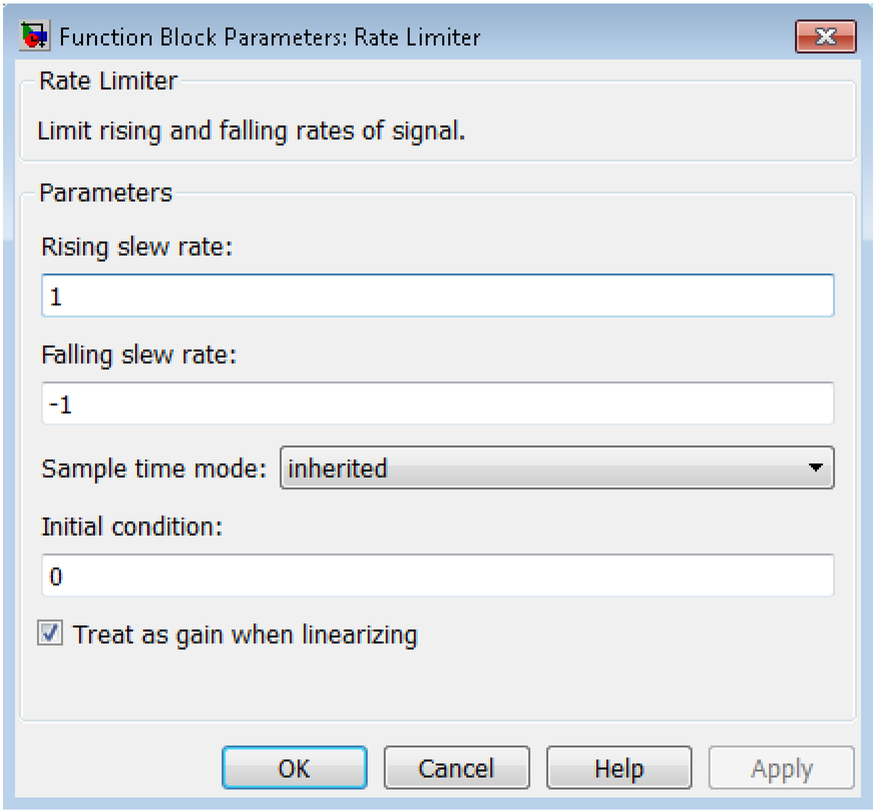
Note You cannot use a Rate Limiter block inside a Triggered Subsystem. Use the Rate Limiter Dynamic block instead.

Data Type Support

The Rate Limiter block accepts and outputs signals of any numeric data type that Simulink supports, except `Boolean`. The Rate Limiter block supports fixed-point data types.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



Function Block Parameters: Rate Limiter

Rate Limiter

Limit rising and falling rates of signal.

Parameters

Rising slew rate:

1

Falling slew rate:

-1

Sample time mode: inherited

Initial condition:

0

☑ Treat as gain when linearizing

Rising slew rate

Specify the limit of the derivative of an increasing input signal. This parameter is tunable for fixed-point inputs.

Falling slew rate

Specify the limit of the derivative of a decreasing input signal. This parameter is tunable for fixed-point inputs.

Sample time mode

Specify the sample time mode, `continuous` or `inherited` from the driving block.

Initial condition
Set the initial output of the simulation. Simulink software does not allow you to set the initial condition of this block to `inf` or `NaN`.

Treat as gain when linearizing
Linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the linearization commands to treat the gain as 1; otherwise, the commands treat the gain as 0.

Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous or inherited (specified in the **Sample time mode** parameter)
Scalar Expansion	Yes, of input and parameters
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also

Rate Limiter Dynamic

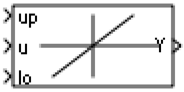
Purpose

Limit rising and falling rates of signal

Library

Discontinuities

Description



The Rate Limiter Dynamic block limits the rising and falling rates of the signal.

- The external signal `up` sets the upper limit on the rising rate of the signal.
- The external signal `lo` sets the lower limit on the falling rate of the signal.

Follow these guidelines when using the Rate Limiter Dynamic block:

- Ensure that the data types of `up` and `lo` are the same as the data type of the input signal `u`.

  When the lower limit uses a signed type and the input signal uses an unsigned type, the output signal keeps increasing regardless of the input and the limits.
- Use a fixed-step solver to simulate models that contain this block.

  Because the Rate Limiter Dynamic block does not support continuous sample time, simulation with a variable-step solver causes an error.

Data Type Support

The Rate Limiter Dynamic block accepts input signals of the following data types:

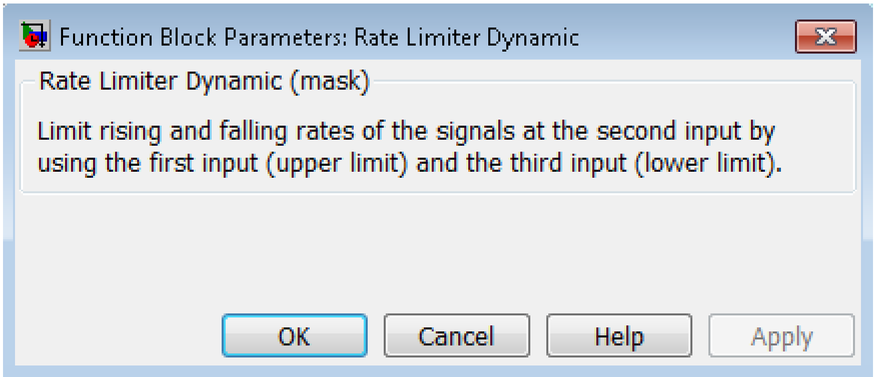
- Floating point

- Built-in integer
- Fixed point

The data type of the output signal `Y` matches the data type of the input signal `u`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Function Block Parameters: Rate Limiter Dynamic

Rate Limiter Dynamic (mask)

Limit rising and falling rates of the signals at the second input by using the first input (upper limit) and the third input (lower limit).

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

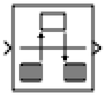
Rate Limiter

Rate Transition

Purpose Handle transfer of data between blocks operating at different rates

Library Signal Attributes

Description



Transition Handling Options

The Rate Transition block transfers data from the output of a block operating at one rate to the input of a block operating at a different rate. Use the block parameters to trade data integrity and deterministic transfer for faster response or lower memory requirements. To learn about data integrity and deterministic data transfer, see “Data Transfer Problems” in the Simulink Coder documentation.

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**
• 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:

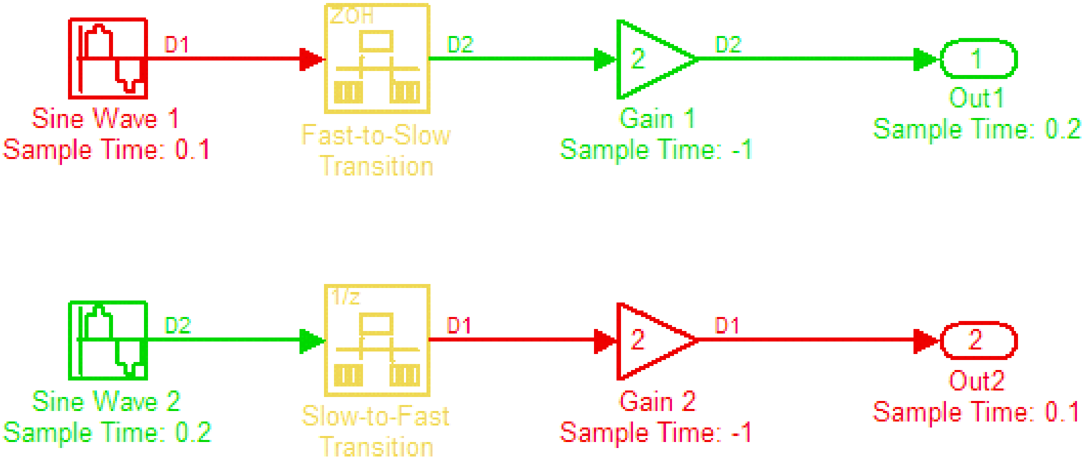
- Sample times of the ports to which the block connects (see “Effects of Synchronous Sample Times” on page 1-1369 and “Effects of Asynchronous Sample Times” on page 1-1371)
- Priorities of the tasks for the source and destination sample times (see “Sample time properties” in the Simulink documentation)
- 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” in the Simulink documentation) to display the relative rates that the block bridges. Consider, for example, the following model:


D1
ZOH
D2
2
D2
1
Sine Wave 1
Sample Time: 0.1
Fast-to-Slow
Transition
Gain 1
Sample Time: -1
Out1
Sample Time: 0.2
D2
1/z
D1
2
D1
2
Sine Wave 2
Sample Time: 0.2
Slow-to-Fast
Transition
Gain 2
Sample Time: -1
Out2
Sample Time: 0.1

Sample-time colors and the block behavior label show that the Rate Transition block at the top of the diagram acts as a zero-order hold in a fast-to-slow transition and the bottom Rate Transition block acts as a unit delay in a slow-to-fast transition.

For more information, see “Handle Rate Transitions” in the Simulink Coder documentation.

Effects of Synchronous Sample Times

The following table summarizes how each label appears if the sample times of the input and output ports (`inTs` and `outTs`) are periodic, or synchronous.

Block Settings		**Block Label**		
Rate Transition	**Conditions for Rate Transition Block**	**With Data Integrity and Determinism**	**With Only Data Integrity**	**Without Data Integrity or Determinism**
`inTs = outTs` (Equal)	`inTsOffset` < `outTsOffset`	None (error)	`Buf`	`Copy` or `NoOp` (see note that follows the table)
	`inTsOffset` = `outTsOffset`	`Copy` or `NoOp` (see note that follows the table)	`Copy` or `NoOp` (see note that follows the table)	
	`inTsOffset` > `outTsOffset`	None (error)	`Db_buf`	
`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				

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`: 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 **Block reduction** in the **Optimization** pane of the Configuration Parameters dialog box, `Copy` reduces to `NoOp`. No code generation occurs for a Rate Transition block with a `NoOp` label. To prevent a block from being reduced when block reduction is on, add a test point to the block output (see “Test Points” in the Simulink documentation).

Effects of Asynchronous Sample Times

The following table summarizes how each label appears if the sample time of the input or output port (`inTs` or `outTs`) is not periodic, or asynchronous.

Block Settings	Block Label		
	With Data Integrity and Determinism	With Only Data Integrity	Without Data Integrity or Determinism
`inTs` = `outTs`	`Copy`	`Copy`	`Copy`
`inTs` ≠ `outTs`	None (error)	`Db_buf`	

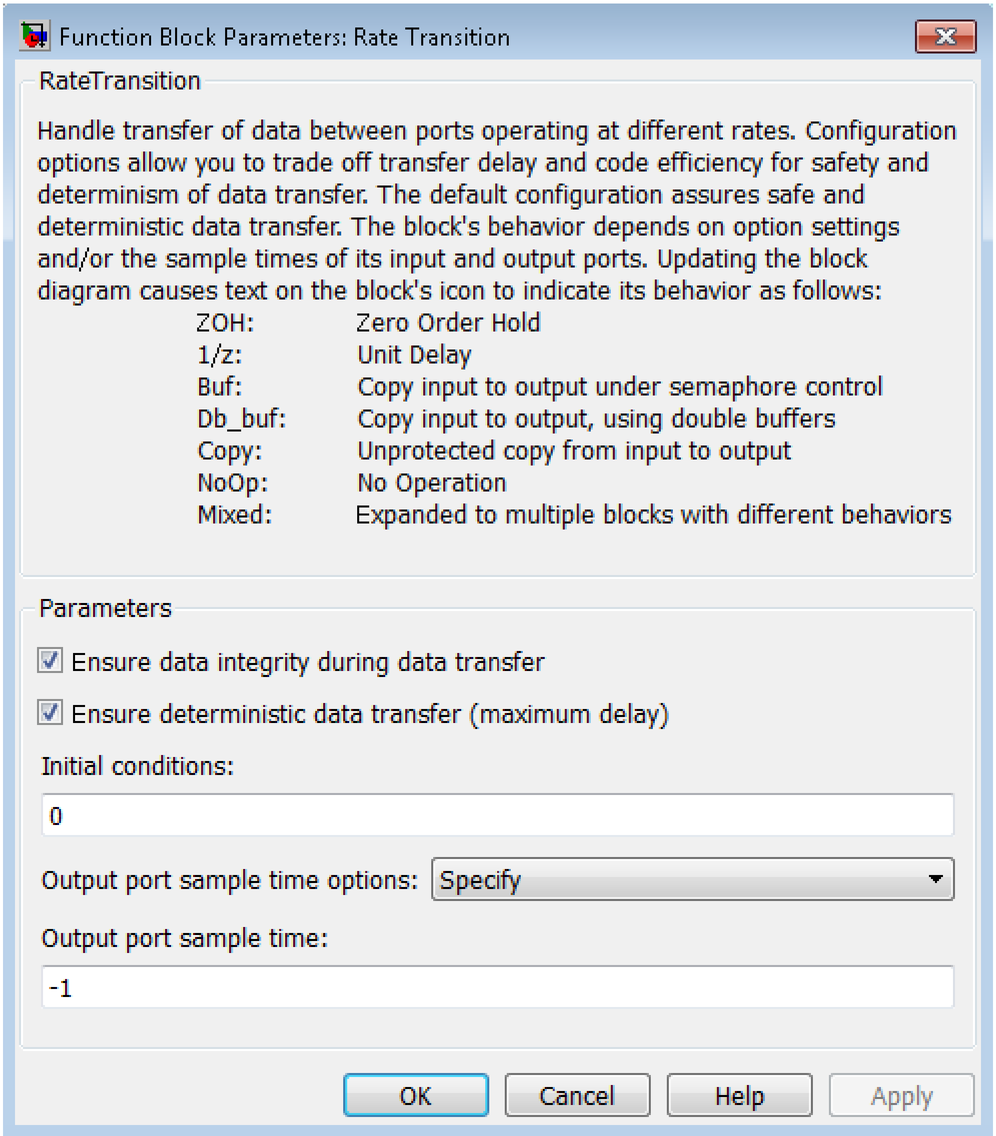
KEY

- `inTs, outTs`: Sample times of input and output ports, respectively

Data Type Support

The Rate Transition block accepts signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



RateTransition

Handle transfer of data between ports operating at different rates. Configuration options allow you to trade off transfer delay and code efficiency for safety and determinism of data transfer. The default configuration assures safe and deterministic data transfer. The block's behavior depends on option settings and/or the sample times of its input and output ports. Updating the block diagram causes text on the block's icon to indicate its behavior as follows:

ZOH:	Zero Order Hold
1/z:	Unit Delay
Buf:	Copy input to output under semaphore control
Db_buf:	Copy input to output, using double buffers
Copy:	Unprotected copy from input to output
NoOp:	No Operation
Mixed:	Expanded to multiple blocks with different behaviors

Parameters

☑ Ensure data integrity during data transfer

☑ Ensure deterministic data transfer (maximum delay)

Initial conditions:

0

Output port sample time options: Specify

Output port sample time:

-1

Parameters

and Dialog Box

Ensure data integrity during data transfer

Selecting this check box results in generated code that ensures data integrity when the block transfers data. If you select this check box and the transfer is nondeterministic (see **Ensure deterministic data transfer** below), the generated code uses double-buffering to prevent the fast block from interrupting the data transfer. Otherwise, the generated code uses a copy operation to effect the data transfer. The copy operation consumes less memory than double-buffering but is also interruptible, which can lead to loss of data during nondeterministic data transfers. Select this check box if you want the generated code to operate with maximum responsiveness (i.e., nondeterministically) and data integrity. For more information, see "Rate Transition Block Options" in the Simulink Coder documentation.

Ensure deterministic data transfer (maximum delay)

Selecting this check box results in generated code that transfers data at the sample rate of the slower block, that is, deterministically. If you do not select this check box, data transfers occur as soon as new data is available from the source block and the receiving block is ready to receive the data. You avoid transfer delays, thus ensuring that the system operates with maximum responsiveness. However, transfers can occur unpredictably, which is undesirable in some applications. For more information, see "Rate Transition Block Options" in the Simulink Coder documentation.

Initial conditions

This parameter applies only to slow-to-fast transitions. It specifies the initial output of the Rate Transition block at the beginning of a transition when there is no output from the slow block connected to the input of the Rate Transition block. Simulink does not allow the initial output of this block to be `Inf` or `NaN`.

Output port sample time options

Specifies a mode for setting the output port sample time. The options are:

- `Specify` — Allows you to use the **Output port sample time** parameter to specify the output rate to which the Rate Transition block converts its input rate.
- `Inherit` — Specifies that the Rate Transition block inherits an output rate from the block to which the output port is connected.
- `Multiple of input port sample time` — Allows you to use the **Sample time multiple (>0)** parameter to specify the Rate Transition block output rate as a multiple of its input rate.

If you specify `Inherit` and all blocks connected to the output port also inherit sample time, the fastest sample time in the model applies.

Output port sample time

This parameter is visible when you set **Output port sample time options** to `Specify`. Enter a value that specifies the output rate to which the block converts its input rate. The default value (`-1`) specifies that the Rate Transition block inherits the output rate from the block to which the output port is connected. See “Specify Sample Time” in the Simulink documentation for information on how to specify the output rate.

Sample time multiple (>0)

This parameter is visible when you set **Output port sample time options** to `Multiple of input port sample time`. Enter a positive value that specifies the output rate as a multiple of the input port sample time. The default value (`1`) specifies that the output rate is the same as the input rate. A value of `0.5` specifies that the output rate is half of the input rate, while a value of `2` specifies that the output rate is twice the input rate.

Bus Support

The Rate Transition block is a bus-capable block. The input can be a virtual or nonvirtual bus signal, with the restriction that **Initial conditions** must be zero, a nonzero scalar, or a finite numeric structure. For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Rate Transition block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “About Composite Signals” and “Bus-Capable Blocks” in the Simulink documentation for more information.

You can use an array of buses as an input signal to a Rate Transition block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Characteristics

Bus-capable	Yes, with restrictions as noted above.
Direct Feedthrough	No, for slow-to-fast transitions for which you select the **Ensure data integrity during data transfer** check box. Yes, otherwise.
Sample Time	This block supports discrete-to-discrete transitions.
Scalar Expansion	Yes, of input.
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

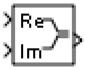
Convert real and/or imaginary inputs to complex signal

Library

Math Operations

Description

The Real-Imag to Complex block converts real and/or imaginary inputs to a complex-valued output signal.



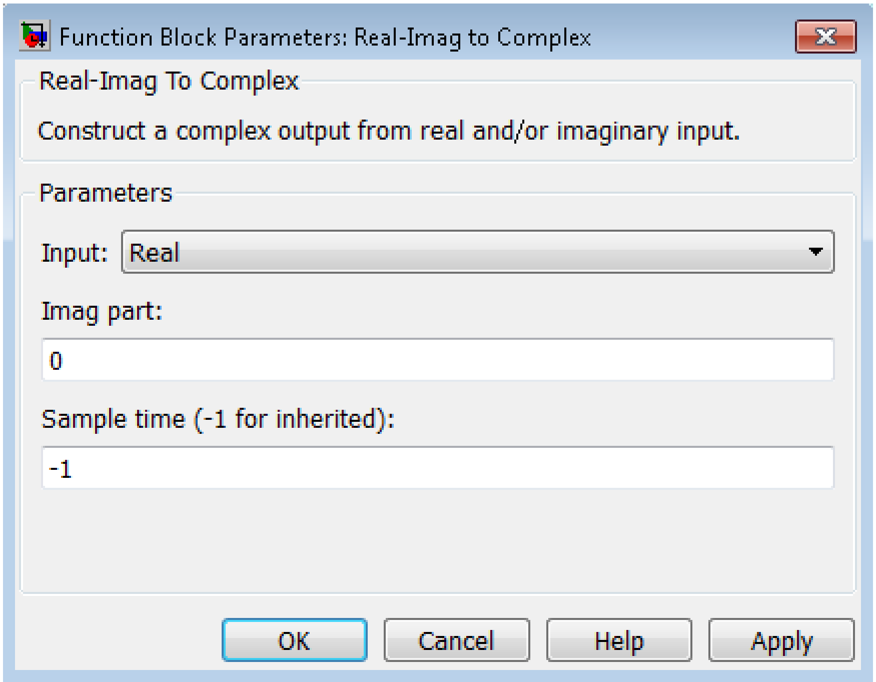
The inputs can both be arrays (vectors or matrices) of equal dimensions, or one input can be an array and the other a scalar. If the block has an array input, the output is a complex array of the same dimensions. The elements of the real input map to the real parts of the corresponding complex output elements. The imaginary input similarly maps to the imaginary parts of the complex output signals. If one input is a scalar, it maps to the corresponding component (real or imaginary) of all the complex output signals.

Data Type Support

The block accepts input signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

Complex fixed-point signals must have trivial slope and zero bias. For more information about support for fixed-point data types, see “Scaling” in the Fixed-Point Designer documentation.



Function Block Parameters: Real-Imag to Complex

Real-Imag To Complex

Construct a complex output from real and/or imaginary input.

Parameters

Input: Real

Imag part:

0

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Input

Specify the kind of input: a real input, an imaginary input, or both.

Real (Imag) part

This parameter appears only when you set **Input** to `Real` or `Imag`. If the input is a real-part signal, this parameter specifies the constant imaginary part of the output signal. If the input is the imaginary part, this parameter specifies the constant real part of the output signal. The title of this parameter changes to reflect its usage.

Sample time (-1 for inherited)
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes, of the input when the function requires two inputs
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

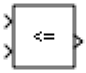
Complex to Real-Imag

Relational Operator

Purpose Perform specified relational operation on inputs

Library Logic and Bit Operations

Description



Two-Input Mode

By default, the Relational Operator block compares two inputs using the **Relational operator** parameter that you specify. The first input corresponds to the top input port and the second input to the bottom input port. (See “How to Rotate a Block” in the Simulink documentation for a description of the port order for various block orientations.)

You can specify one of the following operations in two-input mode:

Operation	Description
`==`	TRUE if the first input is equal to the second input
`~=`	TRUE if the first input is not equal to the second input
`<`	TRUE if the first input is less than the second input
`<=`	TRUE if the first input is less than or equal to the second input
`>=`	TRUE if the first input is greater than or equal to the second input
`>`	TRUE if the first input is greater than the second input

You can specify inputs as scalars, arrays, or a combination of a scalar and an array.

For...	The output is...
Scalar inputs	A scalar
Array inputs	An array of the same dimensions, where each element is the result of an element-by-element comparison of the input arrays
Mixed scalar and array inputs	An array, where each element is the result of a comparison between the scalar and the corresponding array element

The input with the smaller positive range is converted to the data type of the other input offline using round-to-nearest and saturation. This conversion occurs before the comparison.

You can specify the output data type using the **Output data type** parameter. The output equals 1 for TRUE and 0 for FALSE.

Tip Select an output data type that represents zero exactly. Data types that satisfy this condition include signed and unsigned integers and any floating-point data type.

One-Input Mode

When you select one of the following operations for **Relational operator**, the block switches to one-input mode.

Operation	Description
`isInf`	TRUE if the input is `Inf`
`isNaN`	TRUE if the input is `NaN`
`isFinite`	TRUE if the input is finite

For an input that is not floating point, the block produces the following output.

Data Type	Operation	Block Output
• 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

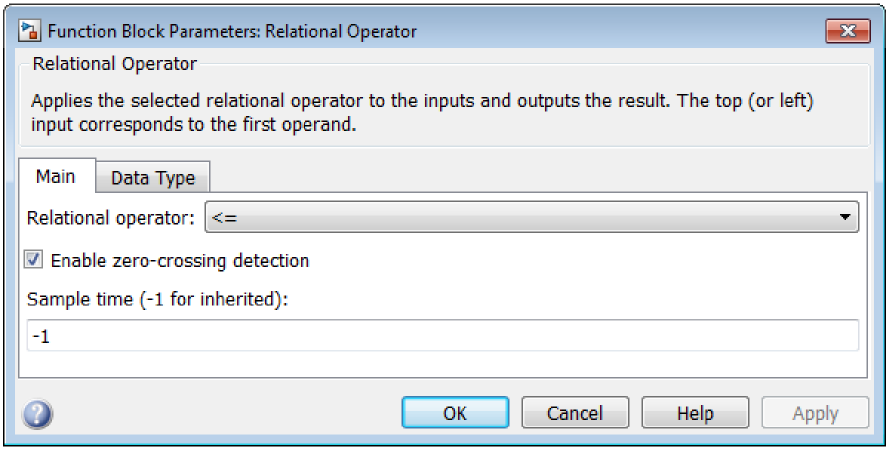
Data Type Support

The Relational Operator block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. For two-input mode, one input can be real and the other complex when the operator is == or ~=. Complex inputs work only for ==, ~=, `isInf`, `isNaN`, and `isFinite`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Relational Operator block dialog box appears as follows:



Function Block Parameters: Relational Operator

Relational Operator

Applies the selected relational operator to the inputs and outputs the result. The top (or left) input corresponds to the first operand.

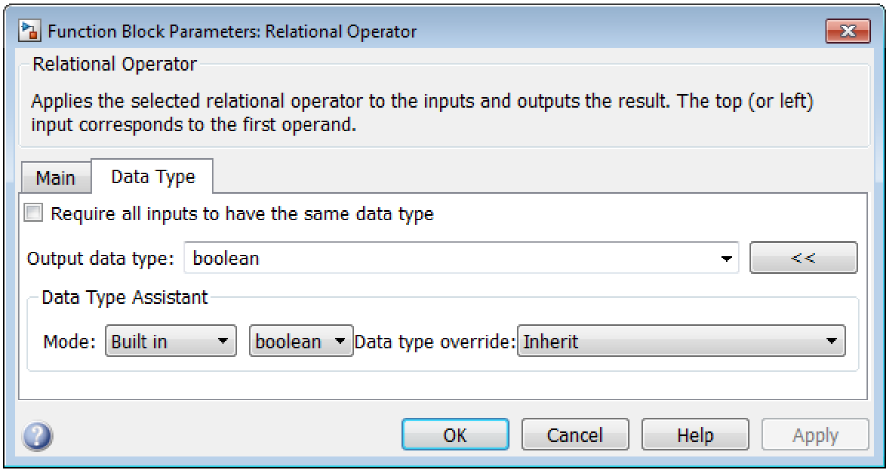
Relational operator: <=

☑ Enable zero-crossing detection

Sample time (-1 for inherited):

-1

The **Data Type** pane of the Relational Operator block dialog box appears as follows:



Function Block Parameters: Relational Operator

Relational Operator

Applies the selected relational operator to the inputs and outputs the result. The top (or left) input corresponds to the first operand.

☐ Require all inputs to have the same data type

Output data type: boolean

Data Type Assistant

Mode: Built in boolean Data type override: Inherit

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Relational operator

Specify the operation for comparing two inputs or determining the signal type of one input.

Settings

Default: `<=`

`==`
: TRUE if the first input is equal to the second input

`~=`
: TRUE if the first input is not equal to the second input

`<`
: TRUE if the first input is less than the second input

`<=`
: TRUE if the first input is less than or equal to the second input

`>=`
: TRUE if the first input is greater than or equal to the second input

`>`
: TRUE if the first input is greater than the second input

`isInf`
: TRUE if the input is `Inf`

`isNaN`
: TRUE if the input is `NaN`

`isFinite`
: TRUE if the input is finite

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Settings

Default: On

☑ On
   Enable zero-crossing detection.

☐ Off
   Do not enable zero-crossing detection.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Settings

Default: Off

☑ On

Require that all inputs have the same data type.

☐ Off

Do not require that all inputs have the same data type.

Dependency

This check box is not available when you select `isInf`, `isNaN`, or `isFinite` for **Relational operator**, because the block is in one-input mode.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `boolean`

`Inherit: Logical (see Configuration Parameters: Optimization)`

Uses the **Implement logic signals as Boolean data** configuration parameter (see "Implement logic signals as Boolean data (vs. double)") to specify the output data type.

Note This option supports models created before the `boolean` option was available. Use one of the other options, preferably `boolean`, for new models.

`boolean`

Specifies output data type is `boolean`.

`fixdt(1,16)`

Specifies output data type is `fixdt(1,16)`.

`<data type expression>`

Uses the name of a data type object, for example, `Simulink.NumericType`.

Tip To enter a built-in data type (`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`), enclose the expression in single quotes. For example, enter `'double'` instead of `double`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Built in`

`Inherit`
: Specifies inheritance rules for data types. Selecting `Inherit` enables `Logical (see Configuration Parameters: Optimization)`.

`Built in`
: Specifies built-in data types. Selecting `Built in` enables `boolean`.

`Fixed point`
: Specifies fixed-point data types.

`Expression`
: Specifies expressions that evaluate to data types.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant".

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specify the fixed-point data as signed.

`Unsigned`
: Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Integer`

`Integer`
: Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0.

Command-Line Information

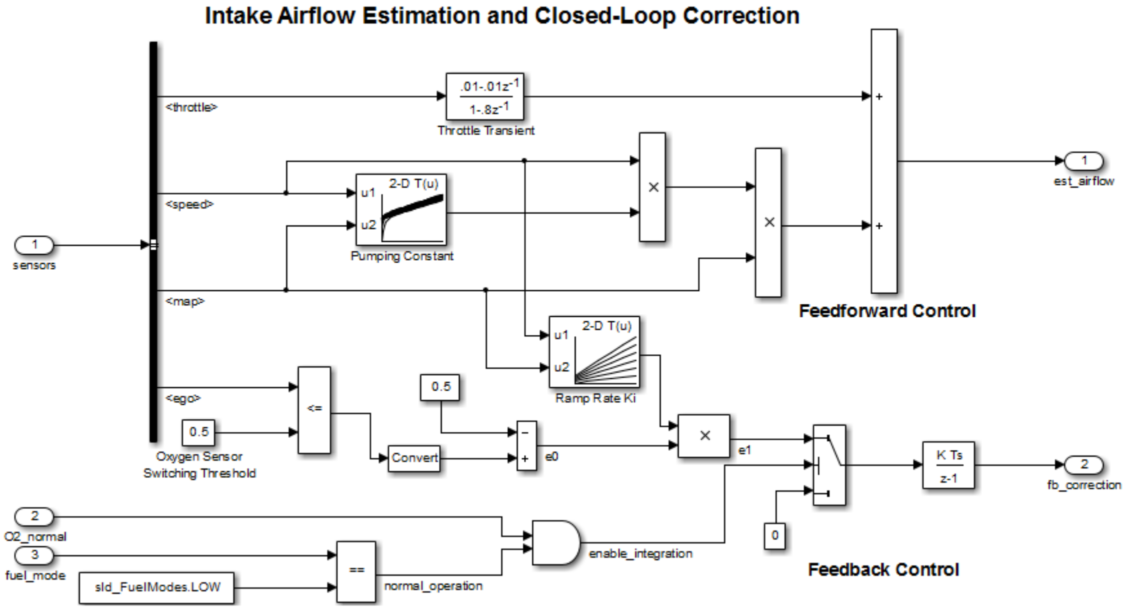
See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type”.

Examples

In the `sldemo_fuelsys` model, the `fuel_rate_control/airflow_calc` subsystem uses two Relational Operator blocks:


Intake Airflow Estimation and Closed-Loop Correction
<throttle>
.01-.01z-1
1-.8z-1
Throttle Transient
<speed>
u1
u2
2-D T(u)
Pumping Constant
×
×
+
+
1
est_airflow
1
sensors
<map>
Feedforward Control
u1
u2
2-D T(u)
Ramp Rate Ki
<ego>
<=
0.5
0.5
Oxygen Sensor
Switching Threshold
-
+
Convert
e0
×
e1
K Ts
z-1
2
fb_correction
0
2
O2_normal
3
fuel_mode
sld_FuelModes.LOW
==
normal_operation
enable_integration
Feedback Control

Both Relational Operator blocks operate in two-input mode.

The block that uses this operator...	Compares...
`<=`	The value of the oxygen sensor to the threshold value, 0.5
`==`	The value of the fuel mode to the ideal value, `sld_FuelModes.LOW`

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of inputs
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled

Relay

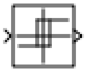
Purpose

Switch output between two constants

Library

Discontinuities

Description



The Relay block allows its output to switch between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.

The **Switch on point** value must be greater than or equal to the **Switch off point**. Specifying a **Switch on point** value greater than the **Switch off point** models hysteresis, whereas specifying equal values models a switch with a threshold at that value.

Note When the initial input falls *between* the **Switch off point** and **Switch on point** values, the initial output is the value when the relay is off.

Data Type Support

The Relay block accepts signals of the following data types:

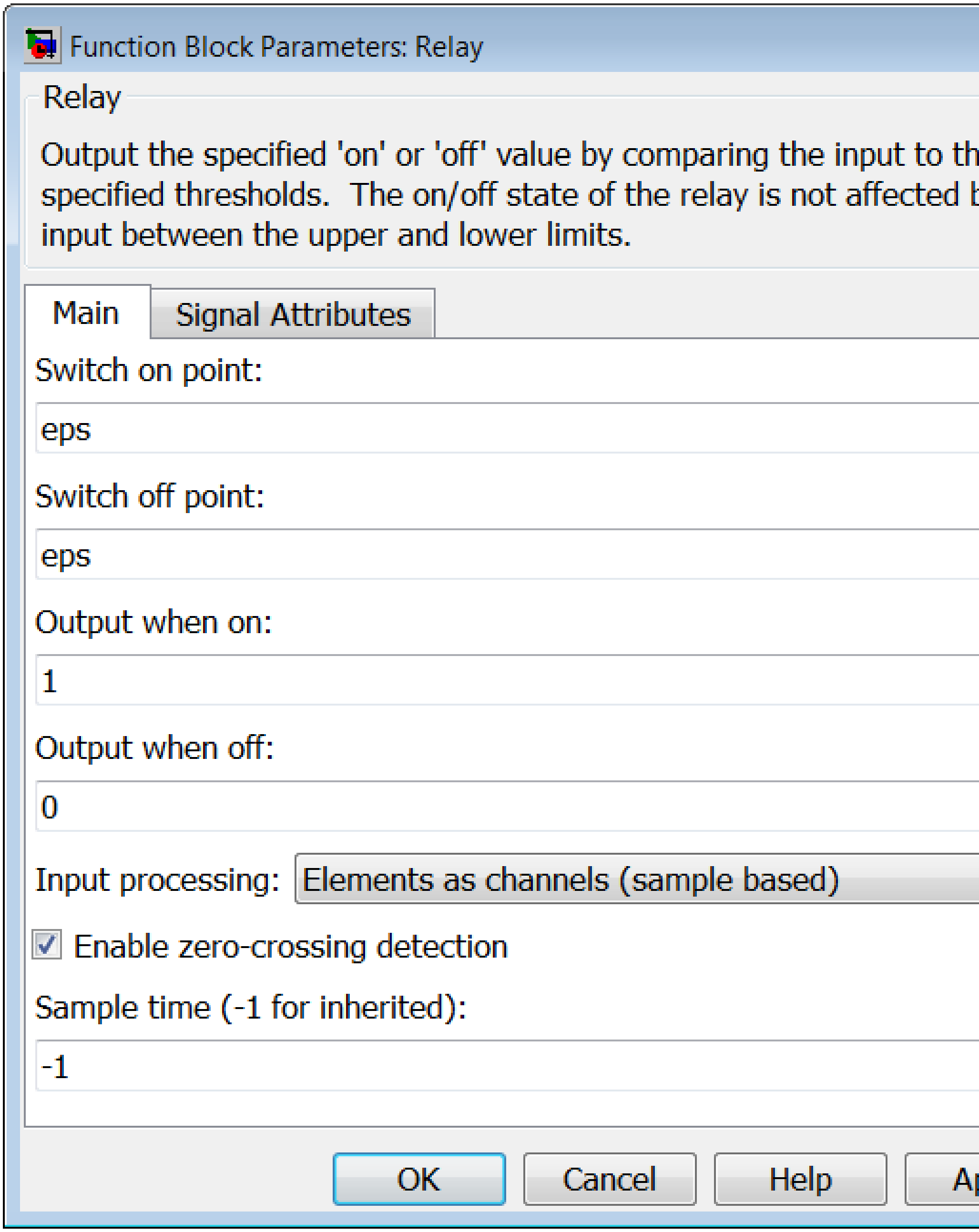
- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated (output only)

If **Output when on** or **Output when off** is an enumerated value, both must be of the same enumerated type as the output.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Relay block dialog box appears as follows:



Function Block Parameters: Relay

Relay

Output the specified 'on' or 'off' value by comparing the input to th
specified thresholds. The on/off state of the relay is not affected b
input between the upper and lower limits.

Switch on point: eps

Switch off point: eps

Output when on: 1

Output when off: 0

Input processing: Elements as channels (sample based)

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

Switch on point

The “on” threshold for the relay. The **Switch on point** parameter is converted to the input data type offline using round-to-nearest and saturation.

Switch off point

The “off” threshold for the relay. The **Switch off point** parameter is converted to the input data type offline using round-to-nearest and saturation.

Output when on

The output when the relay is on.

Output when off

The output when the relay is off.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

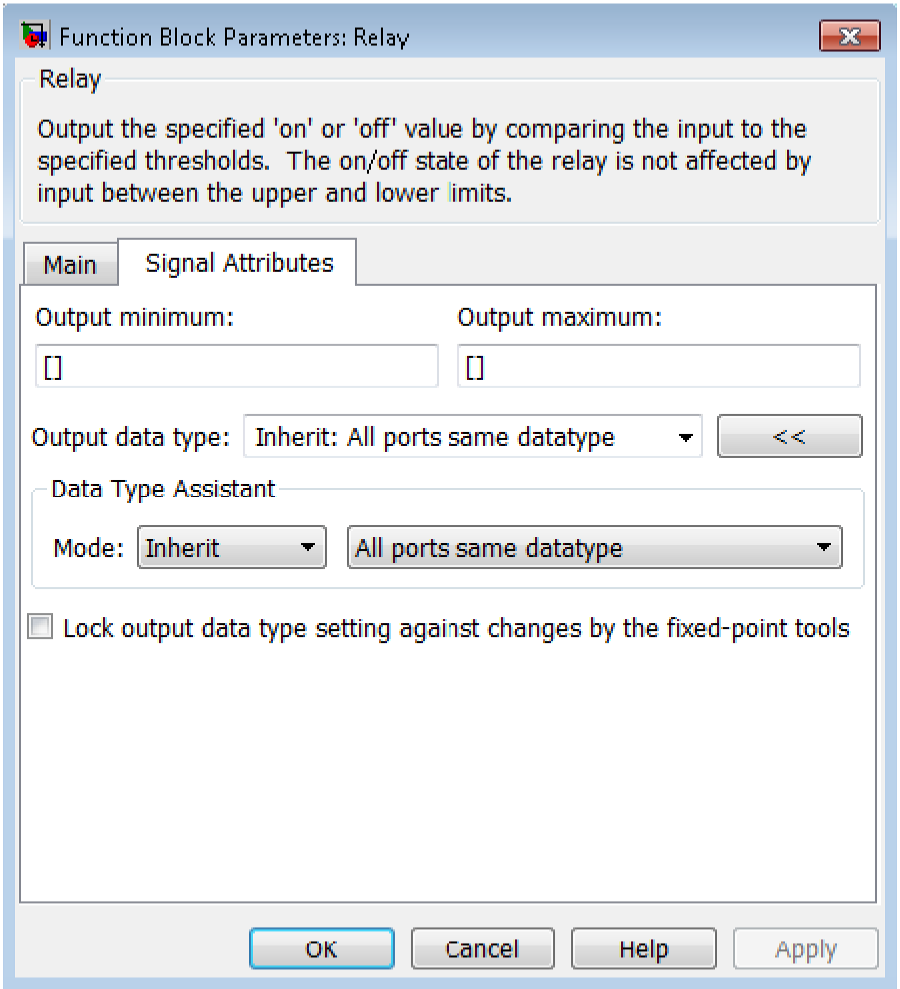
Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

Enable zero-crossing detection
Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Sample time (-1 for inherited)
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation for more information.

The **Signal Attributes** pane of the Relay block dialog box appears as follows:



Function Block Parameters: Relay

Relay

Output the specified 'on' or 'off' value by comparing the input to the specified thresholds. The on/off state of the relay is not affected by input between the upper and lower limits.

Output minimum: []

Output maximum: []

Output data type: Inherit: All ports same datatype

Data Type Assistant

Mode: Inherit — All ports same datatype

☐ Lock output data type setting against changes by the fixed-point tools

Output minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`
- 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 “Specify Block Output Data Types” in *Simulink User’s Guide* for more information.

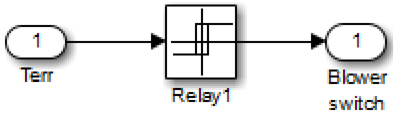
Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting”.

Examples

The `sldemo_househeat` model shows how you can use the Relay block.

The Relay block appears in the Thermostat subsystem.



The thermostat allows fluctuations of 5 degrees Fahrenheit above or below the desired room temperature. If air temperature drops below 65 degrees Fahrenheit, the thermostat turns on the heater. The Relay block outputs a value of 1 to turn on the heater and a value of 0 to turn off the heater.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes

Multidimensionalized	No
Zero-Crossing Detection	Yes, if enabled

Repeating Sequence

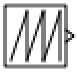
Purpose

Generate arbitrarily shaped periodic signal

Library

Sources

Description



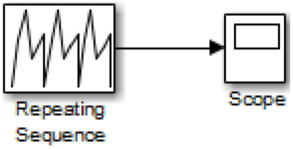
The Repeating Sequence block outputs a periodic scalar signal having a waveform that you specify using the **Time values** and **Output values** parameters. The **Time values** parameter specifies a vector of output times. The **Output values** parameter specifies a vector of signal amplitudes at the corresponding output times. Together, the two parameters specify a sampling of the output waveform at points measured from the beginning of the interval over which the waveform repeats (the period of the signal).

By default, both parameters are `[0 2]`. These default settings specify a sawtooth waveform that repeats every `2` seconds from the start of the simulation and has a maximum amplitude of `2`.

Algorithm

The block sets the input period as the difference between the first and last value of the **Time values** parameter. The output at any time `t` is the output at time `t = t-n*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.

In the following model, the Repeating Sequence block defines **Time values** as [0:0.1:0.5] and **Output values** as [0 15 25 09 13 17]. The stop time of the simulation is 0.7 second.

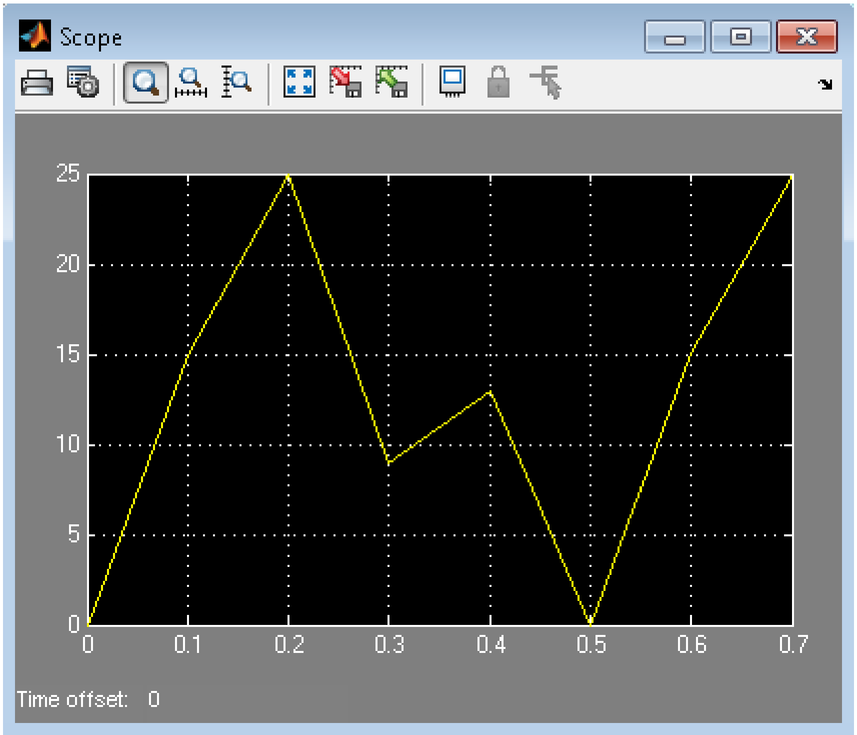


For the Repeating Sequence block:

- The input period is 0.5.

- The output at any time `t` is the output at time `t = t-0.5n`, where `n` = 0, 1, 2, and so on.
- The sequence repeats at `t = 0.5n`, where `n` = 0, 1, 2, and so on.

When you run the model, you get the following results:

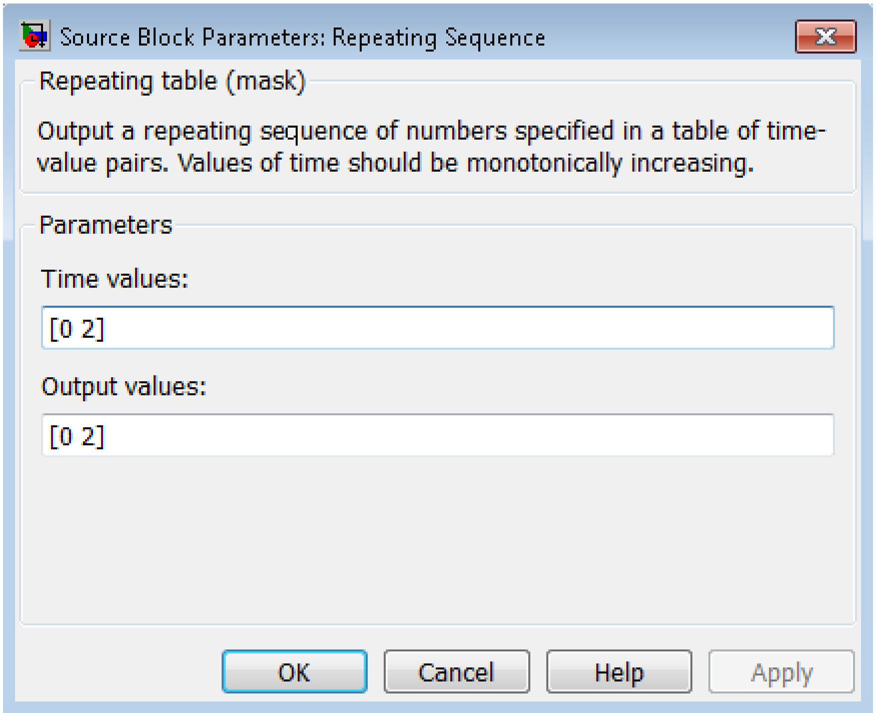

Scope
0
5
10
15
20
25
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
Time offset: 0

At t = 0.5, the expected output is equal to the output at t = 0, which is 0. Therefore, the last value in the **Output values** vector [0 15 25 09 13 17] does not appear.

Data Type Support

The Repeating Sequence block outputs real signals of type `double`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.


Source Block Parameters: Repeating Sequence
Repeating table (mask)
Parameters
OK
Cancel
Help
Apply

Output a repeating sequence of numbers specified in a table of time-value pairs. Values of time should be monotonically increasing.

Time values:
[0 2]

Output values:
[0 2]

Parameters and Dialog Box

Time values

Specify a vector of monotonically increasing time values. The default is `[0 2]`.

Output values

Specify a vector of output values. Each element corresponds to the time value in the same column. The default is `[0 2]`.

Examples

The following Simulink examples show how to use the Repeating Sequence block:

- `sldemo_fuelsys`
- `sldemo_hydrod`
- `sldemo_VariableTransportDelay`

Characteristics

Sample Time	Continuous
Scalar Expansion	No
Dimensionalized	No
Zero-Crossing Detection	No

See Also

Repeating Sequence Interpolated, Repeating Sequence Stair

Repeating Sequence Interpolated

Purpose

Output discrete-time sequence and repeat, interpolating between data points

Library

Sources

Description

The Repeating Sequence Interpolated block outputs a discrete-time sequence and then repeats it. Between data points, the block uses the method you specify for the **Lookup Method** parameter to determine the output.

Data Type Support

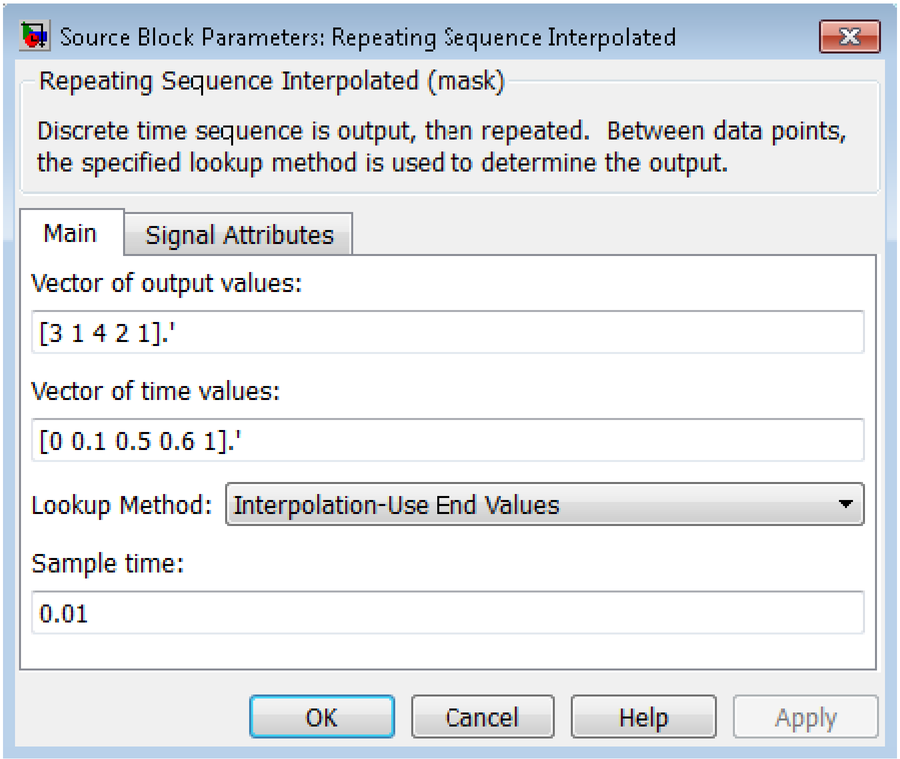
The Repeating Sequence Interpolated block outputs signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Repeating Sequence Interpolated block dialog box appears as follows:


Source Block Parameters: Repeating Sequence Interpolated
Main
Signal Attributes
OK
Cancel
Help
Apply

Repeating Sequence Interpolated (mask)

Discrete time sequence is output, then repeated. Between data points, the specified lookup method is used to determine the output.

Vector of output values:

[3 1 4 2 1].'

Vector of time values:

[0 0.1 0.5 0.6 1].'

Lookup Method: Interpolation-Use End Values

Sample time:

0.01

Vector of output values
: Specify the column vector containing output values of the discrete time sequence.

Vector of time values
: Specify the column vector containing time values. The time values must be strictly increasing, and the vector must have the same size as the vector of output values.

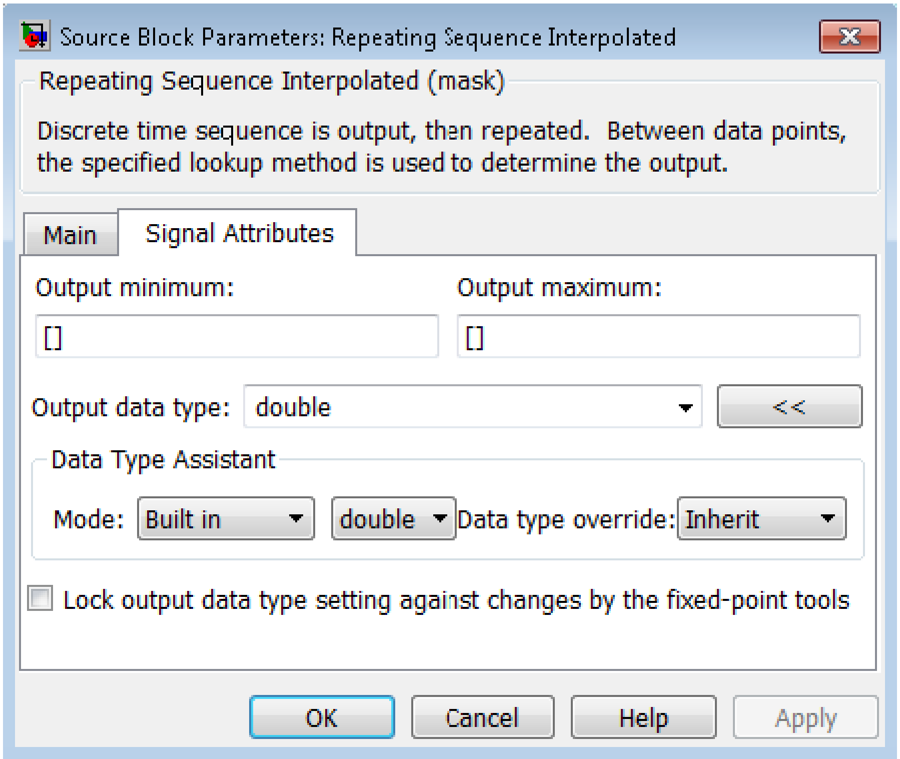
Lookup Method

Specify the lookup method to determine the output between data points.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

The **Signal Attributes** pane of the Repeating Sequence Interpolated block dialog box appears as follows:



Source Block Parameters: Repeating Sequence Interpolated

Repeating Sequence Interpolated (mask)

Discrete time sequence is output, then repeated. Between data points, the specified lookup method is used to determine the output.

Output minimum: []

Output maximum: []

Output data type: double

Data Type Assistant

Mode: Built in double Data type override: Inherit

☐ Lock output data type setting against changes by the fixed-point tools

Output minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Block Output Data Types” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting”.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Repeating Sequence, Repeating Sequence Stair

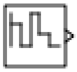
Purpose

Output and repeat discrete time sequence

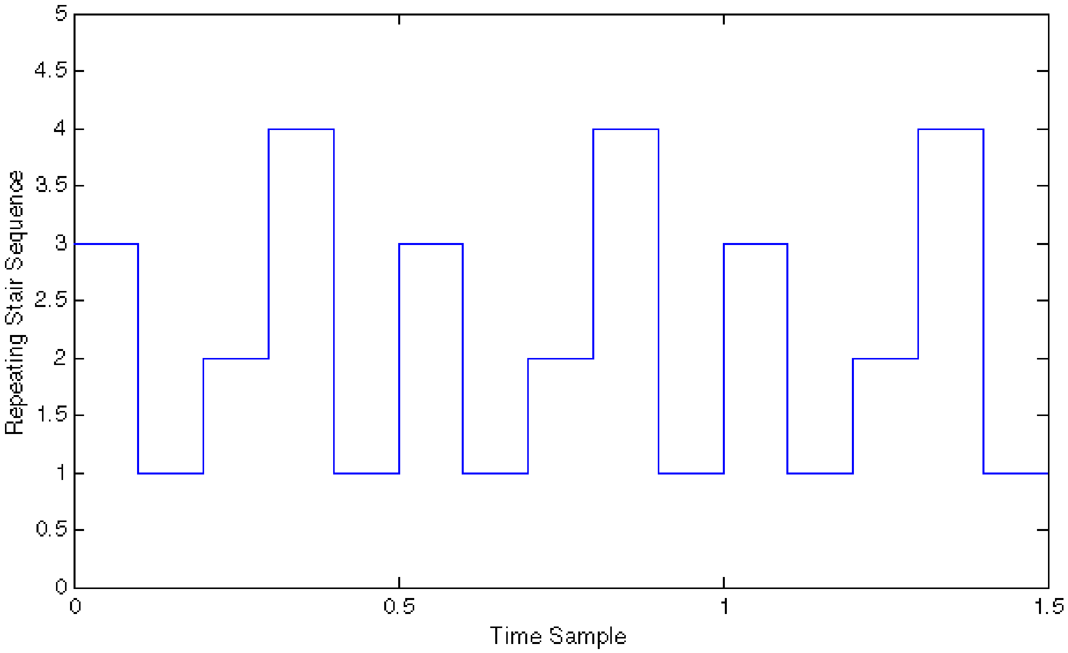
Library

Sources

Description



The Repeating Sequence Stair block outputs and repeats a stair sequence that you specify with the **Vector of output values** parameter. For example, you can specify the vector as `[3 1 2 4 1]'`, which produces the following stair sequence:



Data Type Support

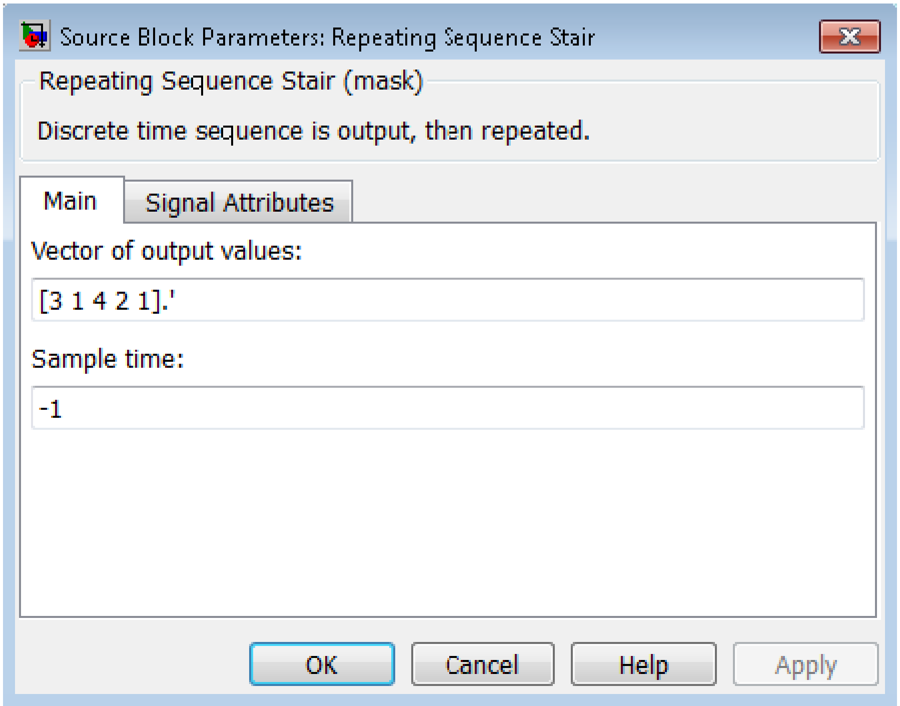
The Repeating Sequence Stair block outputs signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Repeating Sequence Stair block dialog box appears as follows:



Source Block Parameters: Repeating Sequence Stair

Repeating Sequence Stair (mask)

Discrete time sequence is output, then repeated.

Vector of output values: [3 1 4 2 1].'

Sample time: -1

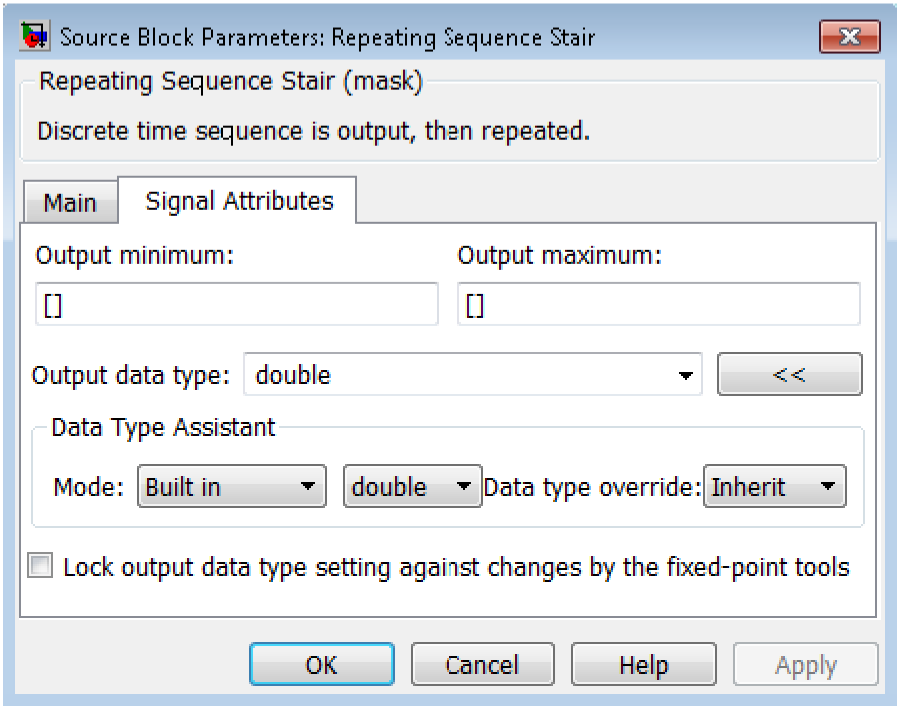
Vector of output values

Specify the vector containing values of the repeating stair sequence.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

The **Signal Attributes** pane of the Repeating Sequence Stair block dialog box appears as follows:


Source Block Parameters: Repeating Sequence Stair
Main
Signal Attributes
<<
OK
Cancel
Help
Apply

Repeating Sequence Stair (mask)

Discrete time sequence is output, then repeated.

Output minimum: []

Output maximum: []

Output data type: double

Data Type Assistant

Mode: Built in double Data type override: Inherit

☐ Lock output data type setting against changes by the fixed-point tools

Output minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")
- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Check Parameter Values”)
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Block Output Data Types” in the *Simulink User’s Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting”.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Zero-Crossing Detection	No

See Also Repeating Sequence, Repeating Sequence Interpolated

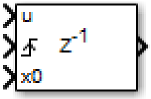
Purpose

Delay input signal by variable sample period and reset with external signal

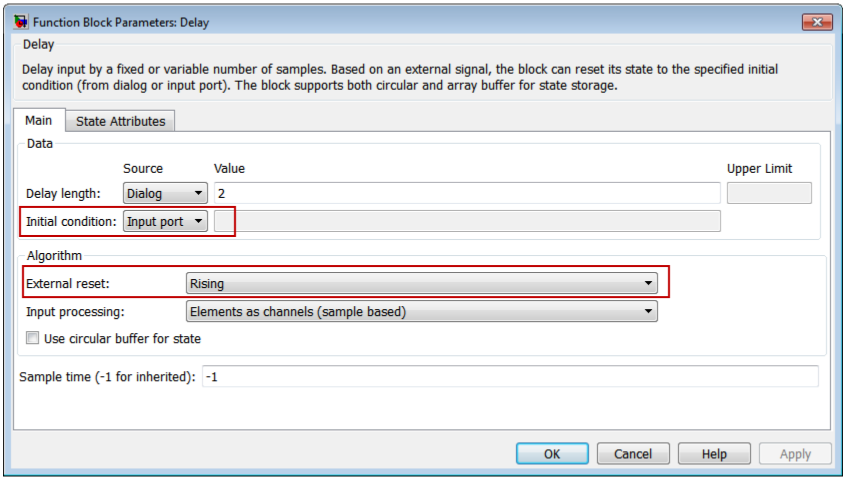
Library

Discrete

Description



The Resettable Delay block is a variant of the Delay block that has the source of the initial condition set to `Input port` and the external reset algorithm set to `Rising`, by default.



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:	Dialog	2	
Initial condition:	Input port		

Algorithm

External reset: Rising

Input processing: Elements as channels (sample based)

☐ Use circular buffer for state

Sample time (-1 for inherited): -1

See Also

- Delay
- Variable Integer Delay

Reshape

Purpose Change dimensionality of signal

Library Math Operations

Description The Reshape block changes the dimensionality of the input signal to a dimensionality that you specify, using the block's **Output dimensionality** parameter. For example, you can use the block to change an N-element vector to a 1-by-N or N-by-1 matrix signal, and vice versa.

The **Output dimensionality** parameter lets you select any of the following output options.

Output Dimensionality	Description
1-D array	Converts a multidimensional array to a vector (1-D array) array signal. The output vector consists of the first column of the input matrix followed by the second column, etc. (This option leaves a vector input unchanged.)
Column vector	Converts a vector, matrix, or multidimensional input signal to a column matrix, i.e., an M-by-1 matrix, where M is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.

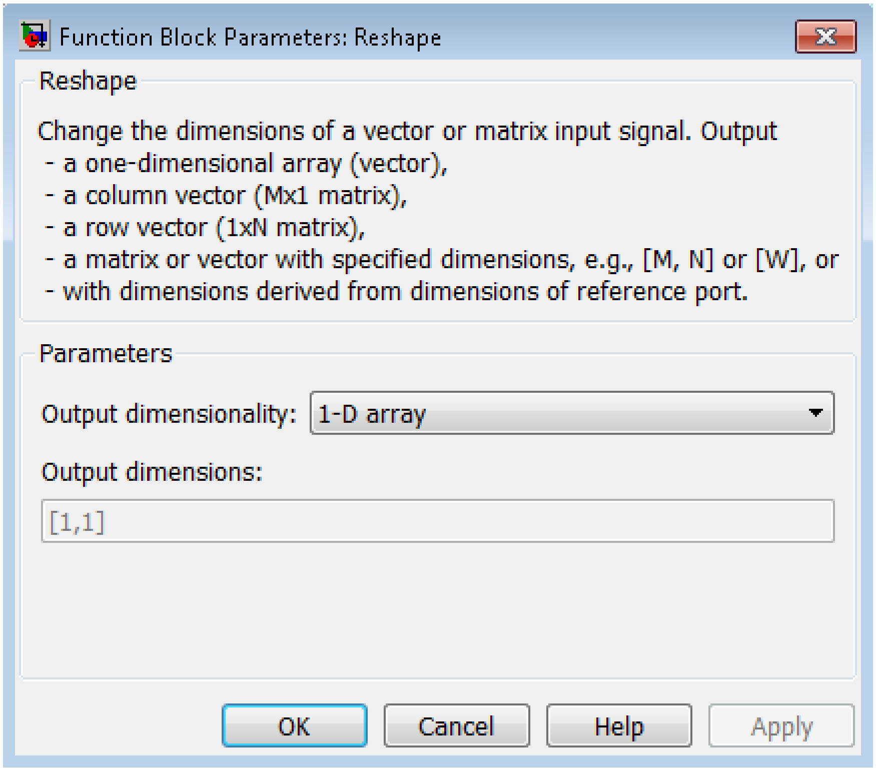
Output Dimensionality	**Description**
Row vector	Converts a vector, matrix, or multidimensional input signal to a row matrix, i.e., a 1-by-N matrix where N is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.
Customize	Converts the input signal to an output signal whose dimensions you specify, using the **Output dimensions** parameter. The value of the **Output dimensions** parameter can be a one- or multi-element vector. A value of `[N]` outputs a vector of size N. A value of `[M N]` outputs an M-by-N matrix. The number of elements of the input signal must match the number of elements specified by the **Output dimensions** parameter. For multidimensional arrays, the conversion is done along the first dimension.
Derive from reference input port	Creates a second input port, Ref, on the block. Derives the dimensions of the output signal from the dimensions of the signal input to the Ref input port. Selecting this option disables the **Output dimensions** parameter. When you select this parameter, the input signals for both inport ports, U and Ref, must have the same sampling mode (sample-based or frame-based).

Data Type Support

The Reshape block accepts and outputs signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

You can use an array of buses as an input signal to a Reshape block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.



Function Block Parameters: Reshape
Reshape
Parameters
OK
Cancel
Help
Apply

Change the dimensions of a vector or matrix input signal. Output
- a one-dimensional array (vector),
- a column vector (Mx1 matrix),
- a row vector (1xN matrix),
- a matrix or vector with specified dimensions, e.g., [M, N] or [W], or
- with dimensions derived from dimensions of reference port.

Output dimensionality: 1-D array

Output dimensions:

[1,1]

Parameters and Dialog Box

Output dimensionality

Specify the dimensionality of the output signal.

Output dimensions

Specify a custom output dimensionality. This parameter is available only when you set **Output dimensionality** to `Customize`.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block

Scalar Expansion	N/A
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

Apply rounding function to signal

Library

Math Operations

Description

The Rounding Function block applies a rounding function to the input signal to produce the output signal.

You can select one of the following rounding functions from the **Function** list:

- `floor`

  Rounds each element of the input signal to the nearest integer value towards minus infinity.

- `ceil`

  Rounds each element of the input signal to the nearest integer towards positive infinity.

- `round`

  Rounds each element of the input signal to the nearest integer.

- `fix`

  Rounds each element of the input signal to the nearest integer towards zero.

The name of the selected function appears on the block.

The input signal can be a scalar, vector, or matrix signal having real- or complex-valued elements of type `double`. The output signal has the same dimensions, data type, and numeric type as the input. Each element of the output signal is the result of applying the selected rounding function to the corresponding element of the input signal.

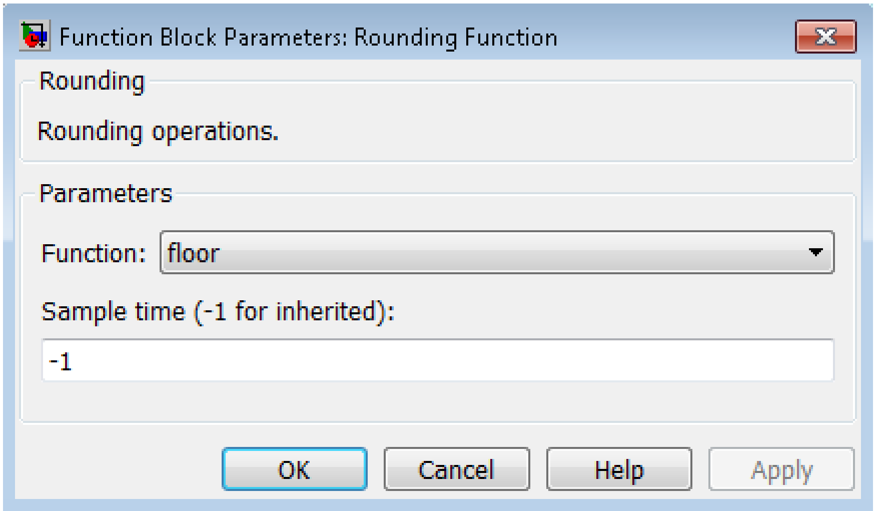
Tip Use the Rounding Function block instead of the Fcn block when you want vector or matrix output, because the Fcn block produces only scalar output.

Also, the Rounding Function block provides two more rounding modes. The Fcn block supports `floor` and `ceil`, but does not support `round` and `fix`.

Data Type Support

The Rounding Function block accepts and outputs real signals of type `double` or `single`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.


Function Block Parameters: Rounding Function
Rounding
Parameters
OK
Cancel
Help
Apply

Rounding operations.

Function: floor

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Function

Specify the rounding function.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	N/A
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	No

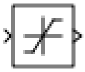
Purpose

Limit range of signal

Library

Discontinuities

Description



The Saturation block imposes upper and lower limits on an input signal.

When the input is...	Where...	The block output is the...
Within the range specified by the **Lower limit** and **Upper limit** parameters	Lower limit ≤ Input value ≤ Upper limit	Input value
Less than the **Lower limit** parameter	Input value < Lower limit	Lower limit
Greater than the **Upper limit** parameter	Input value > Upper limit	Upper limit

When the **Lower limit** and **Upper limit** parameters have the same value, the block output is that value.

Data Type Support

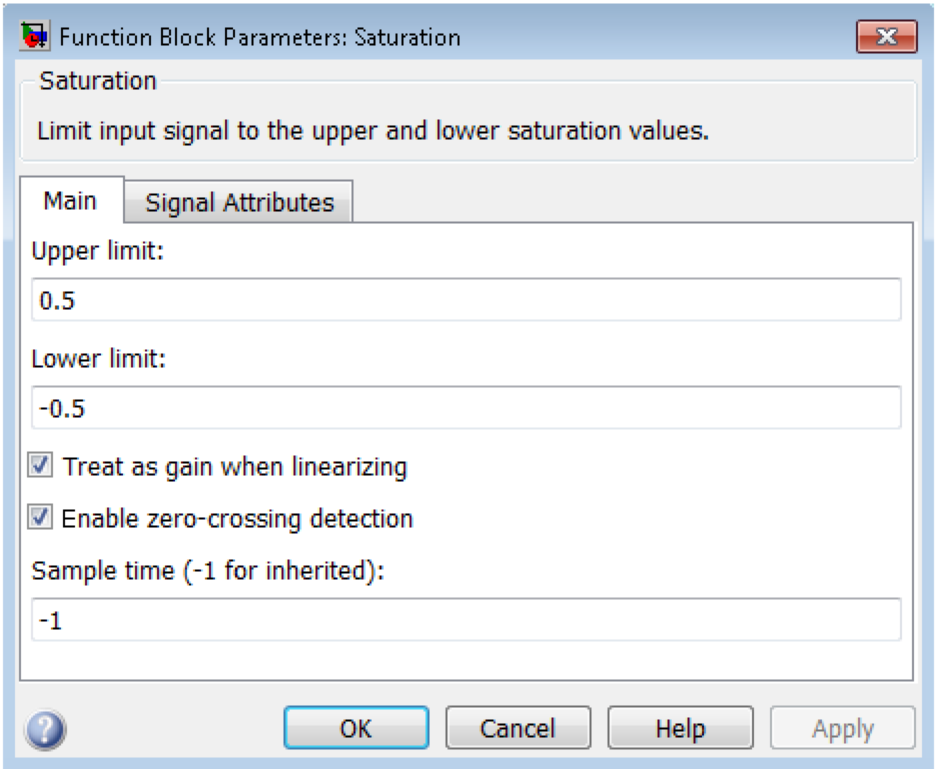
The Saturation block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Saturation block dialog box appears as follows:



Function Block Parameters: Saturation

Saturation

Limit input signal to the upper and lower saturation values.

Main
Signal Attributes

Upper limit: 0.5

Lower limit: -0.5

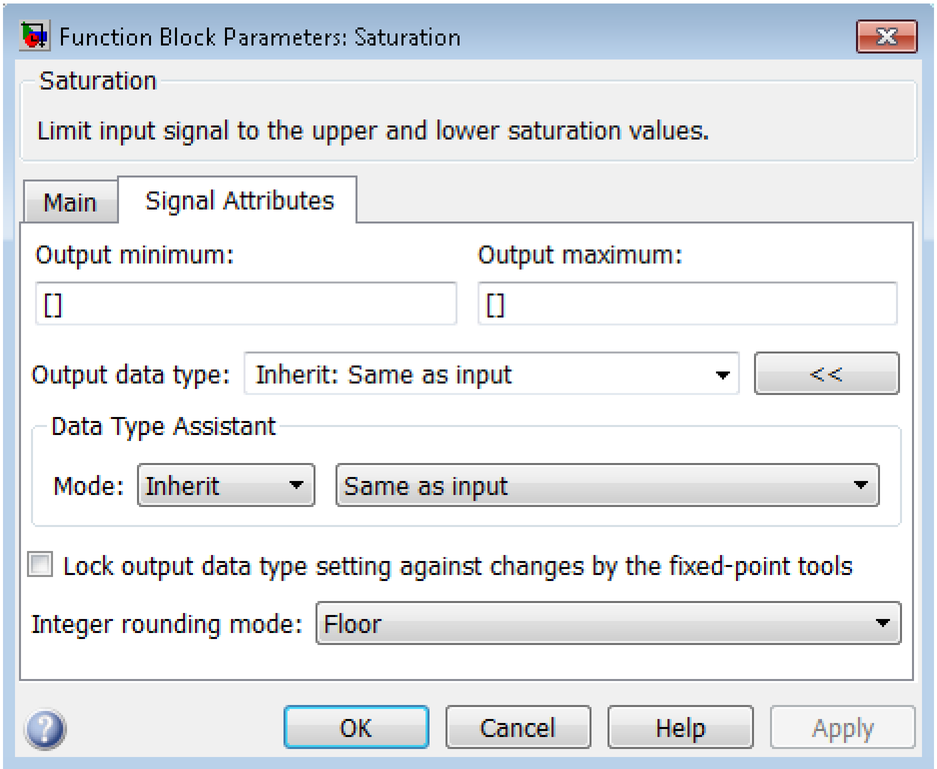
☑ Treat as gain when linearizing

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

OK
Cancel
Help
Apply

The **Signal Attributes** pane of the Saturation block dialog box appears as follows:



Function Block Parameters: Saturation

Saturation

Limit input signal to the upper and lower saturation values.

Output minimum: []

Output maximum: []

Output data type: Inherit: Same as input

Data Type Assistant

Mode: Inherit — Same as input

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Upper limit

Specify the upper bound on the input signal.

Settings

Default: `0.5`

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

Tip

- When the input signal to the Saturation block is above this value, the output of the block is clipped to this value.
- The **Upper limit** parameter is converted to the output data type offline using round-to-nearest and saturation.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Lower limit

Specify the lower bound on the input signal.

Settings

Default: `-0.5`

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

Tips

- When the input signal to the Saturation block is below this value, the output of the block is clipped to this value.
- The **Lower limit** parameter is converted to the output data type offline using round-to-nearest and saturation.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Treat as gain when linearizing

Select this parameter to cause the linearization commands to treat the gain as 1

Settings

Default: On

☑ On

Select to cause the linearization commands to treat the gain as 1.

☐ Off

Clear to cause the linearization commands to treat the gain as 0.

Tips

Linearization commands in Simulink software treat this block as a gain in state space.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Settings

Default: On

☑ On

Enable zero-crossing detection.

☐ Off

Do not enable zero-crossing detection.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
: Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`
: Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`
: Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`
: Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`
: Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`
: Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`
: Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the minimum to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Same as input`

`Inherit:  Inherit via back propagation`
Use data type of the driving block.

`Inherit:  Same as input`
Use data type of input signal.

`double`
Output data type is `double`.

`single`
Output data type is `single`.

`int8`
Output data type is `int8`.

`uint8`
Output data type is `uint8`.

`int16`
Output data type is `int16`.

`uint16`
Output data type is `uint16`.

`int32`
Output data type is `int32`.

`uint32`
Output data type is `uint32`.

`fixdt(1,16,0)`
Output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
Output data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specify Block Output Data Types".

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via back propagation`
- `Same as input` (default)

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
Specify the fixed-point data as signed.

`Unsigned`
Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`

`Binary point`
: Specify binary point location.

`Slope and bias`
: Enter slope and bias.

`Best precision`
: Specify best-precision values.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

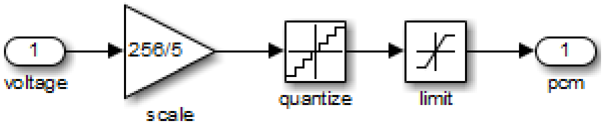
See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Examples

The `sldemo_boiler` model shows how you can use the Saturation block.

The Saturation block appears in the `Boiler Plant model/digital thermometer/ADC` subsystem.



The ADC subsystem digitizes the input analog voltage by:

- Multiplying the analog voltage by 256/5 with the Gain block
- Rounding the value to integer floor with the Quantizer block
- Limiting the output to a maximum of 255 (the largest unsigned 8-bit integer value) with the Saturation block

For more information, see “Explore the Fixed-Point "Bang-Bang Control" Model” in the Stateflow documentation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of parameters and input
Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled.

See Also

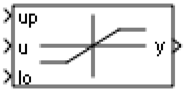
Saturation Dynamic

Saturation Dynamic

Purpose Bound range of input

Library Discontinuities

Description



The Saturation Dynamic block bounds the range of an input signal to upper and lower saturation values. An input signal outside of these limits saturates to one of the bounds where:

- The input below the lower limit is set to the lower limit.
- The input above the upper limit is set to the upper limit.

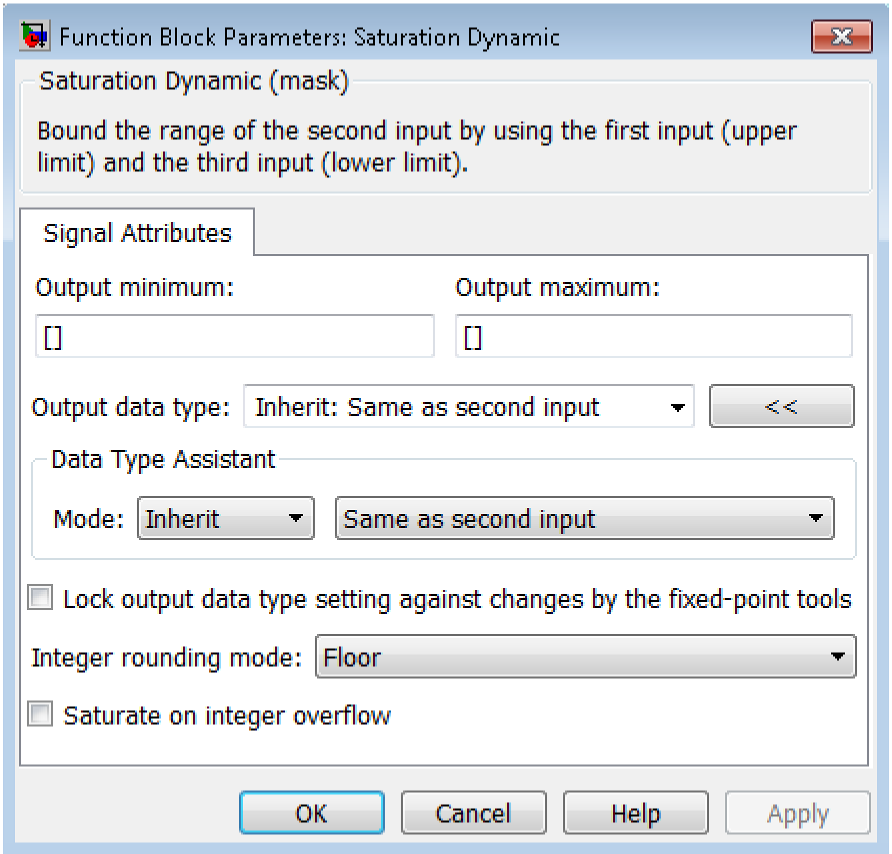
The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

Data Type Support The Saturation Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

Tip Although this block accepts Boolean signals, avoid this usage.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Function Block Parameters: Saturation Dynamic

Saturation Dynamic (mask)

Bound the range of the second input by using the first input (upper limit) and the third input (lower limit).

Signal Attributes

Output minimum: []

Output maximum: []

Output data type: Inherit: Same as second input

<<

Data Type Assistant

Mode: Inherit — Same as second input

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate on integer overflow

OK
Cancel
Help
Apply

Parameters and Dialog Box

Output minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as second input`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Block Output Data Types” in the *Simulink User’s Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 “Checking 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.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also Saturation

Scope and Floating Scope

Purpose

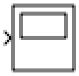
Display signals generated during simulation

Library

Sinks

Description

The Scope block displays inputs signals with respect to simulation time.



- **Scope window** — If a Scope window is closed at the start of a simulation, scope data is still written to the connected Scope. As a result, if you open a Scope after a simulation, the Scope window displays the input signal or signals.
- **Plotting signals** — If the input signal is continuous, the Scope draws a point-to-point plot. If the signal is discrete, the Scope draws a stair-step plot.
- **Time step values** — The Scope block only displays major time step values. The scope displays additional interpolated points between major time steps if specified by the refine parameter.
- **Multiple y-axes (graphs)** — A Scope window can display multiple y-axes (graphs) with one graph per input port. All of the y-axes have a common time range on the x-axis. The Scope block allows you to adjust the amount of time and the range of input values displayed. You can modify the Scope parameter values during a simulation.
- **Data type support** — The Scope block accepts real (not complex) signals of any data type that Simulink supports, including fixed-point and enumerated data types. The Scope block also accepts homogeneous vectors. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Note A Scope block is not the same as a Scope Viewer. For information about Scope Viewers, see “Signal Viewer Tasks”

Note If you want to display internal data from a library block that you create, do not use a Scope block inside the library block. Instead, provide your library block with output ports, and then connect the ports to a Scope block external to the library block.

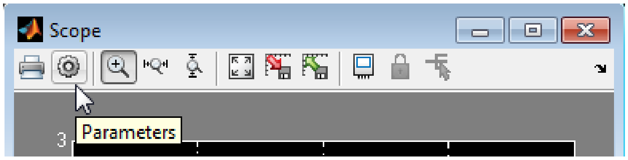
Scope Parameters

Use scope parameters to change axis (graph) settings, data logging options, and figure color and style.

Note Changing Scope dialog parameters using `set_param` may put the Scope in an undefined state.

---

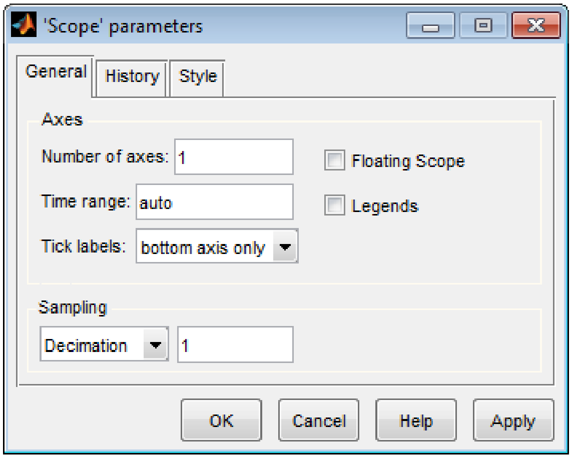
1. On the **Scope** window toolbar, click the Parameters button



2. Edit the scope parameters.
3. Click the **Apply** or **OK** button. The values you entered or selected apply to the next simulation.

General Pane

Set the axis parameters, time range, tick labels, and decimation or sample time.



Axes

Number of axes: 1

Time range: auto

Tick labels: bottom axis only

☐ Floating Scope

☐ Legends

Sampling

Decimation 1

Number of axes

Number of y-axes (graphs) and input ports on the Scope block. With the exception of the Floating Scope, there is no limit to the number of y-axes a Scope window can contain. All axes share the same time base (x-axis), but have independent y-axes.

Time range

Range of x-axis in Scope window. Do not enter a variable name in this field. Enter either,

- A number — Each screen displays an amount of time data corresponding to the length of the range
- `auto` — Sets the x-axis to the duration of the simulation

Tick labels

Specify how to display labels on the tick marks. The options are:

- `all` — Place tick labels on the outside of the x-axes and y-axes
- `inside` — Place tick labels inside all axes (available only on scope viewers)
- `bottom-axis only` — Place tick labels only on the x-axes
- `none` — Do not label ticks

Sampling
Specify the number of data points to plot. Select either,

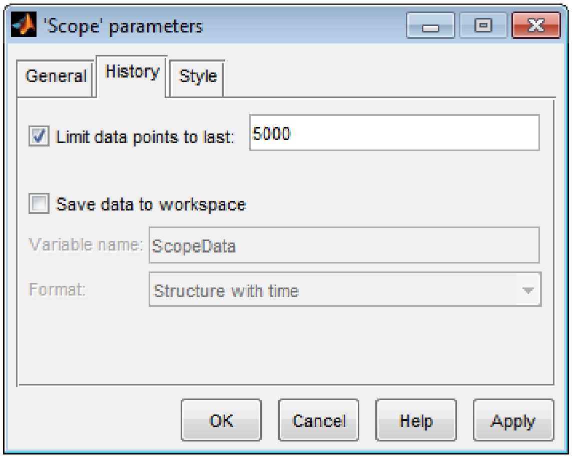
- `Decimation` — Plot signal data for every `N`th time step. Entering `1` plots every time step.
- `Sample time` — Plot signal data for each time step at a specified time interval. This parameter is useful when you are using a variable-step solver where the interval between time steps might not be the same.

After you make a selection, enter a number in the data field.

Floating scope
Changes a Scope block into a Floating Scope block.

Legends
Displays a legend of signal names with color and style.

History Pane



☑ Limit data points to last: 5000

☐ Save data to workspace

Variable name: ScopeData

Format: Structure with time

Use this pane to control the amount of data a Scope collects during a simulation and where to save the data.

Limit data points to last

Specify the number of data points a Scope collects. If you select this option, enter a value in the data field. The Scope relies on its data history for zooming and auto-scaling operations. If the number of data points is limited to 1,000 and the simulation generates 2,000 data points, only the last 1,000 are available for regenerating the display.

Note If you do not select the **Limit data points to last** check box, the memory consumed by MATLAB steadily increases as the simulation stores the entire simulation history. The result may be an out of memory error.

For simulations with a small time step, check the memory in your system. If the available memory is limited, select `Decimation` from the **Sampling** drop down list and enter a value greater than `1`. This reduces the number of points a simulation stores.

For simulations with the **Stop time** set to `inf`, always select the **Limit data points to last** check box and enter a value.

Save data to workspace

Select this option to save collected data to the MATLAB workspace. If you select this option, the **Variable name** and **Format** fields become active. Simulink logs fixed-point data as `double`.

Note Data logging for a Floating Scope is not available. When using a Floating Scope, **Save data to workspace** is disabled.

Variable name

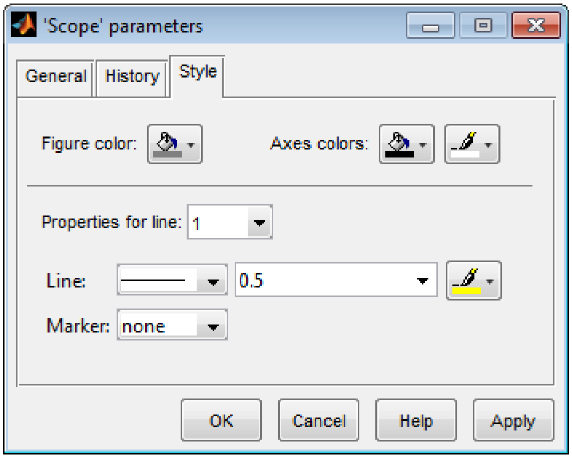
Enter a valid MATLAB variable name. The name must be unique among all data logging variables in your model. Other data logging variables include those defined on other Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs.

Format

Select one of three formats to save data:

- `Array` — Use for a Scope with one set of axes.
- `Structure` — Use for Scopes with more than one set of axes and when you do not want to save time data.
- `Structure with time` — Use when you want to save time data.

Style Pane



'Scope' parameters

General | History | Style

Figure color:

Axes colors:

Properties for line: 1

Line: 0.5

Marker: none

Figure color

Select the background color for the Scope window.

Axes colors

Select the background color for axes (graphs) with the first color pallet. Select the grid and label color with the second pallet.

Properties for line

For graphs with more than one line, select a line number.

Line

Select line style, width, and color.

The Scope block displays one input per axes. When displaying a vector or matrix of signals on the same axis, the Scope block assigns colors to each signal, in this order: yellow, magenta, cyan, red, green, and dark blue. If there are more than six signals, the Scope block recycles through the colors a second time.

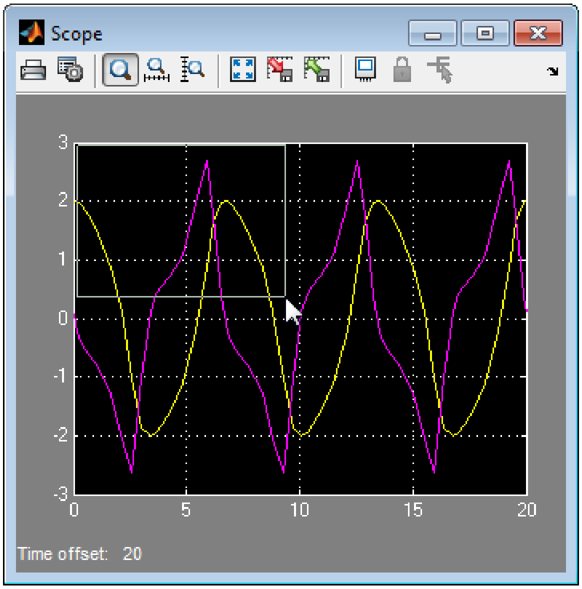
Marker

Select the marker style.

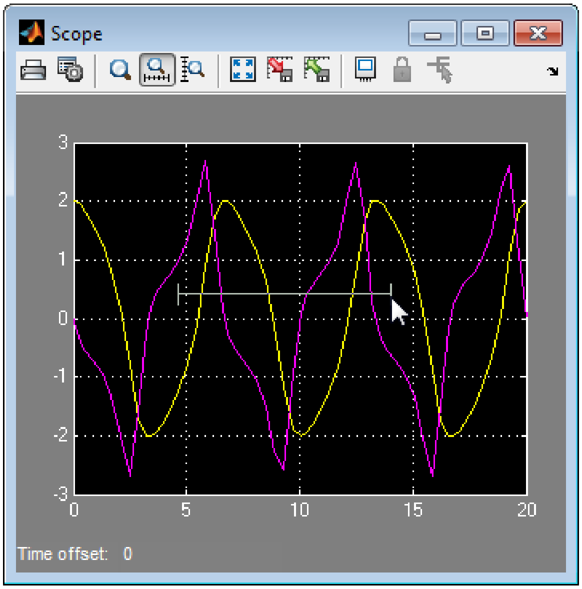
Scope Zooming

To zoom in on data in both the x and y directions, select the left most **Zoom** toolbar button, and then define the zoom region with a bounded box. When you release the mouse button, the Scope displays the data in that area. You can also click a point in the area you want to zoom in on.

This figure shows a zoomed region before you release the mouse button.



To zoom in on data in just the *x* direction, click the middle **Zoom** toolbar button. Define the zoom region by positioning the pointer at one end of the region, pressing and holding down the mouse button, then moving the pointer to the other end of the region. This figure shows the Scope after you define the zoom region, but before you release the mouse button.



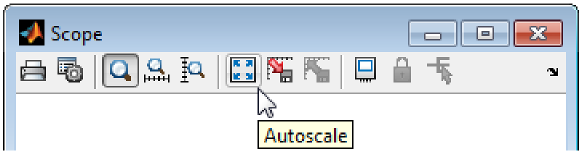
When you release the mouse button, the Scope displays the magnified region.

Zooming in the *y* direction works in the same way, except that you click the right most **Zoom** toolbar button before defining the zoom region.

Note Since Floating Scope does not buffer the data it displays, you cannot zoom the contents. Simulink disables (grays) the **Zoom** button on the toolbar of a Floating Scope.

Autoscaling the Scope Axes

User the **Autoscale** button to view the minimum and maximum points of your signals.

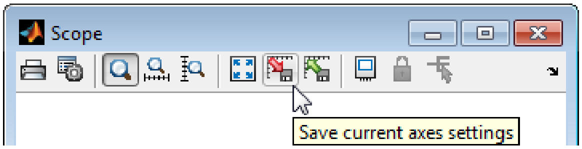


If you click the **Autoscale** button while a simulation is running, the axes are autoscaled based on the data that appears on the current screen, and the autoscale limits are saved as the defaults. Then next simulation you run also uses the new axes limits.

Note Since the Floating Scope does not buffer the data it displays, you cannot autoscale the contents. Simulink disables the **Autoscale** button on the toolbar of a Floating Scope.

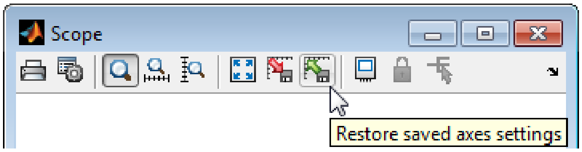
Saving and Restoring the Axes Settings

Use the **Save current axes setting** toolbar button to store the current *x-axis* and *y-axis* settings.



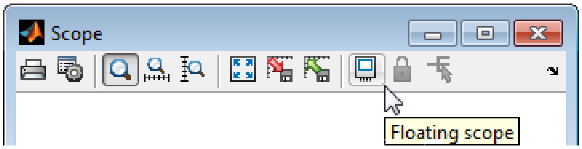
If you select the Save current axes settings button, the block specifies its current y-limits as the values of the **Y-min** and **Y-max** parameters (see “Y-Axis Limits” on page 1-1469. Similarly, the block specifies its current x-axis limits as the value of the **Time range** parameter (see “General Pane” on page 1-1460).

Use the **Restore saved axes settings** button to restore the saved axis settings.



Floating Scope

You can change a Scope block to a Floating Scope block from the toobar.

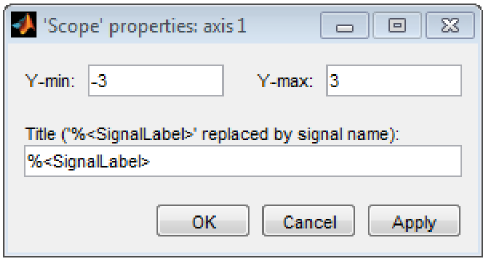


To add signals to a Floating Scope, click the Signal Selector button, and then select signals from your model. For more information, see "Signal Selector".

Tip Before using a Floating Scope in a simulation, clear the **Signal storage reuse** check box in the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box. For more information, see "Optimization Pane: Signals and Parameters".

Y-Axis Limits

Set *y*-axis limits by right-clicking an axis (graph) and selecting **Axes Properties**. The following dialog box opens.



Y-min: -3
Y-max: 3
Title ('%<SignalLabel>' replaced by signal name):
%<SignalLabel>

Y-min

Enter the minimum value for the *y*-axis.

Y-max

Enter the maximum value for the *y*-axis.

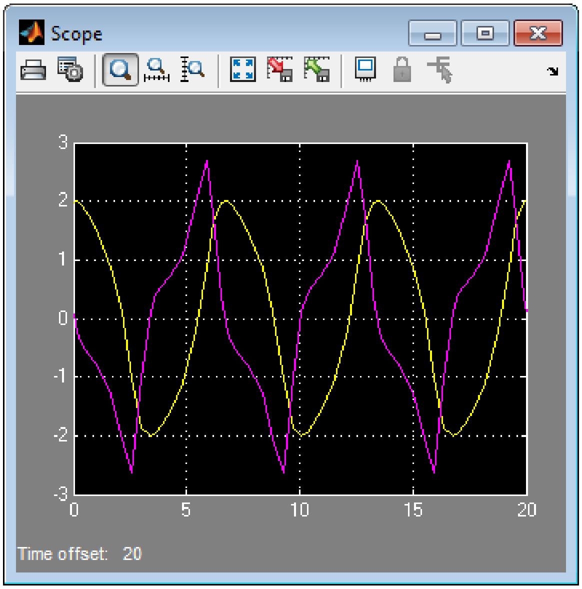
Title

Enter the title of the plot. You can include a signal label in the title by typing `%<SignalLabel>` as part of the title string (`%<SignalLabel>` is replaced by the signal label).

Note You cannot add a title to a Floating Scope window.

Time Offset

This figure shows the Scope block displaying the output of the `vdp` model. The simulation runs for `40` seconds. This scope shows the final `20` seconds of the simulation. The **Time offset** field displays the time corresponding to `0` on the horizontal axis. Therefore, you have to add the offset to the fixed time range values on the *x*-axis to get the actual time.



Creating a MATLAB Figure from Scope Block Data

If you need more drawing tools than provided with the Scope window, you can save scope data to the MATLAB workspace, replot the scope data in a Figure window, and then use the Plotting Tools.

1. On the Scope window toolbar, click the Parameters button, and then select the **History** tab.

2. Select the **Save data to workspace** check box, and in the **Variable name** field, enter a valid MATLAB variable name. For example, enter

   ```
   ScopeData
   ```

3. From the **Format** list, select `Structure with time`. Click **OK** or **Apply**.

4. Run a simulation.

5 In the MATLAB Command Window, type:

```
simplot(ScopeData)
```

A Figure window opens with a plot of the scope data.

6 Edit the plot. See “Plots and Plotting Tools” in the MATLAB documentation.

Characteristics

Sample Time	Inherited from driving block or you can set
States	0
Multidimensionalized	Yes

Second-Order Integrator, Second-Order Integrator Limited

Purpose

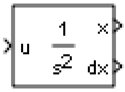
Integrate input signal twice

Library

Continuous

Description

The Second-Order Integrator block and the Second-Order Integrator Limited block solve the second-order initial value problem:



$$\frac{d^2x}{dt^2} = u,$$

$$\left.\frac{dx}{dt}\right|_{t=0} = dx_o,$$

$$\left.x\right|_{t=0} = x_o,$$

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. In order to satisfy this relationship throughout the simulation, Simulink places various constraints on the block parameters and behavior.

The Second-Order Itegrator 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 1-1475.

Simulink software can use a number of different numerical integration methods to compute the outputs of the block. Each has advantages for specific applications. Use the **Solver** pane of the Configuration Parameters dialog box to select the technique best suited to your application. (For more information, see “Choose a Solver”.) 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:

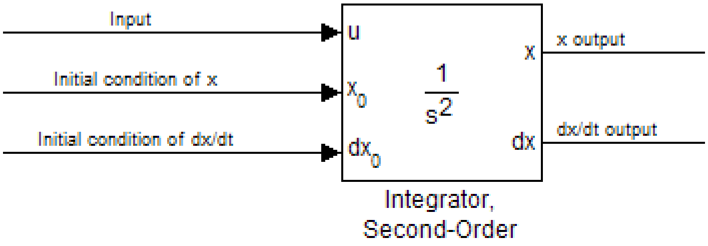
- Specify whether the source of each state initial condition is internal or external
- Specify a value for the state initial conditions
- Define upper and lower limits on either or both states
- Specify absolute tolerances for each state
- Specify names for both states
- Choose an external reset condition
- 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.



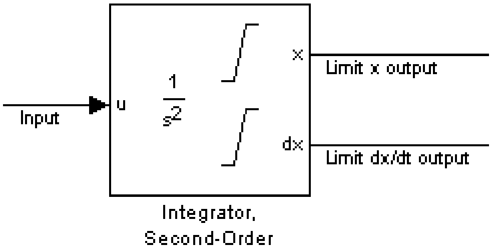
Note

- 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 1-1475) and one or more initial conditions are outside the corresponding limits, then the respective states are initialized to the closest valid value and a set of consistent initial conditions is calculated.

Limiting the States

When modeling a second-order system, you may need to limit the block states. For example, the motion of a piston within a cylinder is governed by Newton's Second Law and has constraints on the piston position (*x*). With the Second-Order Integrator block, you can limit the states *x* and *dx/dt* independent of each other. You can even change the limits during simulation; however, you cannot change whether or not the states are limited. An important rule to follow is that an upper limit must be strictly greater than its corresponding lower limit.

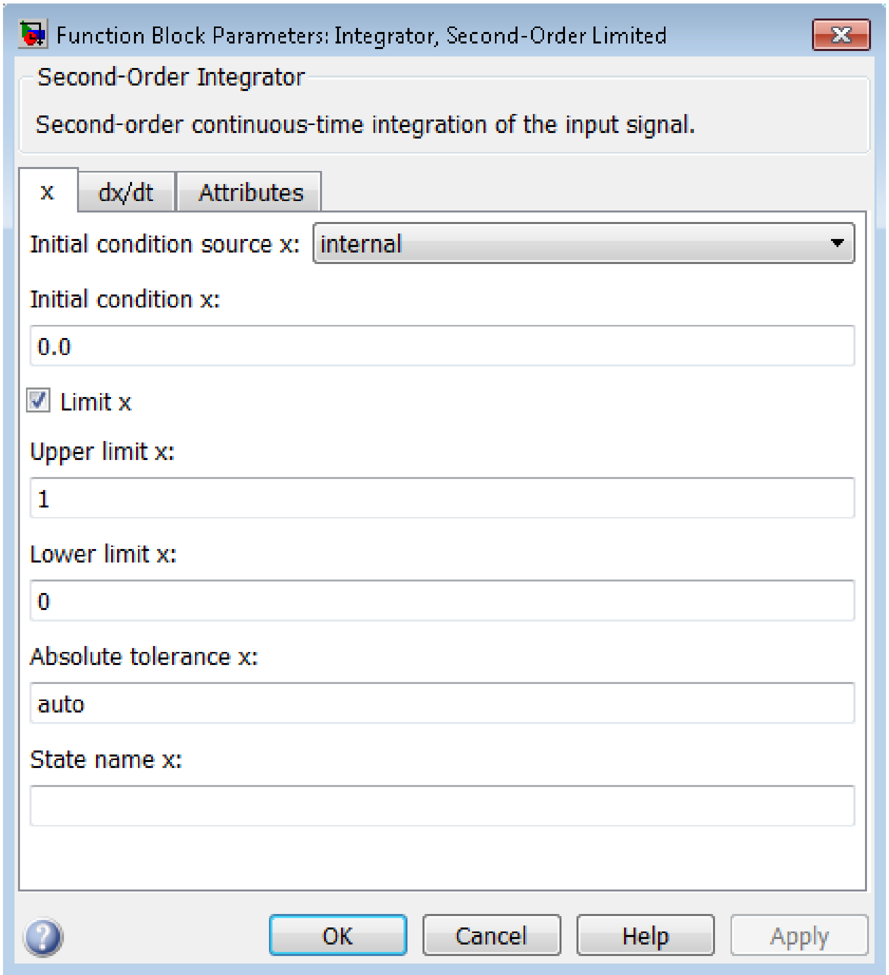
The block appearance changes when you limit one or both states. Shown below is the appearance of the block with both states limited.


Input
u
1
s2
x
Limit x output
dx
Limit dx/dt output
Integrator,
Second-Order

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:

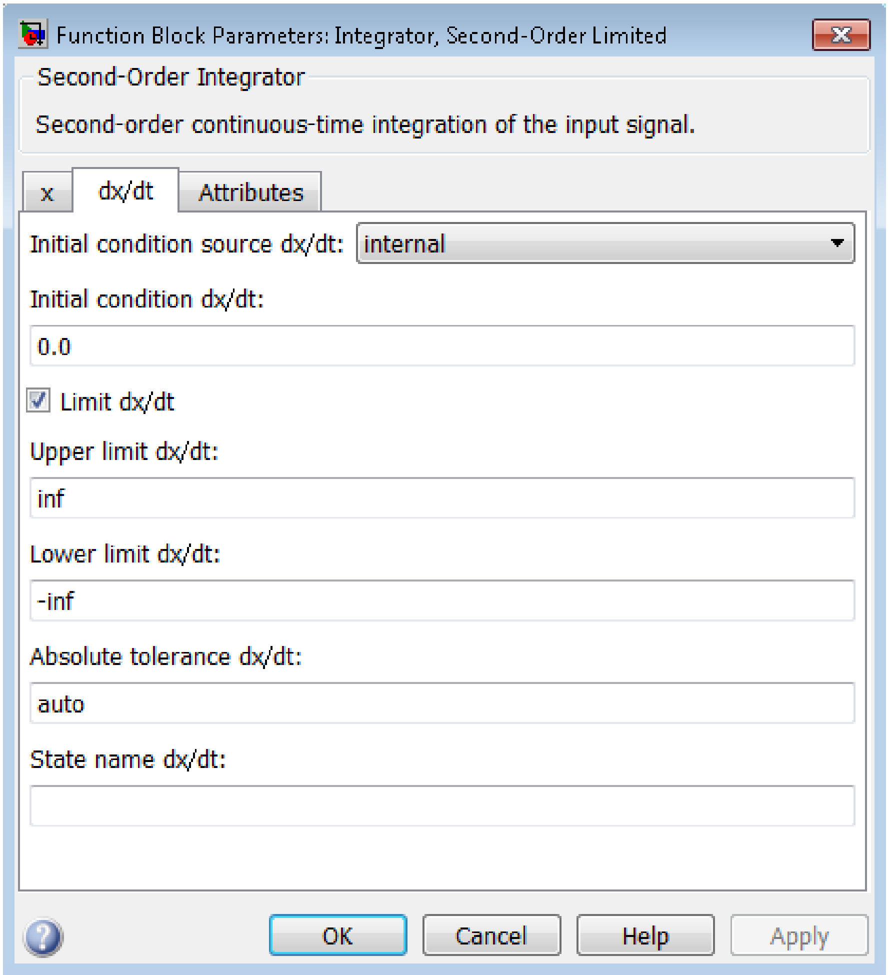
- When x is less than or equal to its lower limit, the value of x is held at its lower limit and dx/dt is set to zero.

- 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 1-1482

Limiting dx/dt only

As with state *x*, state *dx/dt* is set as limited by default on the **dx/dt** pane of the Second-Order Integrator Limited parameter dialog box. You can manually set this parameter, **Limit dx/dt**, on the Second-Order Integrator block. In either case, you must enter the appropriate limits for *dx/dt*.


Function Block Parameters: Integrator, Second-Order Limited
x
dx/dt
Attributes
OK
Cancel
Help
Apply

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:

$$\begin{aligned}&\frac{dx}{dt} = L,\\ &x(t_L) = x_{L,}\end{aligned}$$

where L is the dx/dt limit (upper or lower), t_L is the time when dx/dt reaches this limit, and x_L is the value of state x at that time.

Limiting Both States

When you limit both states, Simulink maintains mathematical consistency of the states by limiting the allowable values of the upper and lower limits for dx/dt. Such limitations are necessary to satisfy the following constraints:

- When x is at its saturation limits, the value of dx/dt must be zero.
- In order for x to leave the upper limit, the value of dx/dt must be strictly negative.
- In order for x to leave its lower limit, the value of dx/dt must be strictly positive.

Consequently, for such cases, the upper limit of dx/dt must be strictly positive and the lower limit of dx/dt must be strictly negative.

When both states are limited, the block determines the states as follows:

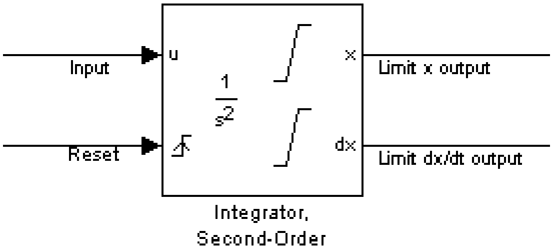
- Whenever x reaches its limits, the resulting behavior is the same as that described in “Limiting x only”.

- Whenever *dx/dt* reaches one of its limits, the resulting behavior is the same as that described in "Limiting dx/dt only" — including the computation of *x* using a first-order ODE when *dx/dt* is held at one of its limits. In such cases, when *x* reaches one of its limits, it is held at that limit and *dx/dt* is set to zero.
- Whenever both reach their respective limits simultaneously, the state *x* behavior overrides *dx/dt* behavior to maintain consistency of the states.

When you limit both states, you can choose to reinitialize *dx/dt* at the time when state *x* reaches saturation. If the reinitialized value is outside specified limits on *dx/dt*, then *dx/dt* is reinitialized to the closest valid value and a consistent set of initial conditions is calculated. See "Reinitializing *dx/dt* when *x* reaches saturation" on page 1-1482

Resetting the State

The block can reset its states to the specified initial conditions based on an external signal. To cause the block to reset its states, select one of the **External reset** choices on the **Attributes** pane. A trigger port appears on the block below its input port and indicates the trigger type.



- Select `rising` to reset the states when the reset signal rises from zero to a positive value, from a negative to a positive value, or a negative value to zero.

- Select `falling` to reset the states when the reset signal falls from a positive value to zero, from a positive to a negative value, or from zero to negative.
- Select `either` to reset the states when the reset signal changes from zero to a nonzero value or changes sign.

The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results (see "Algebraic Loops").

Enabling Zero-Crossing Detection

This parameter controls whether zero-crossing detection is enabled for this block. By default, the **Enable zero-crossing detection** parameter is selected on the **Attributes** pane. However, this parameter is only in affect if the **Zero-crossing control**, on the **Solver** pane of the Configuration Parameters dialog, is set to `Use local settings`. For more information, see "Zero-Crossing Detection".

Reinitializing *dx/dt* when *x* reaches saturation

For certain modeling applications, dx/dt must be reinitialized when state x reaches its limits in order to pull x out of saturation immediately. You can achieve this by selecting **Reinitialize dx/dt when x reaches saturation** on the **Attributes** pane.

If this option is on, then at the instant when x reaches saturation, Simulink checks whether the current value of the dx/dt initial condition (parameter or signal) allows the state x to leave saturation immediately. If so, Simulink reinitializes state dx/dt with the value of the initial condition (parameter or signal) at that instant. If not, Simulink ignores this parameter at the current instant and sets dx/dt to zero to make the block states consistent.

This parameter only applies at the time when x actually reaches saturation limit. It does not apply at any future time when x is being held at saturation.

Refer to the sections on limiting the states for more information. For an example, see the `sldemo_bounce` example.

Disregarding State Limits and External Reset for Linearization

For cases where you simplify your model by linearizing it, you can have Simulink disregard the limits of the states and the external reset by selecting **Ignore state limits and the reset for linearization**.

Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see “Specifying Error Tolerances for Variable-Step Solvers”) to compute the output of the integrator blocks. If this value does not provide sufficient error control, specify a more appropriate value for state *x* in the **Absolute tolerance x** field and for state *dx/dt* in the **Absolute tolerance dx/dt** field of the parameter dialog box. Simulink uses the values that you specify to compute the state values of the block.

Specifying the Display of the Output Ports

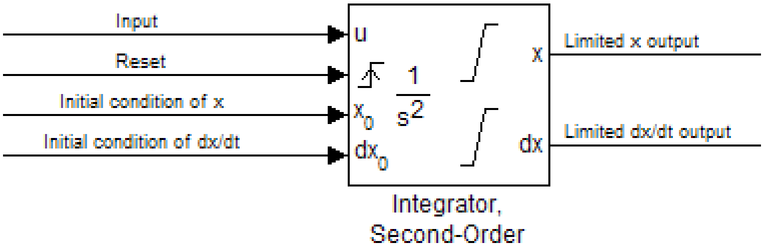
You can control whether or not to display the *x* or the *dx/dt* output port using the `ShowOutput` parameter. You can display one output port or both; however, you must select at least one.

Specifying the State Names

You can specify the name of *x* states and *dx/dt* states using the `StateNameX` and `StateNameDXDT` parameters. However, you must specify names for either both or neither; you cannot specify names for just *x* or just *dx/dt*. Both state names must have identical type and length. Furthermore, the number of names must evenly divide the number of states.

Selecting All Options

When you select all options, the block icon looks like this.


Input
Reset
Initial condition of x
Initial condition of dx/dt
u
x0
dx0
1/s2
x
dx
Limited x output
Limited dx/dt output
Integrator,
Second-Order

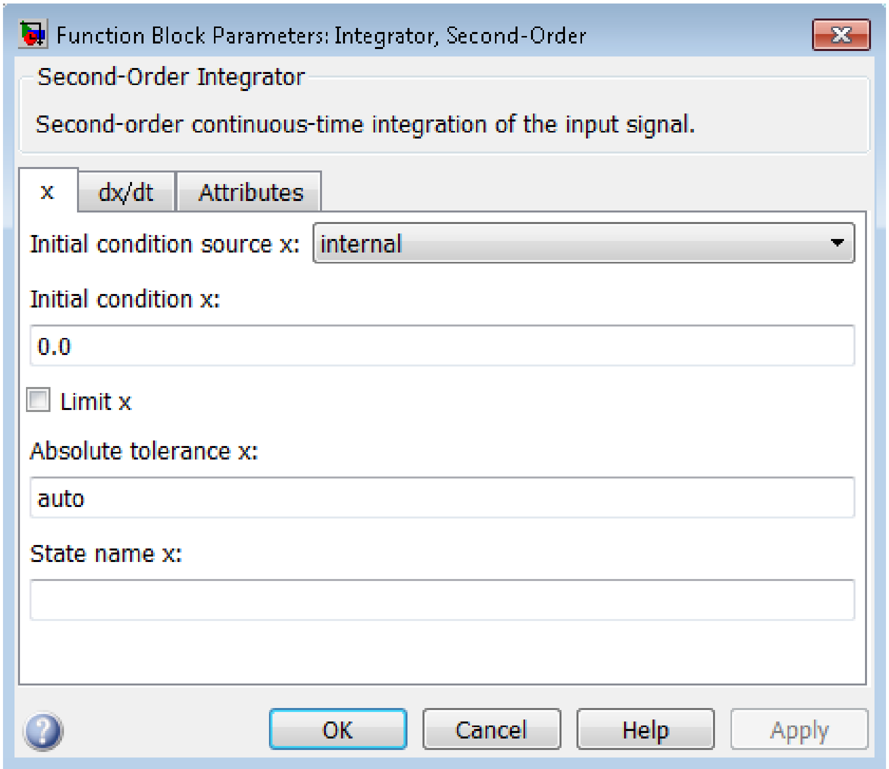
Data Type Support

The Integrator block accepts and outputs signals of type `double` on its data ports. The external reset port accepts signals of type `double` or `Boolean`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **x** pane of the Second-Order Integrator block dialog box appears as follows:


Function Block Parameters: Integrator, Second-Order
Second-Order Integrator
x
dx/dt
Attributes
OK
Cancel
Help
Apply

Second-order continuous-time integration of the input signal.

Initial condition source x: internal

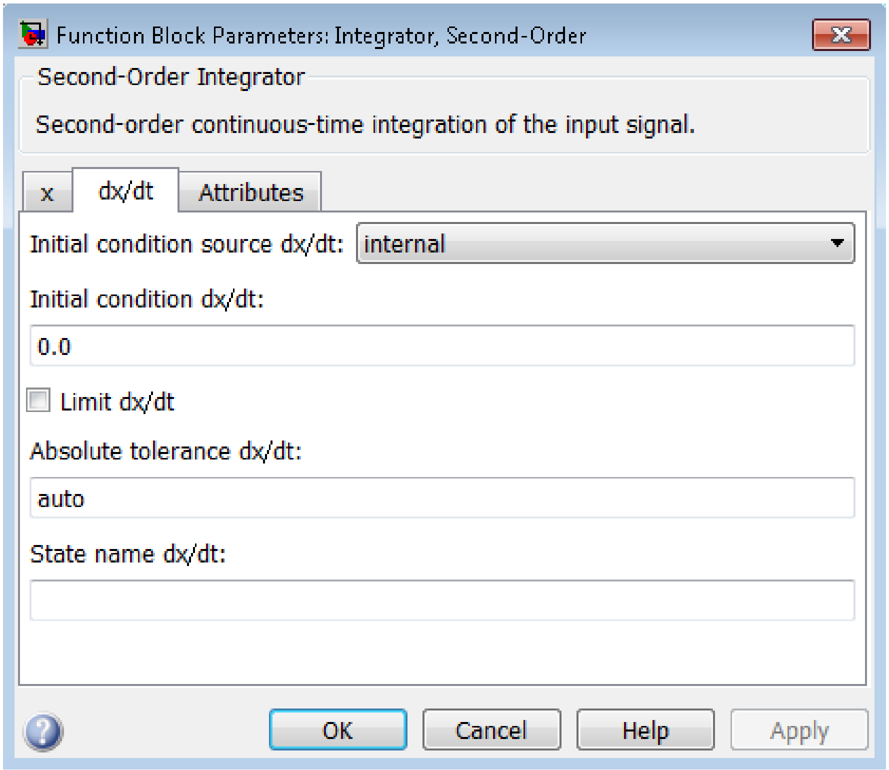
Initial condition x: 0.0

☐ Limit x

Absolute tolerance x: auto

State name x:

The **dx/dt** pane of the Second-Order Integrator block dialog box appears as follows:


Function Block Parameters: Integrator, Second-Order
x
dx/dt
Attributes
OK
Cancel
Help
Apply

Second-Order Integrator

Second-order continuous-time integration of the input signal.

Initial condition source dx/dt: internal

Initial condition dx/dt:

0.0

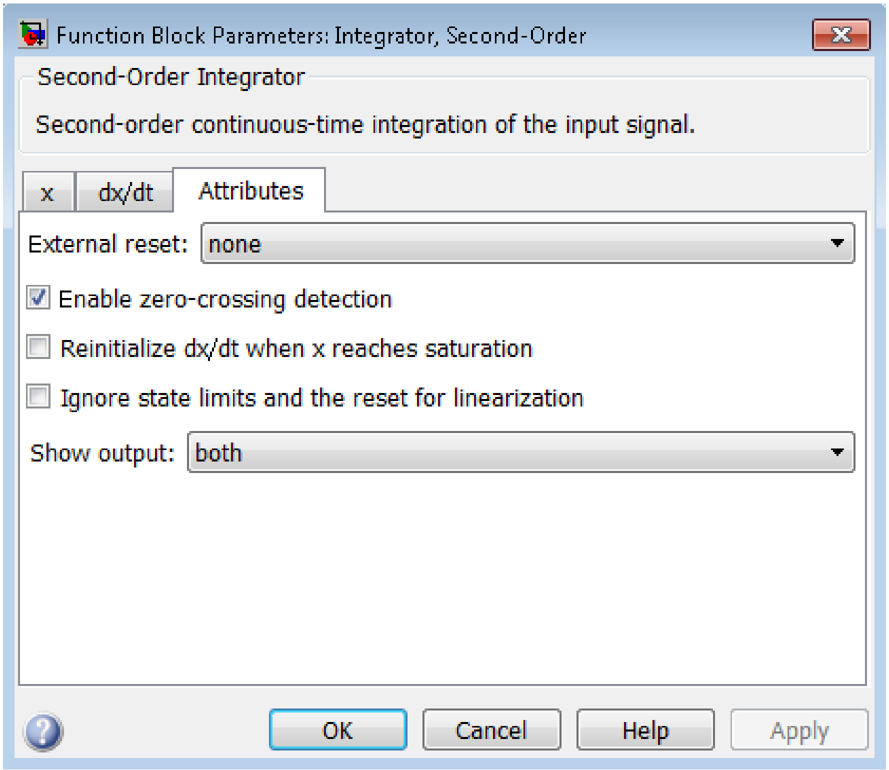
☐ Limit dx/dt

Absolute tolerance dx/dt:

auto

State name dx/dt:

The **Attributes** pane of the Second-Order Integrator block dialog box appears as follows:


Function Block Parameters: Integrator, Second-Order
x
dx/dt
Attributes
OK
Cancel
Help
Apply

Second-Order Integrator

Second-order continuous-time integration of the input signal.

External reset: none

☑ Enable zero-crossing detection

☐ Reinitialize dx/dt when x reaches saturation

☐ Ignore state limits and the reset for linearization

Show output: both

- “Initial condition source x” on page 1-1489
- “Initial condition x” on page 1-1490
- “Limit x” on page 1-1491
- “Upper limit x” on page 1-1492
- “Lower limit x” on page 1-1493
- “Absolute tolerance x” on page 1-1494
- “State name x” on page 1-1495
- “Initial condition source dx/dt” on page 1-1496

- “Initial condition dx/dt” on page 1-1497
- “Limit dx/dt” on page 1-1498
- “Upper limit dx/dt” on page 1-1499
- “Lower limit dx/dt” on page 1-1500
- “Absolute tolerance dx/dt” on page 1-1501
- “State name dx/dt” on page 1-1502
- “External reset” on page 1-1503
- “Enable zero-crossing detection” on page 1-1718
- “Reinitialize dx/dt when x reaches saturation” on page 1-1505
- “Ignore state limits and the reset for linearization” on page 1-1506
- “Show output” on page 1-1507

Initial condition source x

Specify the initial condition source for state x.

Settings

Default: `internal`

`internal`
: Get the initial conditions of state *x* from the **Initial condition x** parameter.

`external`
: Get the initial conditions of state *x* from an external block.

Tip

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.

Selecting `external` disables the **Initial condition x** parameter..

Command-Line Information

Parameter: `ICSourceX`
Type: string
Value: `'internal'` | `'external'`
Default: `'internal'`

Initial condition x

Specify the initial condition of state *x*.

Settings

Default: 0.0

Tip

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Dependencies

Setting **Initial condition source x** to `internal` enables this parameter.

Setting **Initial condition source x** to `external` disables this parameter.

Command-Line Information

Parameter: `ICX`
Type: vector or scalar
Value: `any valid real value`
Default: `0.0`

Limit x

Limit state *x* of the block to a value between the **Lower limit x** and **Upper limit x** parameters.

Settings

Default: `Off` for Second-Order Integrator, `On` for Second-Order Integrator Limited

☑ On

Limit state *x* to a value between the **Lower limit x** and **Upper limit x** parameters.

☐ Off

Do not limit the state x output to a value between the **Lower limit x** and **Upper limit x** parameters.

Dependencies

This parameter enables **Upper limit x** parameter.

This parameter enables **Lower limit x** parameter.

Command-Line Information

Parameter: `LimitX`
Type: string
Value: `'on' | 'off'`
Default: `'off' | 'on'`

Upper limit x

Specify the upper limit of state *x*.

Settings

Default: `inf` for Second-Order Integrator, `1` for Second-Order Integrator Limited

Tip

The upper saturation limit for state *x* must be strictly greater than the lower saturation limit.

Dependency

Limit x enables this parameter.

Command-Line Information

Parameter: `UpperLimitX`
Type: scalar or vector
Value: `any valid real value`
Default: `'inf' | 1`

Lower limit x

Specify the lower limit of state *x*.

Settings

Default: `inf` for Second-Order Integrator, `0` for Second-Order Integrator Limited

Tip

The lower saturation limit for state *x* must be strictly less than the upper saturation limit.

Dependencies

Limit x enables this parameter.

Command-Line Information

Parameter: `LowerLimitX`
Type: scalar or vector
Value: `any valid value`
Default: `'-inf'` | 0

Absolute tolerance x

Specify the absolute tolerance for computing state *x*.

Settings

Default: auto

- You can enter `auto`, –1, a real scalar or a real 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.

Command-Line Information

Parameter: `AbsoluteToleranceX`
Type: string, scalar, or vector with same dimension as x
Value: `any positive real scalar or vector | {'auto'} | {-1}`
Default: `'auto'`

State name x

Assign a unique name to state *x*.

Settings

Default: `''`

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 string or a cell array.

Command-Line Information

Parameter: `StateNameX`
Type: string
Value: `any valid expression`
Default: `''`

Initial condition source dx/dt

Specify the initial condition source for state *dx/dt*.

Settings

Default: `internal`

`internal`
: Get the initial conditions of state *dx/dt* from the **Initial condition dx/dt** parameter.

`external`
: Get the initial conditions of state *dx/dt* from an external block.

Tip

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.

Selecting `external` disables the **Initial condition dx/dt** parameter.

Command-Line Information

Parameter: `ICSourceDXDT`
Type: string
Value: `'internal' | 'external'`
Default: `'internal'`

Initial condition dx/dt

Specify the initial condition of state *dx/dt*.

Settings

Default: 0.0

Tip

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.

Command-Line Information

Parameter: `ICDXDT`
Type: scalar or vector
Value: `any valid real value`
Default: `0.0`

Limit dx/dt

Limit the dx/dt state of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters.

Settings

Default: `Off` for Second-Order Integrator, `On` for Second-Order Integrator Limited

☑ On

Limit state *dx/dt* of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters.

☐ Off

Do not limit state *dx/dt* of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters.

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

This parameter enables **Upper limit dx/dt**.

This parameter enables **Lower limit dx/dt**.

Command-Line Information

Parameter: `LimitDXDT`
Type: string
Value: `'On' | 'Off'`
Default: `'Off' | 'On'`

Upper limit dx/dt

Specify the upper limit for state *dx/dt*.

Settings

Default: `'inf'`

Tip

If you limit *x*, then this parameter must have a strictly positive value.

Dependencies

Limit dx/dt enables this parameter.

Command-Line Information

Parameter: `UpperLimitDXDT`
Type: scalar or vector
Value: `any valid real value`
Default: `'inf'`

Lower limit dx/dt

Specify the lower limit for state *dx/dt*.

Settings

Default: `'-inf'`

Tip

If you limit *x*, then this parameter must have a strictly negative value.

Dependencies

Limit dx/dt enables this parameter.

Command-Line Information

Parameter: `LowerLimitDXDT`
Type: scalar or vector
Value: `any valid real value`
Default: `'-inf'`

Absolute tolerance dx/dt

Specify the absolute tolerance for computing state *dx/dt*.

Settings

Default: `auto`

- You can enter `auto` or a numeric value.
- If you enter `auto`, 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.

Command-Line Information

Parameter: `AbsoluteToleranceDXDT`
Type: scalar
Value: `any valid positive real value`
Default: `'auto'`

State name dx/dt

Assign a unique name to state *dx/dt*.

Settings

Default: `''`

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 string or a cell array.

Command-Line Information

Parameter: `StateNameDXDT`
Type: string
Value: `any valid expression`
Default: `''`

External reset

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

Settings

Default: `none`

`none`
: Do not reset the state to initial conditions.

`rising`
: Reset the state when the reset signal rises from a zero to a positive value or from a negative to a positive value.

`falling`
: Reset the state when the reset signal falls from a positive value to zero or from a positive to a negative value.

`either`
: Reset the state when the reset signal changes from zero to a nonzero value or changes sign.

Command-Line Information

Parameter: `ExternalReset`
Type: string
Value: `'none' | 'rising' | 'falling' | 'either'`
Default: `'none'`

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Settings

Default: On

☑ On
Enable zero-crossing detection.

☐ Off
Do not enable zero-crossing detection.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Reinitialize dx/dt when x reaches saturation

At the instant when state *x* reaches saturation, reset *dx/dt* to its current initial conditions.

Settings

Default: Off

☑ On

Reset *dx/dt* to its initial conditions when *x* becomes saturated.

☐ Off

Do not reset *dx/dt* to its initial conditions when *x* becomes saturated.

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.

Command-Line Information

Parameter: `ReinitDXDTwhenXreachesSaturation`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Ignore state limits and the reset for linearization

For linearization purposes, have Simulink ignore the specified state limits and the external reset.

Settings

Default: Off

☑ On

Ignore the specified state limits and the external reset.

☐ Off

Apply the specified state limits and the external reset setting.

Command-Line Information

Parameter: `IgnoreStateLimitsAndResetForLinearization`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Show output

Specify the output ports on the block.

Settings

Default: `both`

`both`
: Show both *x* and *dx/dt* output ports.

`x`
: Show only the *x* output port.

`dx/dt`
: Show only the *dx/dt* output port.

Command-Line Information

Parameter: `ShowOutput`
Type: string
Value: `'both'` | `'x'` | `'dxdt'`
Default: `'both'`

Selector

Purpose

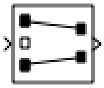
Select input elements from vector, matrix, or multidimensional signal

Library

Signal Routing

Description

The Selector block generates as output selected or reordered elements of an input vector, matrix, or multidimensional signal.



A Selector block accepts vector, matrix, or multidimensional signals as input. The parameter dialog box and the block's appearance change to reflect the number of dimensions of the input.

Based on the value you enter for the **Number of input dimensions** parameter, a table of indexing settings is displayed. Each row of the table corresponds to one of the input dimensions in **Number of input dimensions**. For each dimension, you define the elements of the signal to work with. Specify a vector signal as a 1-D signal and a matrix signal as a 2-D signal. When you configure the Selector block for multidimensional signal operations, the block icon changes.

For example, assume a 6-D signal with a one-based index mode. The table of the Selector block dialog changes to include one row for each dimension. Suppose that you define each dimension with the following entries:

- Dimension 1
  - **Index Option**, select `Select all`
- Dimension 2
  - **Index Option**, select `Starting index (dialog)`
  - **Index**, enter `2`
  - **Output Size**, enter `5`
- Dimension 3
  - **Index Option**, select `Index vector (dialog)`
  - **Index**, enter `[1 3 5]`
- Dimension 4

  - **Index Option**, select `Starting index (port)`
  - **Output Size**, enter `8`
- Dimension 5
  - **Index Option**, select `Index vector (port)`
- Dimension 6
  - **Index Option**, select `Starting and ending indices (port)`

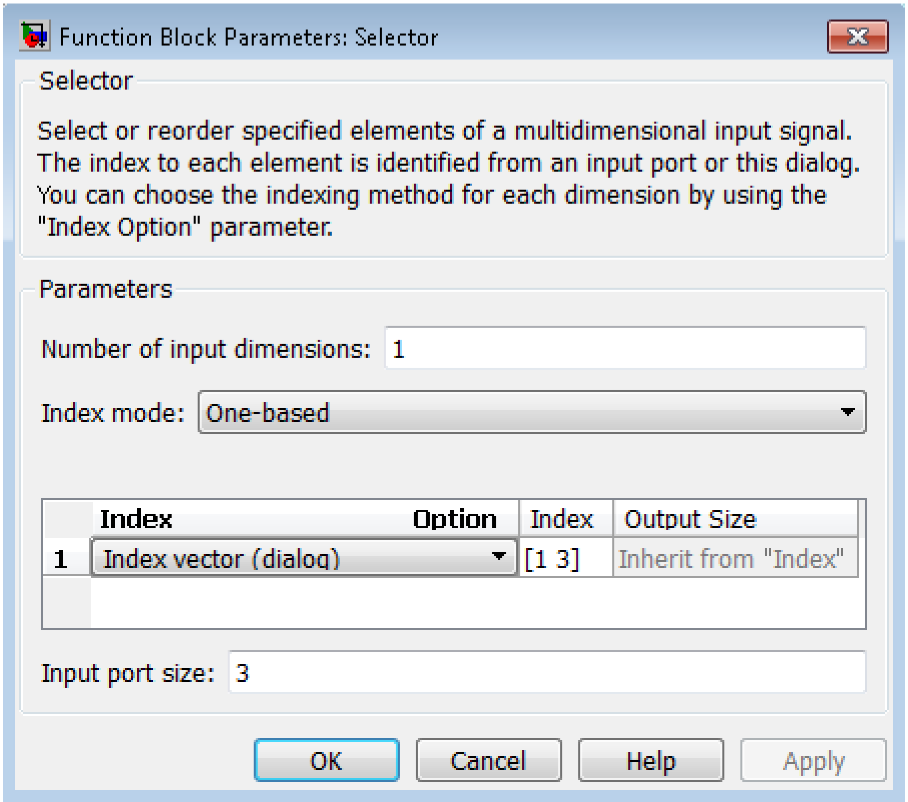
The output will be `Y=U(1:end,2:6,[1 3 5],Idx4:Idx4+7,Idx5, Idx6(1):Idx6(2))`, where `Idx4`, `Idx5`, and `Idx6` are the index ports for dimensions 4, 5, and 6.

You can use an array of buses as an input signal to a Selector block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

Data Type Support

The data port of the Selector block accepts signals of any signal type and any data type that Simulink supports, including fixed-point and enumerated data types. The data port accepts mixed-type signals. The index port accepts built-in data types, but not Boolean. The elements of the output have the same type as the corresponding selected input elements.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Select or reorder specified elements of a multidimensional input signal. The index to each element is identified from an input port or this dialog. You can choose the indexing method for each dimension by using the "Index Option" parameter.

Number of input dimensions: 1

Index mode: One-based

	Index Option	Index	Output Size
1	Index vector (dialog)	[1 3]	Inherit from "Index"

Input port size: 3

Parameters and Dialog Box

Number of input dimensions

Enter the number of dimensions of the input signal.

Index mode

Specifies the indexing mode: `One-based` or `Zero-based`. If `One-based` is selected, an index of 1 specifies the first element of the input vector, 2, the second element, and so on. If `Zero-based` is selected, an index of 0 specifies the first element of the input vector, 1, the second element, and so on.

Index Option

Define, by dimension, how the elements of the signal are to be indexed. From the list, select:

- `Select all`

  No further configuration is required. All elements are selected.

- `Index vector (dialog)`

  Enables the **Index** column. Enter the vector of indices of the elements.

- `Index vector (port)`

  No further configuration is required.

- `Starting index (dialog)`

  Enables the **Index** and **Output Size** columns. Enter the starting index of the range of elements to select in the **Index** column and the number of elements to select in the **Output Size** column.

- `Starting index (port)`

  Enables the **Output Size** column. Enter the number of elements to be selected in the **Output Size** column.

- `Starting and ending indices (port)`

  No further configuration is required.

  Note 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.

The **Index** and **Output Size** columns appear as needed.

Index

If the **Index Option** is `Index vector (dialog)`, enter the index of each element in which you are interested.

If the **Index Option** is `Starting index vector (dialog)`, enter the starting index of the range of elements to be selected.

Output Size

Enter the width (number of elements from the starting point) of the block output signal.

Input port size

Specify the width of the block input signal (`-1` for inherited) — 1-D signals only.

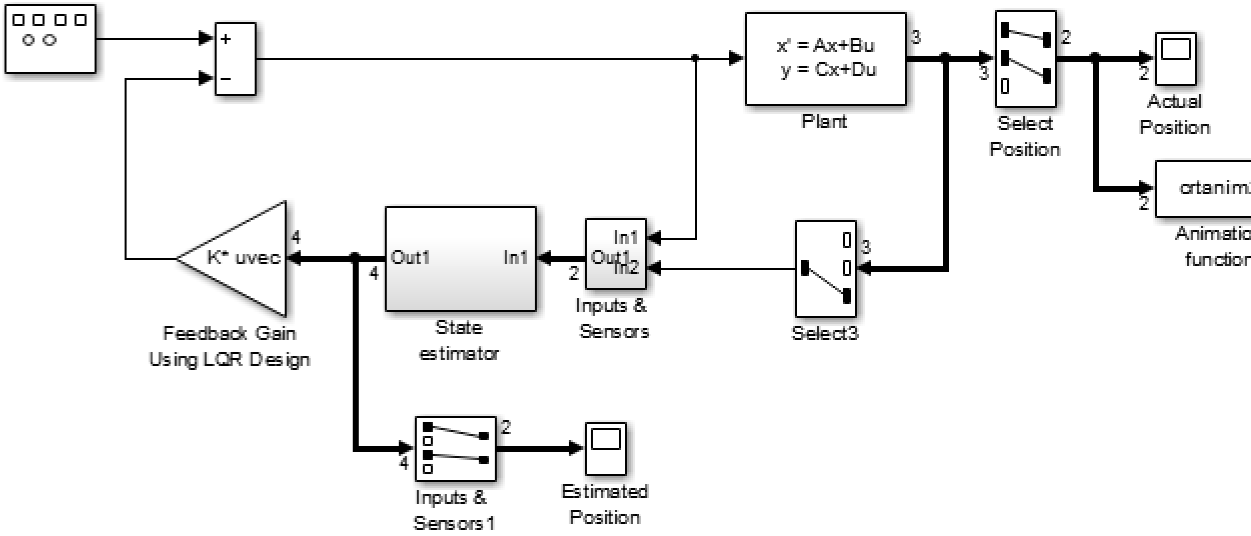
Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Note For 1-D signals, the **Sample time** parameter is available only when **Index Option** is `Starting index (port)`, `Index vector (port)`, or `Starting and ending indices (port)`. For all other **Index Option** settings, the Selector block becomes a virtual block and the **Sample time** parameter does not appear.

Examples

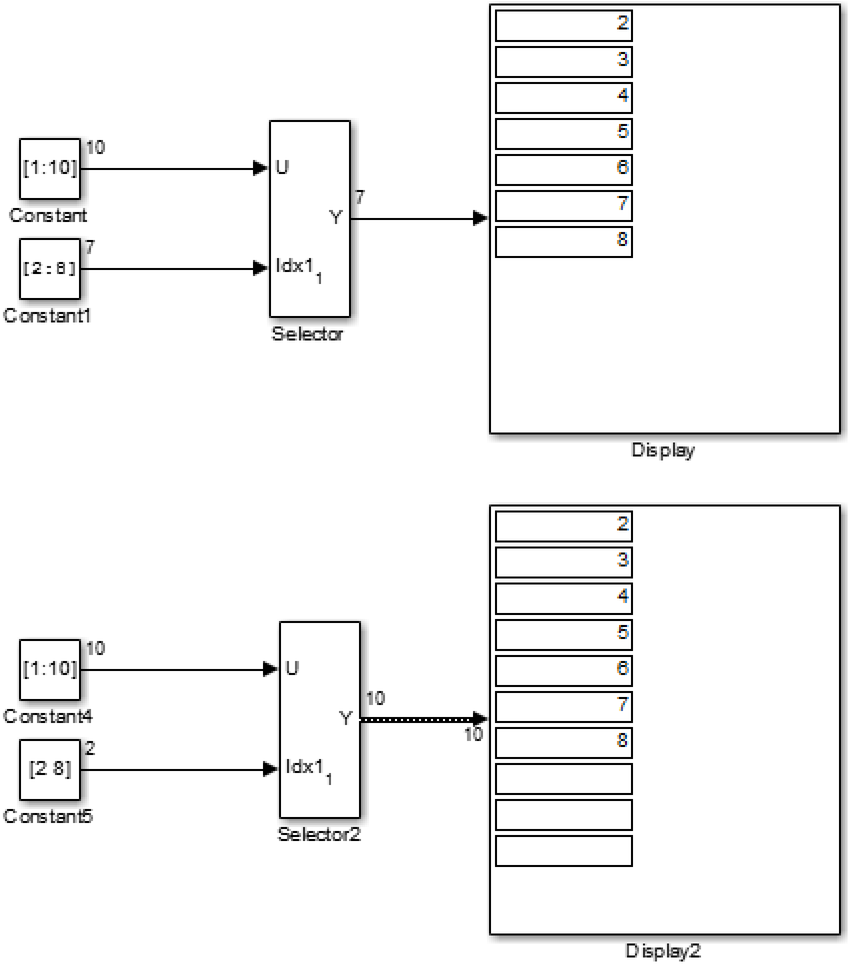
The `sldemo_dblcart1` model shows the use of three Selector blocks. The following diagram shows what the model looks like after you enable **Display > Signals & Ports > Signal Dimensions** and simulate the model.


+
−
x' = Ax+Bu
y = Cx+Du
Plant
3
3
2
Select Position
2
Actual Position
2
ortanim
Animatio
functio
4
K* uvec
Feedback Gain Using LQR Design
4
Out1
In1
State estimator
2
In1
Out1
In2
Inputs & Sensors
3
Select3
4
2
Inputs & Sensors1
Estimated Position

All three Selector blocks set the **Index Option** to `Index vector (dialog)`, which allows you to specify the indices of the specific signals that you want to select, using the **Index** parameter. The **Input port size** parameter is set to the dimension of the largest input signal.

The following model shows the result of simulating two Selector blocks that have the same kind of input signals, but which have two different **Index Option** settings.

Both Selector blocks select 7 values from the input signal that feeds the `U` port. However, the `Selector` block outputs a fixed-size signal, whereas the `Selector2` block outputs a variable-size signal whose compiled signal dimension is 10 instead of 7.


[1:10]
10
Constant
[2:8]
7
Constant1
U
Y
Idx1 1
7
Selector
2
3
4
5
6
7
8
Display
[1:10]
10
Constant4
[2 8]
2
Constant5
U
Y
Idx1 1
10
10
Selector2
2
3
4
5
6
7
8
Display2

The `Selector` block sets **Index Option** to `Index vector (port)`, which uses the input signal from `Constant1` as the index vector. The dimension of the input signal is 7, so the `Display` block shows the 7 values of the `Constant1` block. The `Selector2` block sets the **Input port size** parameter to `10`, which is the size of the largest input signal to the `Selector1` block.

The `Selector2` block uses the same configuration as the `Selector` block, except that the **Index Option** is set to `Starting and ending indices (port)`. The output uses the size of **Input port size** parameter (`10`), even though the size of the input signal is 7. The `Display2` block shows empty boxes for the three extra dimensions.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes
Multidimensionalized	Yes
Virtual	Yes, when **Number of input dimensions** is `1` and **Index Option** is `Select all`, `Index vector (dialog)`, or `Starting index (dialog)` For more information, see "Virtual Blocks" in the Simulink documentation.
Zero-Crossing Detection	No

Purpose

Include S-function in model

Library

User-Defined Functions

Description

The S-Function block provides access to S-functions from a block diagram. The S-function named as the **S-function name** parameter can be a Level-1 MATLAB or a Level-1 or Level-2 C MEX S-function (see "S-Function Basics" for information on how to create S-functions).

Note Use the Level-2 MATLAB S-Function block to include a Level-2 MATLAB S-function in a block diagram.

The S-Function block allows additional parameters to be passed directly to the named S-function. The function parameters can be specified as MATLAB expressions or as variables separated by commas. For example,

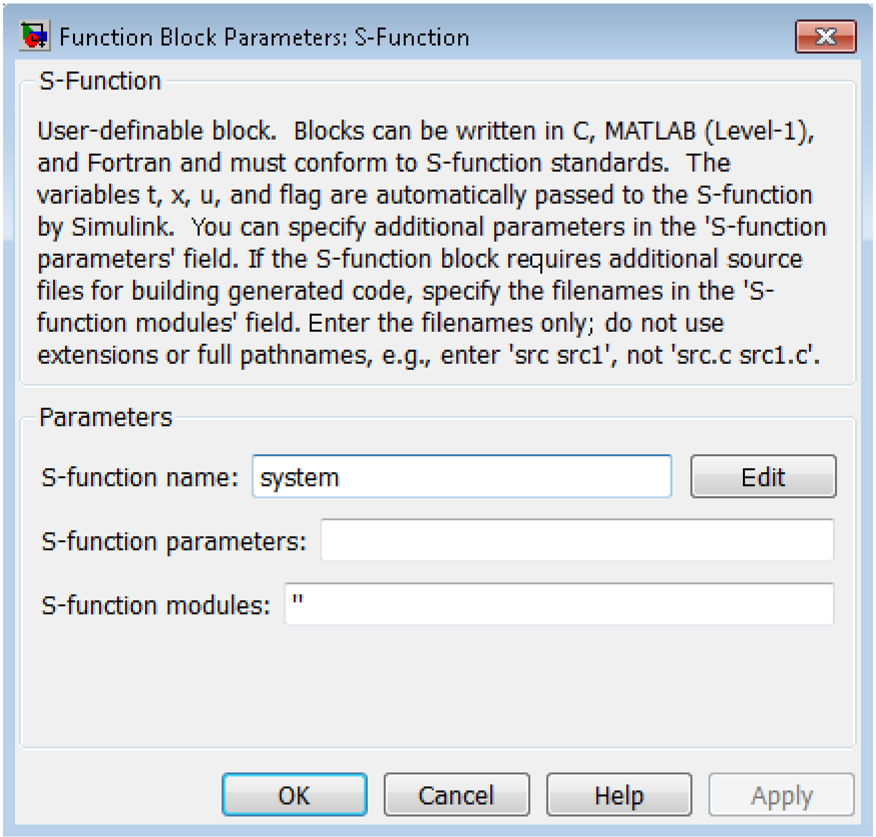
```
A, B, C, D, [eye(2,2);zeros(2,2)]
```

Note that although individual parameters can be enclosed in brackets, the list of parameters must not be enclosed in brackets.

The S-Function block displays the name of the specified S-function and the number of input and output ports specified by the S-function. Signals connected to the inputs must have the dimensions specified by the S-function for the inputs.

Data Type Support

Depends on the implementation of the S-Function block.



Function Block Parameters: S-Function

S-Function

User-definable block. Blocks can be written in C, MATLAB (Level-1), and Fortran and must conform to S-function standards. The variables t, x, u, and flag are automatically passed to the S-function by Simulink. You can specify additional parameters in the 'S-function parameters' field. If the S-function block requires additional source files for building generated code, specify the filenames in the 'S-function modules' field. Enter the filenames only; do not use extensions or full pathnames, e.g., enter 'src src1', not 'src.c src1.c'.

Parameters

S-function name: system

S-function parameters:

S-function modules: ''

Parameters and Dialog Box

S-function name

The S-function name.

S-function parameters

Additional S-function parameters. See the preceding block description for information on how to specify the parameters.

S-function modules

This parameter applies only if this block represents a C MEX S-function and you intend to use the Simulink Coder software to generate code from the model containing the block. If you use it, when you are ready to generate code, you must force the coder to rebuild the top model as explained in "Control Regeneration of Top Model Code".

For more information on using this parameter, see "Specify Additional Source Files for an S-Function" in the Simulink Coder documentation.

Characteristics

Direct Feedthrough	Depends on contents of S-function
Sample Time	Depends on contents of S-function
Scalar Expansion	Depends on contents of S-function
Dimensionalized	Depends on contents of S-function
Multidimensionalized	Yes
Zero-Crossing Detection	No

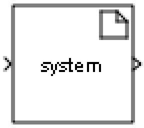
Purpose

Create S-function from C code that you provide

Library

User-Defined Functions

Description



The S-Function Builder block creates a C MEX S-function from specifications and C source code that you provide. See “Build S-Functions Automatically” for detailed instructions on using the S-Function Builder block to generate an S-function.

Instances of the S-Function Builder block also serve as wrappers for generated S-functions in Simulink models. When simulating a model containing instances of an S-Function Builder block, Simulink software invokes the generated S-function associated with each instance to compute the instance’s output at each time step.

Note The S-Function Builder block does not support masking. However, you can mask a Subsystem block that contains an S-Function Builder block. For more information, see “Masking”.

Data Type Support

The S-Function Builder can accept and output complex, 1-D, or 2-D signals and nonvirtual buses. For each of these cases, the signals must have a data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

See “S-Function Builder Dialog Box” in the online documentation for information on using the S-Function Builder block’s parameter dialog box.

Shift Arithmetic

Purpose 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
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

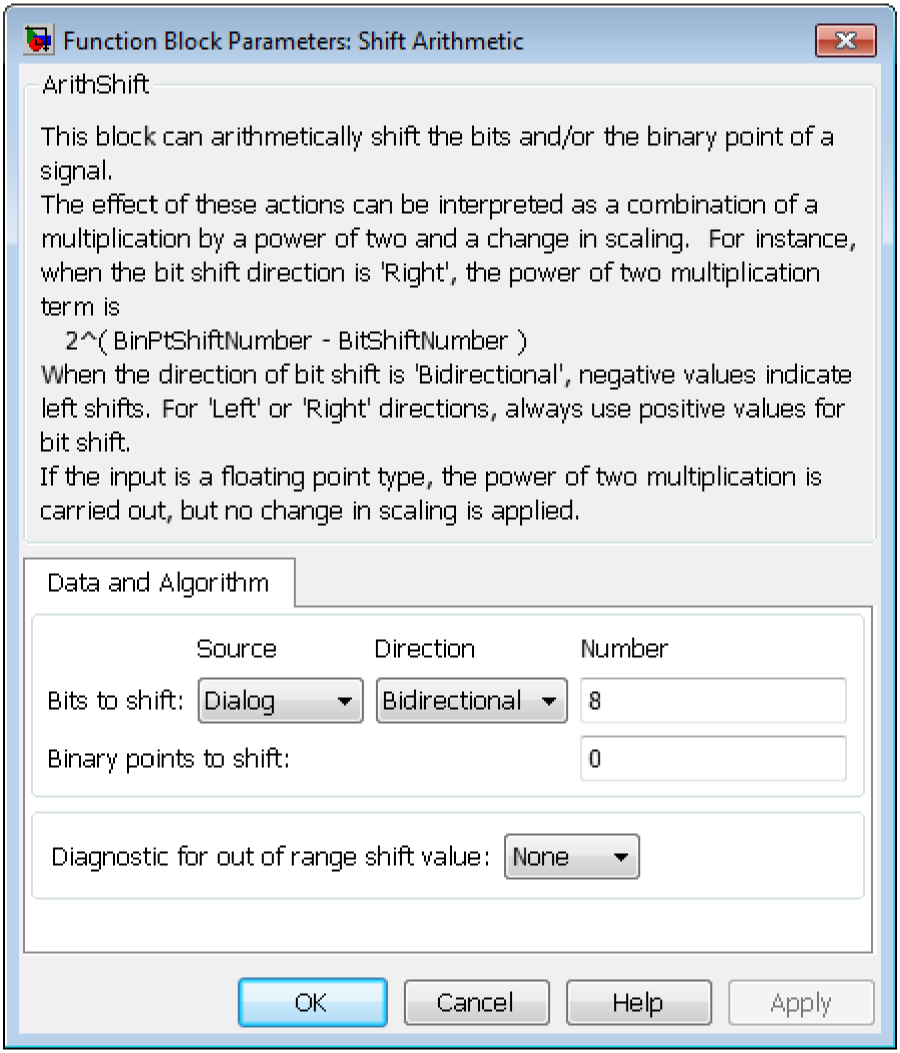
The block parameters support the following data types:

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 and Dialog Box

The Shift Arithmetic block dialog box appears as follows:



ArithShift

This block can arithmetically shift the bits and/or the binary point of a signal.

The effect of these actions can be interpreted as a combination of a multiplication by a power of two and a change in scaling. For instance, when the bit shift direction is 'Right', the power of two multiplication term is

2^(BinPtShiftNumber - BitShiftNumber)

When the direction of bit shift is 'Bidirectional', negative values indicate left shifts. For 'Left' or 'Right' directions, always use positive values for bit shift.

If the input is a floating point type, the power of two multiplication is carried out, but no change in scaling is applied.

Data and Algorithm

	Source	Direction	Number
Bits to shift:	Dialog	Bidirectional	8
Binary points to shift:			0

Diagnostic for out of range shift value: None

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` — No warning or error appears.
- `Warning` — Display a warning in the MATLAB Command Window and continue the simulation.
- `Error` — Halt the simulation and display an error in the Simulation Diagnostics Viewer.

For more information, see “Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values” on page 1-1527.

Check for out-of-range 'Bits to shift' in generated code

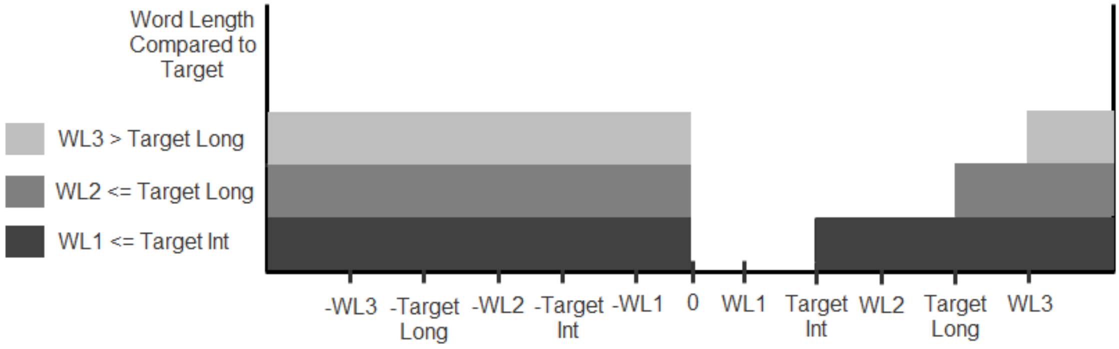
Select this check box to include conditional statements in the generated code that protect against out-of-range bit shift values. This check box is available when **Bits to shift: Source** is `Input port`.

For more information, see "Code Generation for Out-of-Range Bit Shift Values" on page 1-1529.

Out-of-Range Bit Shifts

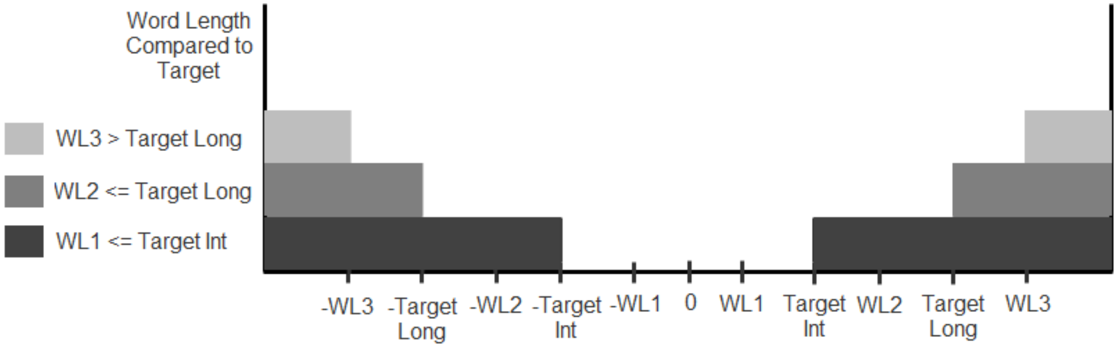
Definition of an Out-of-Range Bit Shift

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.


Word Length Compared to Target
WL3 > Target Long
WL2 <= Target Long
WL1 <= Target Int
-WL3
-Target Long
-WL2
-Target Int
-WL1
0
WL1
Target Int
WL2
Target Long
WL3

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.


Word Length Compared to Target
WL3 > Target Long
WL2 <= Target Long
WL1 <= Target Int
-WL3
-Target Long
-WL2
-Target Int
-WL1
0
WL1
Target Int
WL2
Target Long
WL3

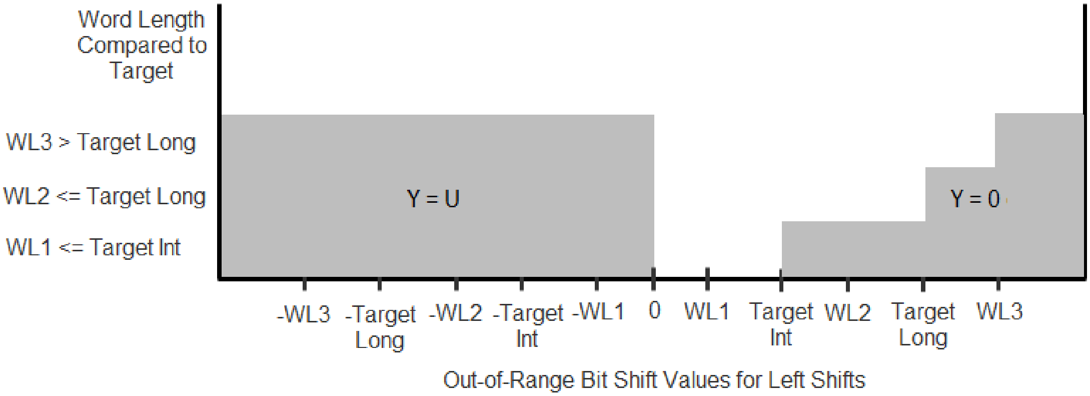
Out-of-Range Bit Shift Values for Bidirectional Shifts

The diagnostic for out-of-range bit shifts responds as follows, depending on the mode of operation:

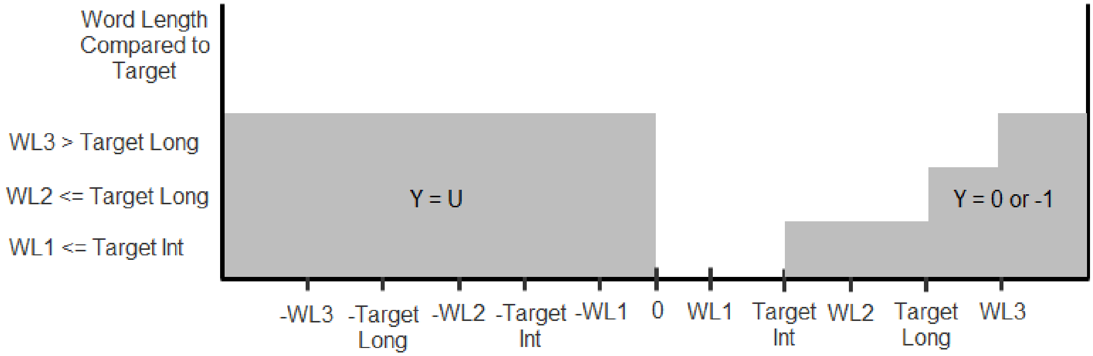
Mode	Diagnostic for out-of-range shift value		
	None	**Warning**	**Error**
Simulation	Do not report any warning or error.	Report a warning but continue simulation.	Report an error and stop simulation.
Accelerator modes and code generation	Has no effect.		

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:

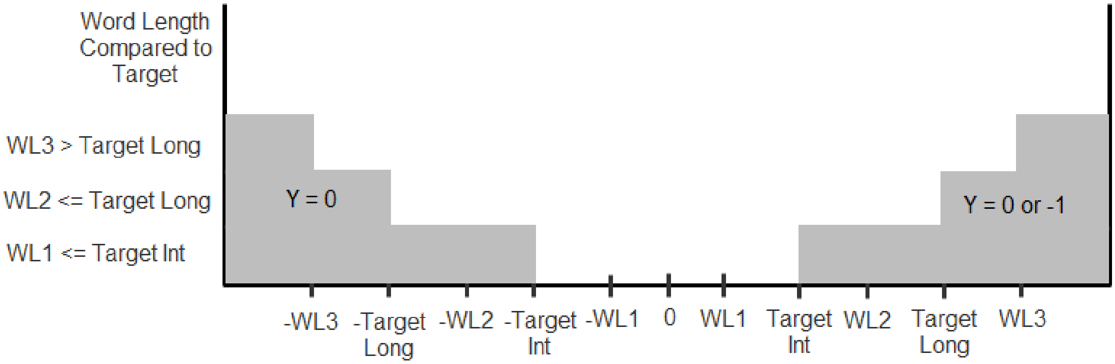


Similarly, the output for an out-of-range bit shift value for right shifts is as follows:


Word Length Compared to Target
WL3 > Target Long
WL2 <= Target Long
WL1 <= Target Int
Y = U
Y = 0 or -1
-WL3
-Target Long
-WL2
-Target Int
-WL1
0
WL1
Target Int
WL2
Target Long
WL3

Out-of-Range Bit Shift Values for Right Shifts

For bidirectional shifts, the output for an out-of-range bit shift value is as follows:


Word Length Compared to Target
WL3 > Target Long
WL2 <= Target Long
WL1 <= Target Int
Y = 0
Y = 0 or -1
-WL3
-Target Long
-WL2
-Target Int
-WL1
0
WL1
Target Int
WL2
Target Long
WL3

Out-of-Range Bit Shift Values for Bidirectional Shifts

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**.

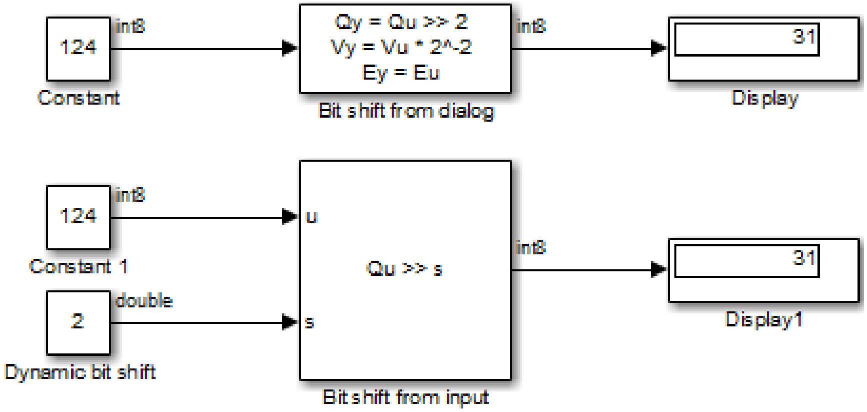
Check Box Setting	Generated Code	Simulation Results Compared to Generated Code
Selected	Includes conditional statements to protect against out-of-range bit shift values.	Simulation and Accelerator mode results match those of code generation.
Cleared	Does not protect against out-of-range bit shift values.	• 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.


124
int8
Constant
Qy = Qu >> 2
Vy = Vu * 2^-2
Ey = Eu
Bit shift from dialog
int8
31
Display
124
int8
Constant 1
2
double
Dynamic bit shift
u
s
Qu >> s
Bit shift from input
int8
31
Display1

The key block parameter settings of the Constant blocks are:

Block	Parameter	Setting
`Constant` and `Constant1`	**Constant value**	`124`
	Output data type	`int8`
`Dynamic bit shift`	**Constant value**	`2`
	Output data type	`Inherit: Inherit from 'Constant value'`

The key block parameter settings of the Shift Arithmetic blocks are:

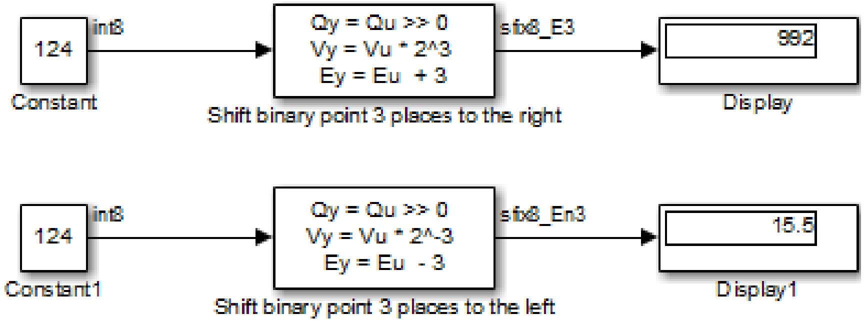
Block	Parameter	Setting
`Bit shift from dialog`	**Bits to shift: Source**	`Dialog`
	Bits to shift: Direction	`Right`
	Bits to shift: Number	`2`
`Bit shift from input`	**Bits to shift: Source**	`Input port`
	Bits to shift: Direction	`Right`

The top Shift Arithmetic block takes an input of 124, which corresponds to 01111100 in binary format. Shifting the number of bits two places to the right produces 00011111 in binary format. Therefore, the block outputs 31.

The bottom Shift Arithmetic block performs the same operation as the top block. However, the bottom block receives the bit shift value through an input port instead of the dialog box. By supplying this value as an input signal, you can change the number of bits to shift during simulation.

Block Output for Binary Point Shifts

The following model shows the effect of binary point shifts.


124
int8
Constant
Qy = Qu >> 0
Vy = Vu * 2^3
Ey = Eu + 3
Shift binary point 3 places to the right
sfix8_E3
992
Display
124
int8
Constant1
Qy = Qu >> 0
Vy = Vu * 2^-3
Ey = Eu - 3
Shift binary point 3 places to the left
sfix8_En3
15.5
Display1

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

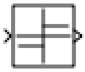
Direct Feedthrough	Yes
Sample Time	Inherited from the driving block

Scalar Expansion	Yes
Dimensionalized	Yes
Multidimensionalized	Yes, for the input port that specifies bit shift values
Zero-Crossing Detection	No

Purpose Indicate sign of input

Library Math Operations

Description

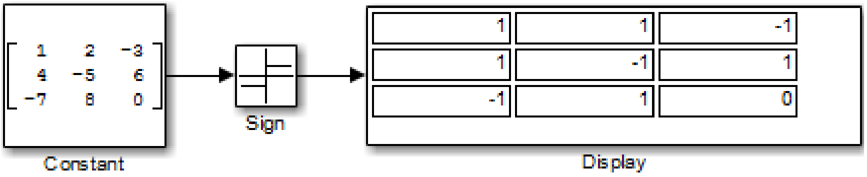


Block Behavior for Real Inputs

For real inputs, the Sign block outputs the sign of the input:

Input	Output
Greater than zero	1
Equal to zero	0
Less than zero	–1

For vector and matrix inputs, the block outputs a vector or matrix where each element is the sign of the corresponding input element, as shown in this example:

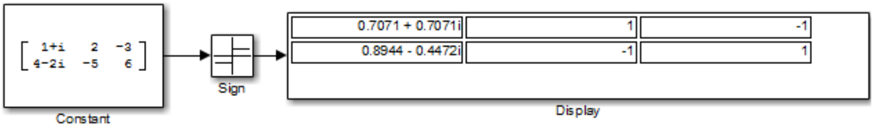


Block Behavior for Complex Inputs

When the input `u` is a complex scalar, the block output matches the MATLAB result for:

```
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:


1+i 2 -3
4-2i -5 6
Constant
Sign
0.7071 + 0.7071i
1
-1
0.8944 - 0.4472i
-1
1
Display

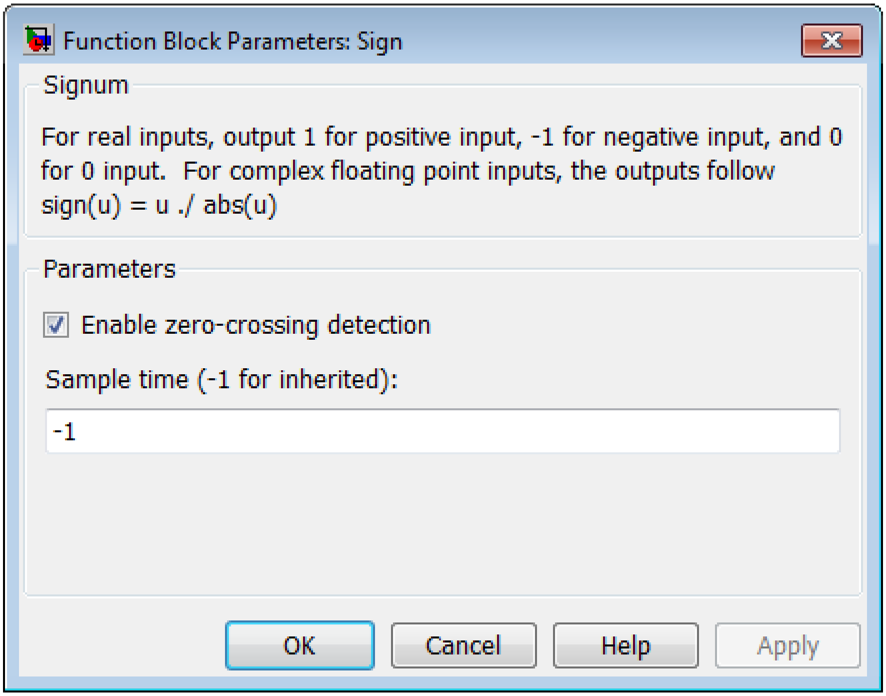
Data Type Support

The Sign block supports real inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The block supports complex inputs only for floating-point data types, `double` and `single`. The output uses the same data type as the input.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Signum

For real inputs, output 1 for positive input, -1 for negative input, and 0 for 0 input. For complex floating point inputs, the outputs follow sign(u) = u ./ abs(u)

Parameters

☑ Enable zero-crossing detection

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Enable zero-crossing detection
Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Sample time (-1 for inherited)
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation for more information.

Examples

The following Simulink examples show how to use the Sign block:

- `sldemo_fuelsys` (in the Engine Gas Dynamics/Throttle & Manifold/Throttle subsystem)
- `sldemo_hardstop`

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	N/A
Dimensionalized	Yes
Multidimensionalized	No
Zero-Crossing Detection	Yes, if enabled

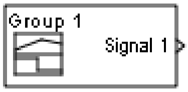
Purpose

Create and generate interchangeable groups of signals whose waveforms are piecewise linear

Library

Sources

Description



The Signal Builder block allows you to create interchangeable groups of piecewise linear signal sources and use them in a model. See “Signal Groups” in the Simulink documentation.

Note Use the `signalbuilder` function to create and access Signal Builder blocks programmatically.

Data Type Support

The Signal Builder block outputs a virtual nonhierarchical bus, scalar, or array of real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The Signal Builder block has the same dialog box as that of a Subsystem block. To display the dialog box, right-click the block and select **Subsystem Parameters**.

Examples

The following examples show how to use the Signal Builder block:

- `sldemo_pid2dof`

- `sf_test_vectors`

Characteristics

Sample Time	Specified in the **Sample time** parameter, accessible by selecting **File > Simulation Options** in the Signal Builder block • Zero represents a continuous sample time. • A positive integer represents a discrete sample time.
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes
Zero Crossing	Yes

Purpose

Convert signal to new type without altering signal values

Library

Signal Attributes

Description

The Signal Conversion block converts a signal from one type to another. Use the **Output** parameter to select the type of conversion to perform.

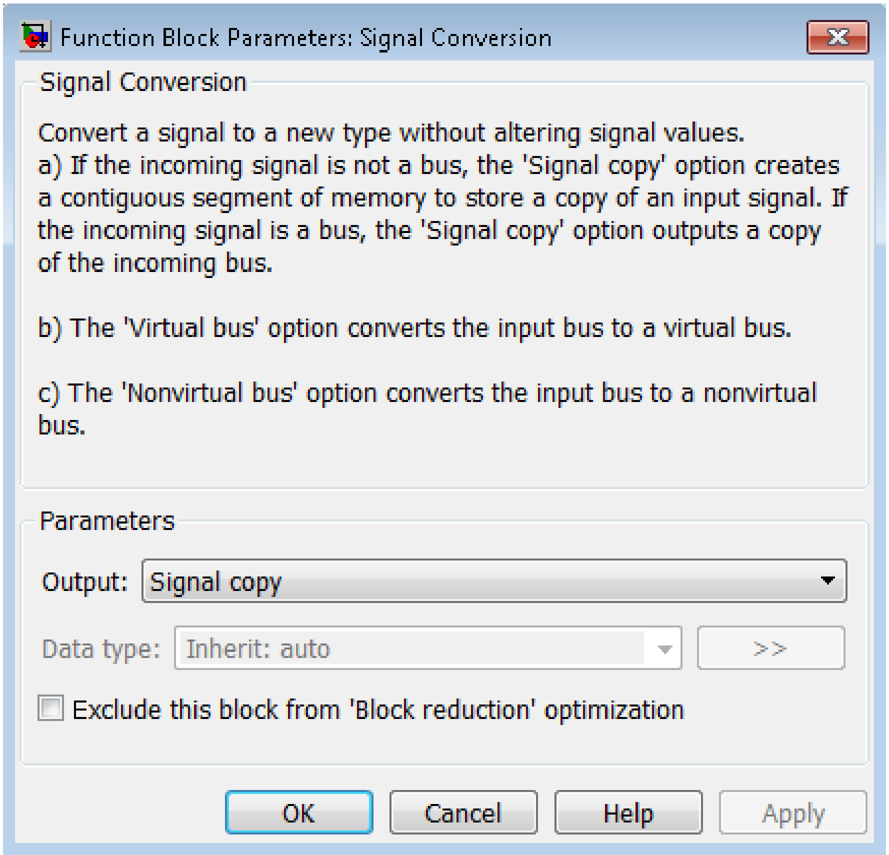
Data Type Support

The Signal Conversion block accepts signals of the following data types:

- Scalar
- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated
- Virtual and nonvirtual bus signals

You can use an array of buses as an input signal to a Signal Conversion block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

For more information about data types, see "Data Types Supported by Simulink" in the Simulink documentation.



Signal Conversion

Convert a signal to a new type without altering signal values.
a) If the incoming signal is not a bus, the 'Signal copy' option creates a contiguous segment of memory to store a copy of an input signal. If the incoming signal is a bus, the 'Signal copy' option outputs a copy of the incoming bus.

b) The 'Virtual bus' option converts the input bus to a virtual bus.

c) The 'Nonvirtual bus' option converts the input bus to a nonvirtual bus.

Parameters

Output: Signal copy

Data type: Inherit: auto

☐ Exclude this block from 'Block reduction' optimization

Parameters and Dialog Box

Output

Specify the type of conversion to perform. The type of conversion that you use depends on your modeling goal.

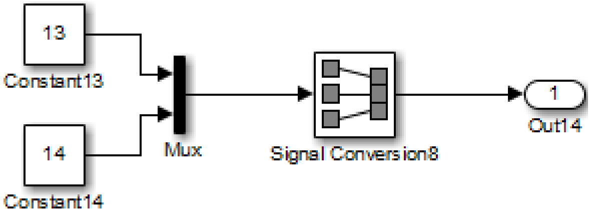
Modeling Goal	**Output Option**
Reduce generated code for a muxed signal. For an example involving Simulink Coder software, see “Reusable Code and Referenced Models”.	`Signal copy`
Connect a block with a constant sample time to an output port of an enabled subsystem. For more information, see “Using Blocks with Constant Sample Times in Enabled Subsystems”.	`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. “Composite Signals” in the Simulink documentation for more information.	`Signal copy`
Save memory by converting a nonvirtual bus to a virtual bus.	`Virtual bus`
Pass a virtual bus signal to a modeling construct that requires a nonvirtual bus, such as a Model block.	`Nonvirtual bus`
Pass a nonvirtual bus signal from a Bus Selector block.	`Nonvirtual bus`

- The `Signal copy` option is the default. The type of conversion that the Signal Conversion block performs using the `Signal copy` option depends on the type of input signal.

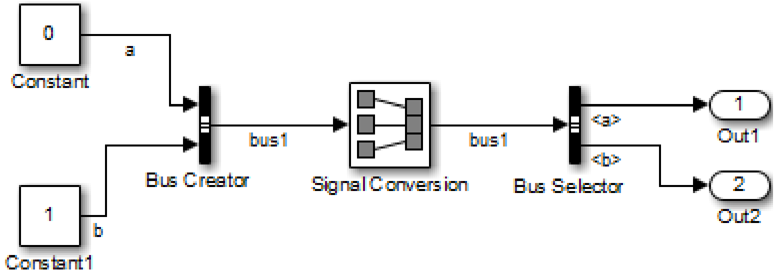
Type of Input Signal	Conversion that the Signal Copy Option Performs
Muxed (nonbus) signal	Converts the muxed signal, whose elements occupy discontiguous areas of memory, to a vector signal, whose elements occupy contiguous areas of memory. The conversion allocates a contiguous area of memory for the elements of the muxed signal and copies the values from the discontiguous areas (represented by the block's input) to the contiguous areas (represented by the block's output) at each time step.
Bus signal	Outputs a contiguous copy of the bus signal that is the input to the Signal Conversion block.

For an array of buses input signal, use the `Signal copy` option.

In the following example, a muxed signal inputs to a Signal Conversion block that has the **Output** parameter set to `Signal copy`. The Signal Conversion block converts the input signal to a vector.

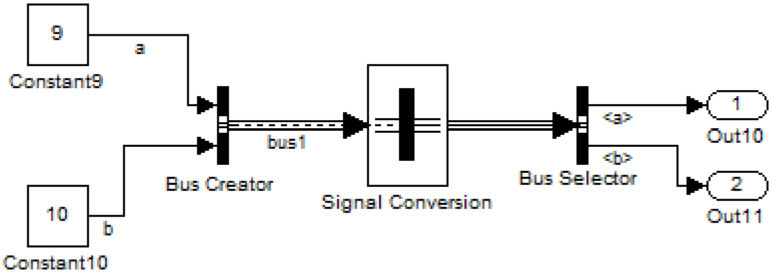


In the following example, a nonvirtual bus signal from a Bus Creator block inputs to a Signal Conversion block that has **Output** set to `Signal copy`. The Signal Conversion block creates another contiguous copy of the input bus signal.


0
Constant
a
1
Constant1
b
Bus Creator
bus1
Signal Conversion
bus1
Bus Selector
<a>
<b>
1
Out1
2
Out2

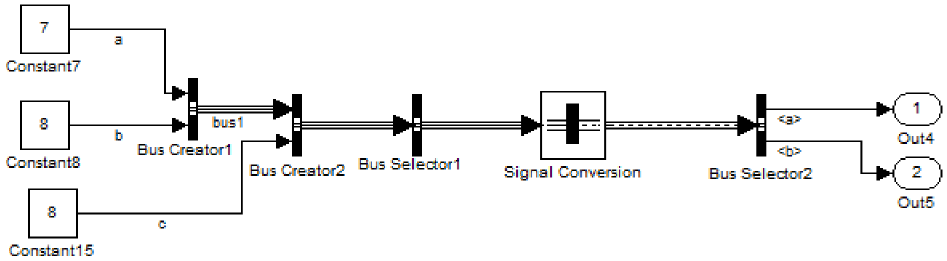
- The `Virtual bus` option converts a nonvirtual bus to a virtual bus.

  In the following example, a Bus Creator block inputs to a Signal Conversion block that has **Output** set to `Virtual bus`. The Signal Conversion block converts the nonvirtual bus signal from the Bus Creator block to a virtual bus signal that inputs to the Bus Selector block.


9
Constant9
a
10
Constant10
b
Bus Creator
bus1
Signal Conversion
Bus Selector
<a>
<b>
1
Out10
2
Out11

- The `Nonvirtual bus` option converts a virtual bus to a nonvirtual bus.

In the following example, the Signal Conversion block converts a virtual bus signal from the first Bus Selector block to a nonvirtual bus signal that inputs to the second Bus Selector block. The Signal Conversion block has its **Output** parameter set to `Nonvirtual bus`, and specifies a bus object that matches the bus signal hierarchy of the bus that the first Bus Creator block outputs.


7
Constant7
a
8
Constant8
b
8
Constant15
c
bus1
Bus Creator1
Bus Creator2
Bus Selector1
Signal Conversion
<a>
<b>
Bus Selector2
1
Out4
2
Out5

Data type

Specify the output data type of the nonvirtual bus that the Signal Conversion block produces.

This option is available only when you set the **Output** parameter to `Nonvirtual bus`.

The default option is `Inherit: auto`, which uses a rule that inherits a data type.

You must specify a `Simulink.Bus` object in the **Data type** parameter for one or both of the following blocks:

- Signal Conversion block
- An upstream Bus Creator block

If you specify a bus object for the Signal Conversion block, but not for its upstream Bus Creator block, then use a bus object that matches the hierarchy of the bus that upstream Bus Creator block outputs.

If you specify a bus object for both the Signal Conversion block and its upstream Bus Creator block, use the same bus object for both blocks.

You can select the button to the right of the Data type parameter to open the Data Type Assistant, which helps you to set the **Data type** parameter.

See “Specify Block Output Data Types” in *Simulink User’s Guide* for more information.

Exclude this block from ’Block reduction’ optimization

This option is available only when you set the **Output** parameter to `Signal copy`. If the elements of the input signal occupy contiguous areas of memory, then as an optimization, Simulink software eliminates the block from the compiled model . If you select the **Exclude this block from ’Block reduction’ optimization** check box, the optimization occurs the next time you compile the model. For more information, see “Block reduction”.

Characteristics

Sample Time	Inherited from the driving block
Scalar Expansion	n/a
Dimensionalized	n/a
Multidimensionalized	Yes
Zero-Crossing Detection	No

Signal Generator

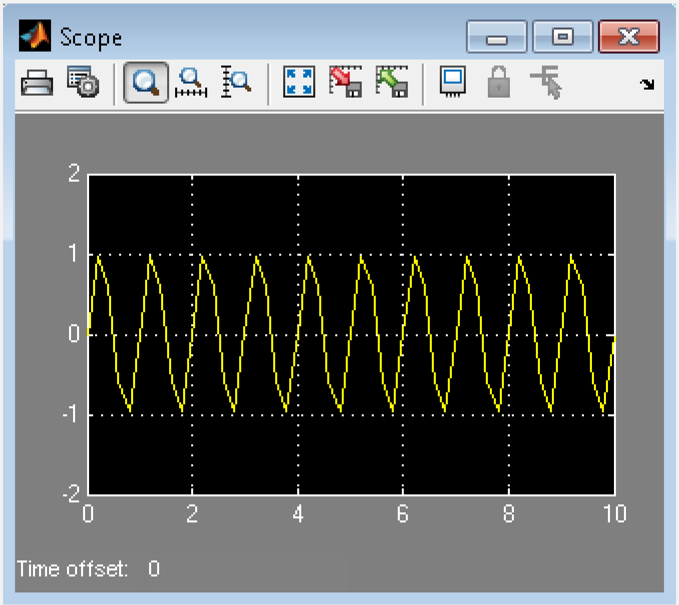
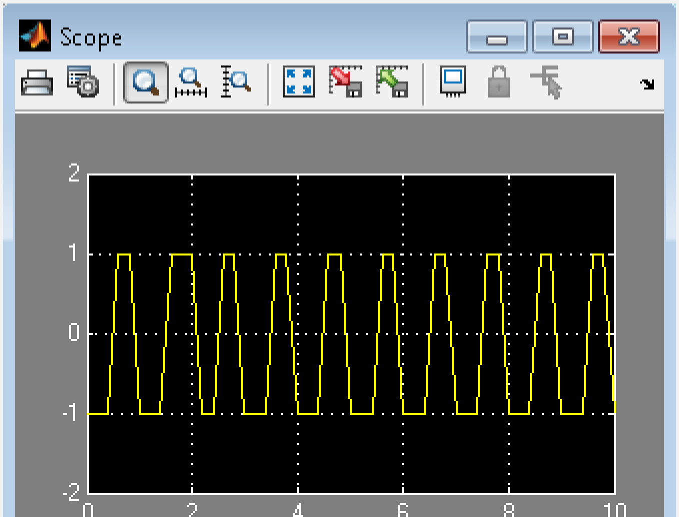
Purpose Generate various waveforms

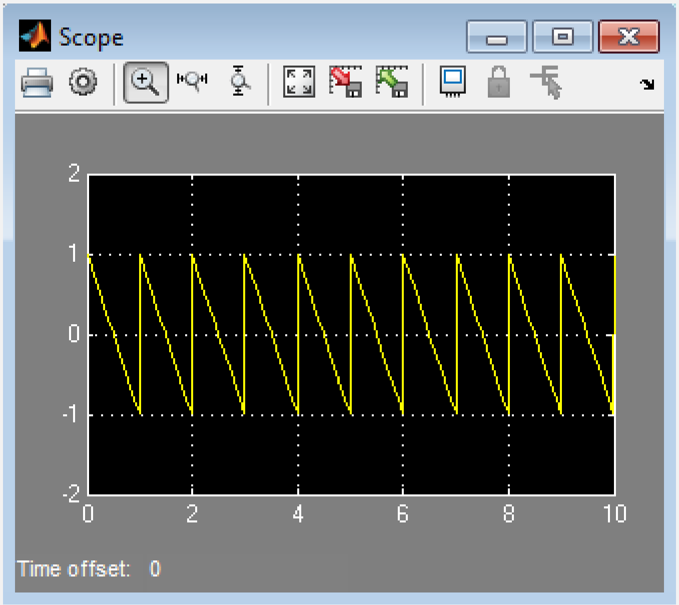
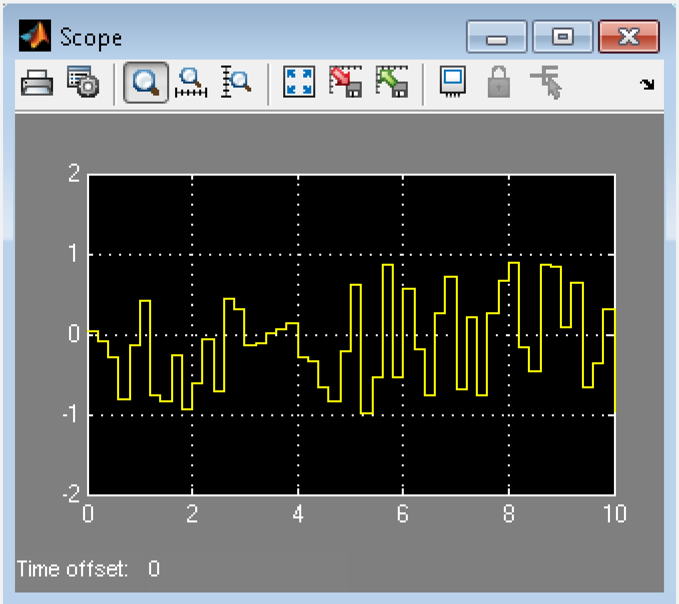
Library Sources

Description

Supported Operations

The Signal Generator block can produce one of four different waveforms: sine wave, square wave, sawtooth wave, and random wave. You can express signal parameters in Hertz (the default) or radians per second. Using default parameter values, you get one of the following waveforms:

Waveform	Scope Output
Sine wave	 Scope 2 1 0 -1 -2 0 2 4 6 8 10 Time offset: 0
Square wave	 Scope 2 1 0 -1 -2 0 2 4 6 8 10

Waveform	Scope Output
Sawtooth wave	 Scope 2 1 0 -1 -2 0 2 4 6 8 10 Time offset: 0
Random wave	 Scope 2 1 0 -1 -2 0 2 4 6 8 10 Time offset: 0

A negative **Amplitude** parameter value causes a 180-degree phase shift. You can generate a phase-shifted wave at other than 180 degrees in many ways. For example, you can connect a Clock block signal to a MATLAB Function block and write the equation for the specific wave.

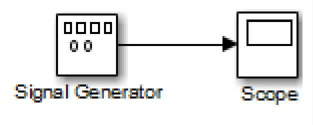
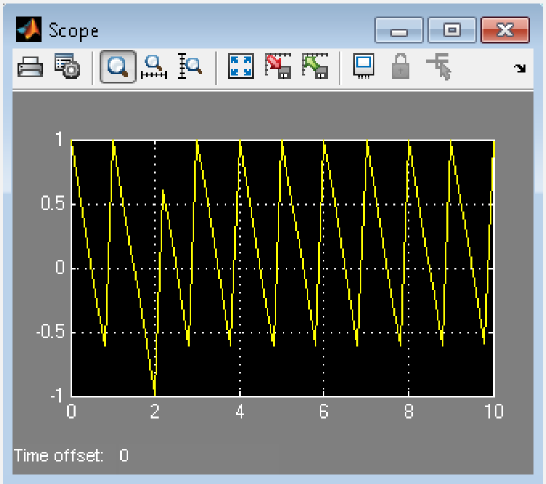
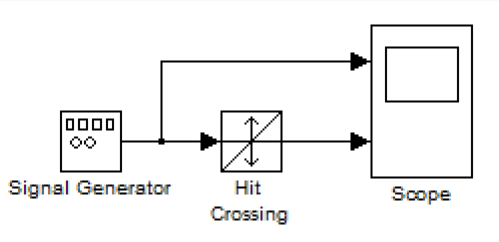
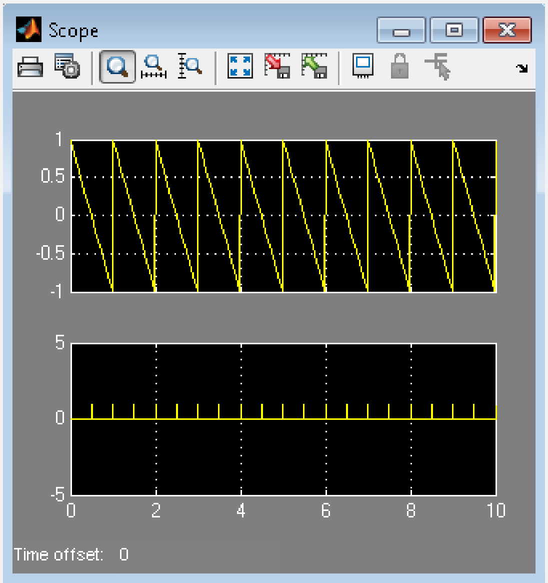
You can vary the output settings of the Signal Generator block while a simulation is in progress. This is useful to determine quickly the response of a system to different types of inputs.

The **Amplitude** and **Frequency** parameters determine the amplitude and frequency of the output signal. The parameters must be of the same dimensions after scalar expansion. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensions as the **Amplitude** and **Frequency** parameters (after scalar expansion). If you select the **Interpret vector parameters as 1-D** check box, the block outputs a vector (1-D) signal if the **Amplitude** and **Frequency** parameters are row or column vectors, that is, single row or column 2-D arrays. Otherwise, the block outputs a signal of the same dimensions as the parameters.

Solver Considerations

If your model uses a fixed-step solver, Simulink uses the same step size for the entire simulation. In this case, the Signal Generator block output provides a uniformly sampled representation of the ideal waveform.

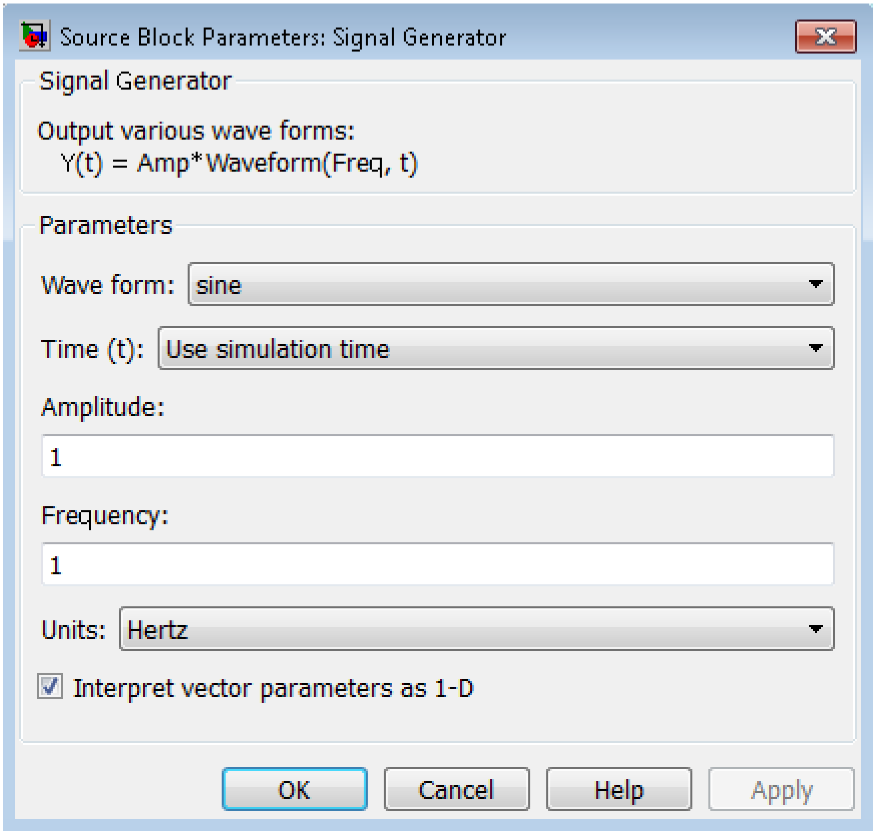
If your model uses a variable-step solver, Simulink might use different step sizes during the simulation. In this case, the Signal Generator block output does not always provide a uniformly sampled representation of the ideal waveform. To ensure that the block output is a uniformly sampled representation, add a Hit Crossing block directly downstream of the Signal Generator block. The following models show the difference in Signal Generator block output with and without the Hit Crossing block.

Model That Uses a Variable-Step Solver	Signal Generator Block Output
 Signal Generator Scope	 Scope 1 0.5 0 -0.5 -1 0 2 4 6 8 10 Time offset: 0
 Signal Generator Hit Crossing Scope	 Scope 1 0.5 0 -0.5 -1 5 0 -5 0 2 4 6 8 10 Time offset: 0

Data Type Support

The Signal Generator block outputs a scalar or array of real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Source Block Parameters: Signal Generator

Signal Generator

Output various wave forms:
Y(t) = Amp*Waveform(Freq, t)

Parameters

Wave form: sine

Time (t): Use simulation time

Amplitude: 1

Frequency: 1

Units: Hertz

☑ Interpret vector parameters as 1-D

Parameters and Dialog Box

Wave form

Specify the wave form: a sine wave, square wave, sawtooth wave, or random wave. The default is a sine wave. This parameter cannot change while a simulation is running.

Time

Specify whether to use simulation time as the source of values for the waveform's time variable or an external signal. If you specify an external time source, the block displays an input port for the time source.

Amplitude

Specify the signal amplitude. The default is `1`.

Frequency

Specify the signal frequency. The default is `1`.

Units

Specify the signal units: Hertz or radians/sec. The default is Hertz.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the **Amplitude** and **Frequency** parameters result in a vector output signal (see "Determining the Output Dimensions of Source Blocks"). This option is not available when an external signal specifies time. In this case, if the **Amplitude** and **Frequency** parameters are column or row matrix values, the output is a 1-D vector.

Examples

The following Simulink examples show how to use the Signal Generator block:

- `sldemo_dblcart1`
- `slexAircraftExample`
- `penddemo`

Characteristics

Sample Time	Continuous
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes

Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Pulse Generator

Purpose

Specify desired dimensions, sample time, data type, numeric type, and other attributes of signal

Library

Signal Attributes

Description

The Signal Specification block allows you to specify the attributes of the signal connected to its input and output ports. If the specified attributes conflict with the attributes specified by the blocks connected to its ports, Simulink software displays an error when it compiles the model. For example, at the beginning of a simulation, if no conflict exists, Simulink eliminates the Signal Specification block from the compiled model. In other words, the Signal Specification block is a virtual block. It exists only to specify the attributes of a signal and plays no role in the simulation of the model.

You can use the Signal Specification block to ensure that the actual attributes of a signal meet desired attributes. For example, suppose that you and a colleague are working on different parts of the same model. You use Signal Specification blocks to connect your part of the model with your colleague's. If your colleague changes the attributes of a signal without informing you, the attributes entering the corresponding Signal Specification block do not match. When you try to simulate the model, you get an error.

You can also use the Signal Specification block to ensure correct propagation of signal attributes throughout a model. The capability of allowing the Simulink to propagate attributes from block to block is powerful. However, if some blocks have unspecified attributes for the signals they accept or output, the model does not have enough information to propagate attributes correctly. For these cases, the Signal Specification block is a good way of providing the information

Simulink needs. Using the Signal Specification block also helps speed up model compilation when blocks are missing signal attributes.

The Signal Specification block supports signal label propagation.

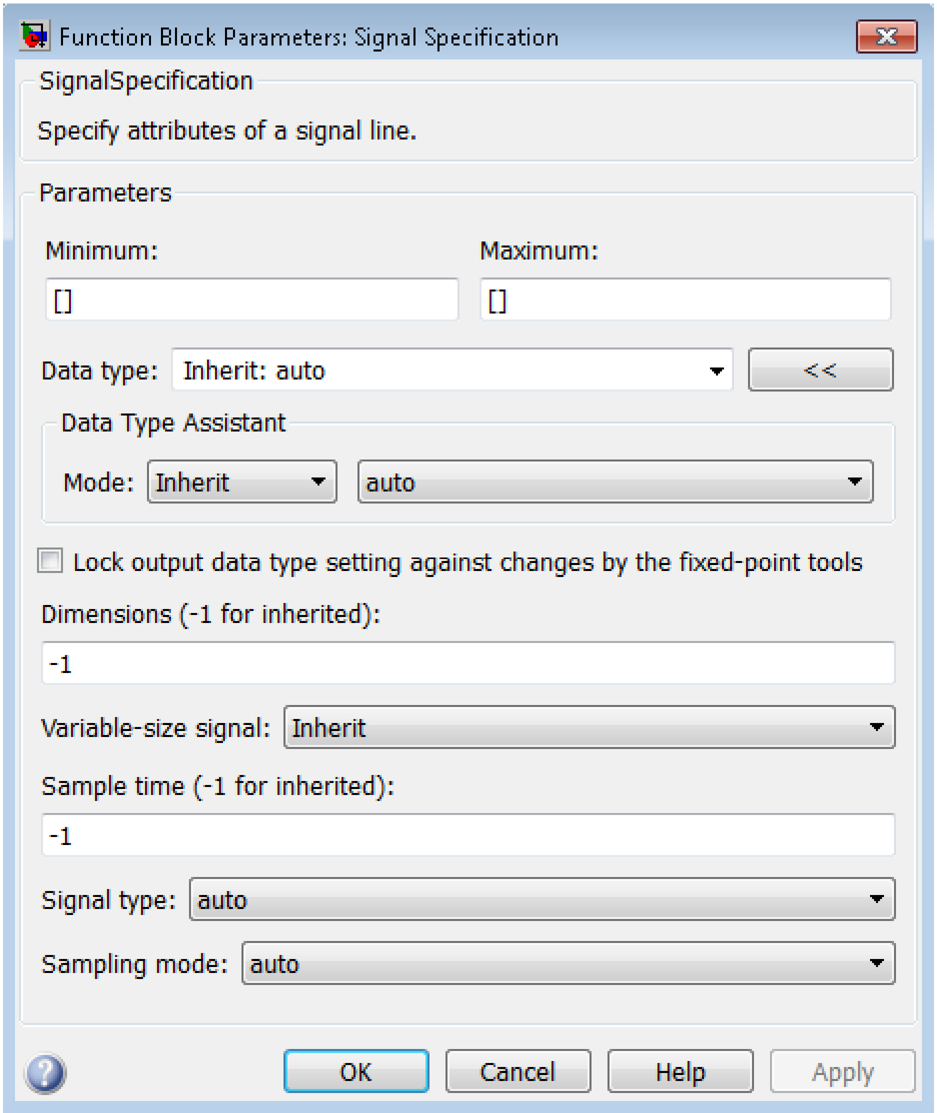
Data Type Support

The Signal Specification block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types, as well as bus objects. The input data type must match the data type specified by the **Data type** parameter.

Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

For more information, see “Data Types Supported by Simulink”.



Function Block Parameters: Signal Specification

SignalSpecification

Specify attributes of a signal line.

Parameters

Minimum: []

Maximum: []

Data type: Inherit: auto

Data Type Assistant

Mode: Inherit auto

☐ Lock output data type setting against changes by the fixed-point tools

Dimensions (-1 for inherited): -1

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Sampling mode: auto

Parameters

and Dialog Box

- “Minimum” on page 1-1561
- “Maximum” on page 1-1562
- “Data type” on page 1-1563
- “Show data type assistant” on page 1-1672
- “Mode” on page 1-1566
- “Data type override” on page 1-1926
- “Signedness” on page 1-1569
- “Word length” on page 1-1570
- “Scaling” on page 1-1736
- “Fraction length” on page 1-1572
- “Slope” on page 1-1573
- “Bias” on page 1-1574
- “Require nonvirtual bus” on page 1-1575
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1721
- “Dimensions (-1 for inherited)” on page 1-1577
- “Variable-size signal” on page 1-1578
- “Sample time (-1 for inherited)” on page 1-1579
- “Signal type” on page 1-1580
- “Sampling mode” on page 1-1581

Minimum

Specify the minimum value for the block output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Command-Line Information

Parameter: `OutMin`
Type: string
Value: Any valid finite real double scalar value
Default: `'[]'`

Maximum

Specify the maximum value for the block output.

Settings

Default:`[]` (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Command-Line Information

Parameter: `OutMax`
Type: string
Value: Any valid finite real double scalar value
Default: `'[]'`

Data type

Specify the output data type.

Settings

Default: `auto`

`Inherit:  auto`
: Inherits the data type.

`double`
: Specifies the data type is `double`.

`single`
: Specifies the data type is `single`.

`int8`
: Specifies the data type is `int8`.

`uint8`
: Specifies the data type is `uint8`.

`int16`
: Specifies the data type is `int16`.

`uint16`
: Specifies the data type is `uint16`.

`int32`
: Specifies the data type is `int32`.

`uint32`
: Specifies the data type is `uint32`.

`boolean`
: Specifies the data type is `boolean`.

`fixdt(1,16,0)`
: Specifies the data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
: Specifies the data type is fixed point `fixdt(1,16,2^0,0)`.

`Enum: <class name>`
Specifies the data type as enumerated.

`Bus: <object name>`
Data type is a bus object.

`<data type expression>`
The name of a data type object, for example `Simulink.NumericType`

Do not specify a bus object as the expression.

Dependency

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Data type** parameters.

Command-Line Information

Parameter: `OutDataTypeStr`
Type: string
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'`

See Also

"Specify Block Output Data Types".

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`
: Specifies inheritance rules for data types. Selecting `Inherit` enables `auto`.

`Built in`
: Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

`Fixed point`
: Specifies fixed-point data types.

`Enumerated`
: Specifies enumerated data types. Selecting `Enumerated` enables you to enter a class name.

`Bus`
: Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or

change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details about the Bus Editor, see “Manage Bus Objects with the Bus Editor”.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Dependency

Clicking the **Show data type assistant** button >> enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specifies fixed-point data as signed.

`Unsigned`
: Specifies the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

"Specifying a Fixed-Point Data Type".

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`

`Binary point`
Specify binary point location.

`Slope and bias`
Enter slope and bias.

`Best precision`
Specify best-precision values.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

"Specifying a Fixed-Point Data Type".

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“Specifying a Fixed-Point Data Type”.

Require nonvirtual bus

If you specify a bus object as the data type, use this parameter to specify whether to accept only nonvirtual bus signals.

Settings

☑

Default: `off`

☐ Off
  Specifies that a signal must come from a *virtual* bus.

☑ On
  Specifies that a signal must come from a *nonvirtual* bus.

Dependencies

The following **Data type** values enable this parameter:

- `Bus:  <object name>`
- `<data type expression>` that specifies a bus object

Command-Line Information

Parameter: `BusOutputAsStruct`
Type: string
Value: `'off' | 'on'`
Default: `'-1off'`

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Dimensions (-1 for inherited)

Specify the dimensions of the input and output signals.

Settings

Default: `-1`

`-1`
: Specifies that signals inherit dimensions.

`n`
: Specifies vector signal of width `n`.

`[m n]`
: Specifies matrix signal having `m` rows and `n` columns.

Command-Line Information

Parameter: `Dimensions`
Type: string
Value: `'-1' | n | [m n]`
Default: `'-1'`

Variable-size signal

Specify a variable-size signal, fixed-size signal, or both.

Settings

Default: `Inherit`

`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.

Command-Line Information

Parameter: `VarSizeSig`
Type: string
Value: `'Inherit' | 'No' | 'Yes'`
Default: `'Inherit'`

See Also

"Variable-Size Signal Basics"

Sample time (-1 for inherited)

Specify the time interval when simulation updates the block.

Settings

Default: `-1`

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

Command-Line Information

Parameter: `SampleTime`
Type: string
Value: Any valid sample time
Default: `'-1'`

See Also

“Specify Sample Time”

Signal type

Specify the numeric type of the input and output signals.

Settings

Default: `auto`

`auto`
  Accepts either `real` or `complex` as the numeric type.

`real`
  Specifies the numeric type as a real number.

`complex`
  Specifies the numeric type as a complex number.

Command-Line Information

Parameter: `SignalType`
Type: string
Value: `'auto'` | `'real'` | `'complex'`
Default: `'auto'`

Sampling mode

Select the sampling mode for this block.

Settings

Default: `auto`

`auto`

Accepts any sampling mode.

`Sample based`

Specifies the output signal to be sample-based.

`Frame based`

Specifies the output signal to be frame-based.

Tip

To generate frame-based signals, you must have the DSP System Toolbox product installed.

Command-Line Information

Parameter: `SamplingMode`
Type: string
Value: `'auto'` | `'Sample based'` | `'Frame based'`
Default: `'auto'`

Bus Support

The Signal Specification block supports virtual and nonvirtual buses. If you specify a bus object as the data type, then set these other block parameters as follows:

Block Parameter	Required Value for a Bus Data Type
Variable-size signal	`No`
Sampling mode	`Sample based`

All elements of the bus input to a Signal Specification block must have the same names as specified in the bus object.

All signals in a nonvirtual bus input to a Signal Specification block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “About Composite Signals” and Bus-Capable Blocks for more information.

The **Model Configuration Parameters > Diagnostics > Connectivity** “Mux blocks used to create bus signals” diagnostic must be set to `Error`.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified by the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Virtual	Yes For more information, see “Virtual Blocks”.
Zero-Crossing Detection	No

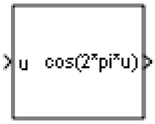
Purpose

Implement fixed-point sine or cosine wave using lookup table approach that exploits quarter wave symmetry

Library

Lookup Tables (Sine block or Cosine block)

Description



The Sine and Cosine block implements a sine and/or cosine wave in fixed point using a lookup table method that exploits quarter wave symmetry.

The Sine and Cosine block can output the following functions of the input signal, depending upon what you select for the **Output formula** parameter:

- $\sin(2\pi u)$
- $\cos(2\pi u)$
- $\exp(i2\pi u)$
- $\sin(2\pi u)$ and $\cos(2\pi u)$

You define the number of lookup table points in the **Number of data points for lookup table** parameter. The block implementation is most efficient when you specify the lookup table data points to be `(2^n)+1`, where *n* is an integer.

Tip To obtain meaningful block output, the block input values should fall within the range [0, 1). For input values that fall outside this range, the values are cast to an unsigned data type, where overflows wrap. For these out-of-range inputs, the block output might not be meaningful.

Use the **Output word length** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

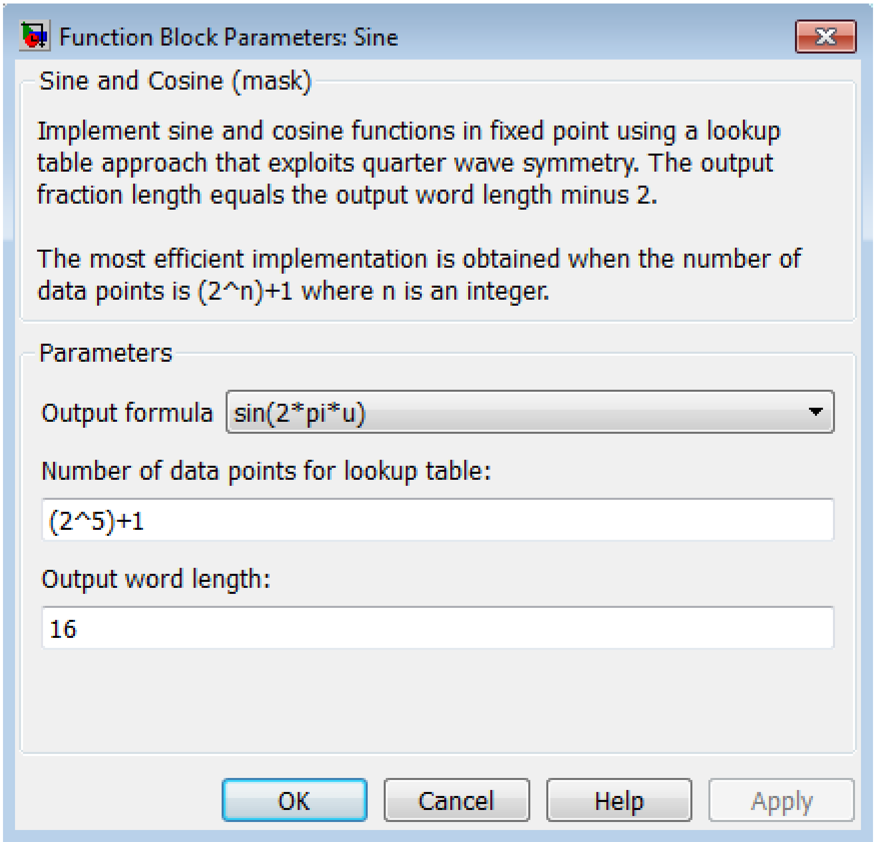
Data Type Support

The Sine and Cosine block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The output of the block is a fixed-point data type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Sine

Sine and Cosine (mask)

Implement sine and cosine functions in fixed point using a lookup table approach that exploits quarter wave symmetry. The output fraction length equals the output word length minus 2.

The most efficient implementation is obtained when the number of data points is (2^n)+1 where n is an integer.

Parameters

Output formula: sin(2*pi*u)

Number of data points for lookup table: (2^5)+1

Output word length: 16

Parameters and Dialog Box

Output formula

Select the signal(s) to output.

Number of data points for lookup table

Specify the number of data points to retrieve from the lookup table. The implementation is most efficient when you specify the lookup table data points to be `(2^n)+1`, where *n* is an integer.

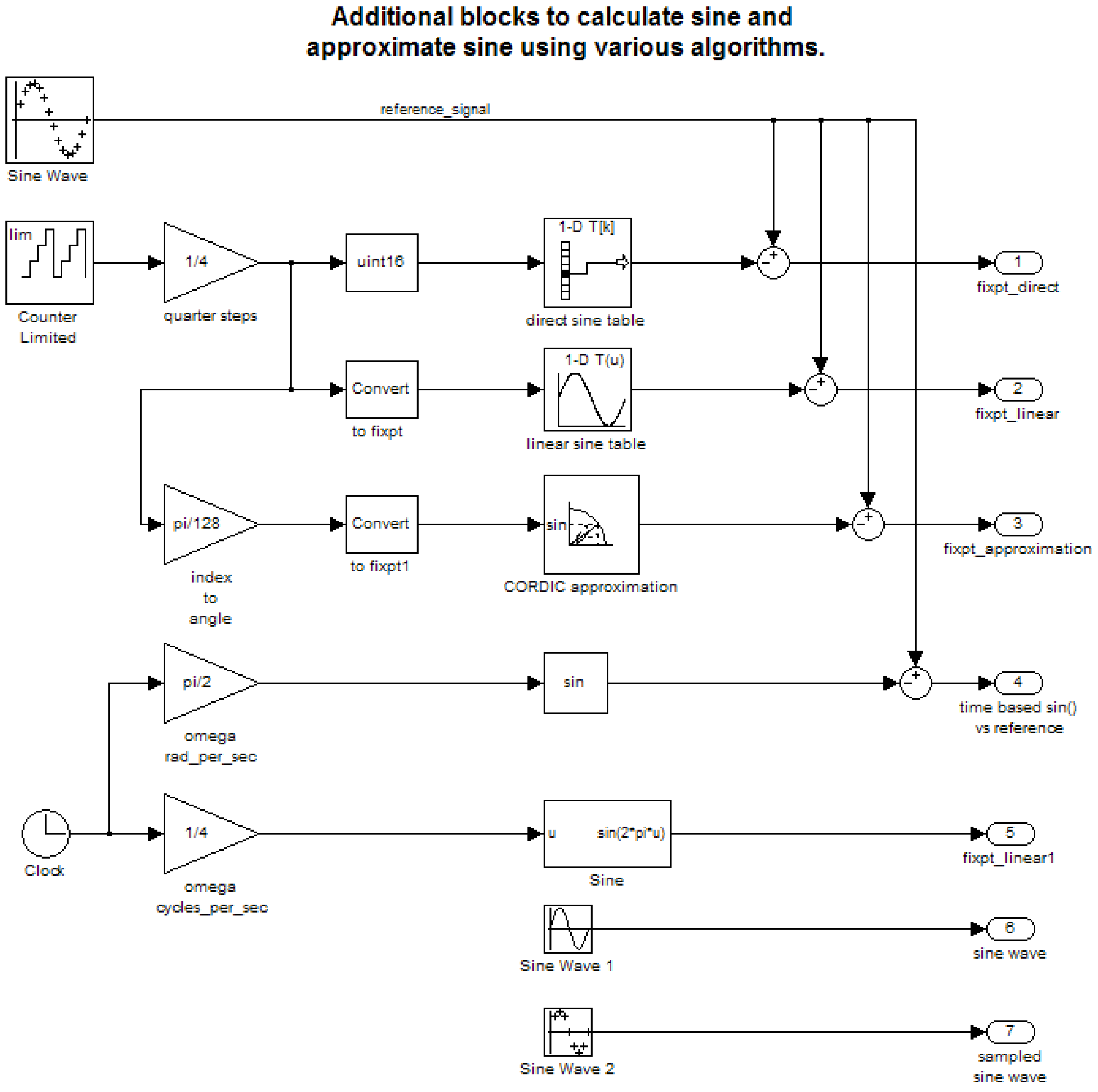
Output word length

Specify the word length for the fixed-point data type of the output signal. The fraction length of the output is the output word length minus 2.

Note The block uses double-precision floating-point values to construct lookup tables. Therefore, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length to values greater than 53 bits does not improve the precision of your output.

Examples

The `sldemo_tonegen_fixpt` model shows how you can use the Sine block to implement a fixed-point sine wave.


Additional blocks to calculate sine and approximate sine using various algorithms.
reference_signal
Sine Wave
lim
Counter Limited
1/4
quarter steps
uint16
1-D T[k]
direct sine table
1
fixpt_direct
Convert
to fixpt
1-D T(u)
linear sine table
2
fixpt_linear
pi/128
index to angle
Convert
to fixpt1
sin
CORDIC approximation
3
fixpt_approximation
pi/2
omega rad_per_sec
sin
4
time based sin() vs reference
Clock
1/4
omega cycles_per_sec
u
sin(2*pi*u)
Sine
5
fixpt_linear1
Sine Wave 1
6
sine wave
Sine Wave 2
7
sampled sine wave

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	N/A
Zero-Crossing Detection	No

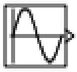
See Also

Sine Wave, Trigonometric Function

Purpose Generate sine wave, using simulation time as time source

Library Sources

Description The Sine Wave block outputs a sinusoidal waveform. The block can operate in time-based or sample-based mode.



Note This block is the same as the Sine Wave Function block that appears in the Math Operations library. If you select `Use external signal` for the **Time** parameter in the block dialog box, you get the Sine Wave Function block.

Time-Based Mode

The output of the Sine Wave block is determined by:

$$y = amplitude \times \sin(frequency \times time + phase) + bias.$$

Time-based mode has two submodes: continuous mode or discrete mode. The value of the **Sample time** parameter determines whether the block operates in continuous mode or discrete mode:

- `0` (the default) causes the block to operate in continuous mode.
- `>0` causes the block to operate in discrete mode.

See "Specify Sample Time" in the online documentation for more information.

Block Behavior in Continuous Mode

A **Sample time** parameter value of `0` causes the block to operate in continuous mode. When operating in continuous mode, the Sine Wave block can become inaccurate due to loss of precision as time becomes very large.

Block Behavior in Discrete Mode

A **Sample time** parameter value greater than zero causes the block to behave as if it were driving a Zero-Order Hold block whose sample time is set to that value.

Using the Sine Wave block in this way, you can build models with sine wave sources that are purely discrete, rather than models that are hybrid continuous/discrete systems. Hybrid systems are inherently more complex and as a result take more time to simulate.

In discrete mode, this block uses a differential incremental algorithm instead of one based on absolute time. As a result, the block can be useful in models intended to run for an indefinite length of time, such as in vibration or fatigue testing.

The differential incremental algorithm computes the sine based on the value computed at the previous sample time. This method uses the following trigonometric identities:

$$\sin(t+\Delta t) = \sin(t)\cos(\Delta t) + \sin(\Delta t)\cos(t)$$
$$\cos(t+\Delta t) = \cos(t)\cos(\Delta t) - \sin(t)\sin(\Delta t)$$

In matrix form, these identities are:

$$\begin{bmatrix} \sin(t+\Delta t) \\ \cos(t+\Delta t) \end{bmatrix} = \begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix} \begin{bmatrix} \sin(t) \\ \cos(t) \end{bmatrix}$$

Because Δt is constant, the following expression is a constant:

$$\begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix}$$

Therefore, the problem becomes one of a matrix multiplication of the value of $\sin(t)$ by a constant matrix to obtain $\sin(t+\Delta t)$.

Discrete mode reduces but does not eliminate the accumulation of round-off errors, for example, `(4*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.

Methods to Handle Round-Off Errors in Discrete Mode

To handle round-off errors when the Sine Wave block operates in time-based discrete mode, use one of the following methods.

Method	Rationale
Insert a Saturation block directly downstream of the Sine Wave block.	By setting saturation limits on the Sine Wave block output, you can remove overshoot due to accumulation of round-off errors.
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.	Unlike the block algorithm, 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 the following formula to compute the output of the Sine Wave block.

$$y = A\sin(2\pi(k+o)/p)+b$$

where

- A is the amplitude of the sine wave.
- p is the number of time samples per sine wave period.
- k is a repeating integer value that ranges from 0 to p–1.
- o is the offset (phase shift) of the signal.
- b is the signal bias.

In this mode, Simulink sets k equal to `0` at the first time step and computes the block output, using the preceding formula. At the next time step, Simulink increments k and recomputes the output of the block. When k reaches p, Simulink resets k to `0` before computing the block output. This process continues until the end of the simulation.

The sample-based method of computing the block output does not depend on the result of the previous time step to compute the result at the current time step. Therefore, this mode avoids the accumulation of round-off errors. However, this mode has one potential drawback. If the Sine Wave block is in a conditionally-executed subsystem that pauses and then resumes execution, the block output might not stay in sync with the rest of the simulation. If the accuracy of your model requires that the output of conditionally-executed Sine Wave blocks remain in sync with the rest of the model, use time-based mode for computing the output of the conditionally-executed blocks.

Parameter Dimensions

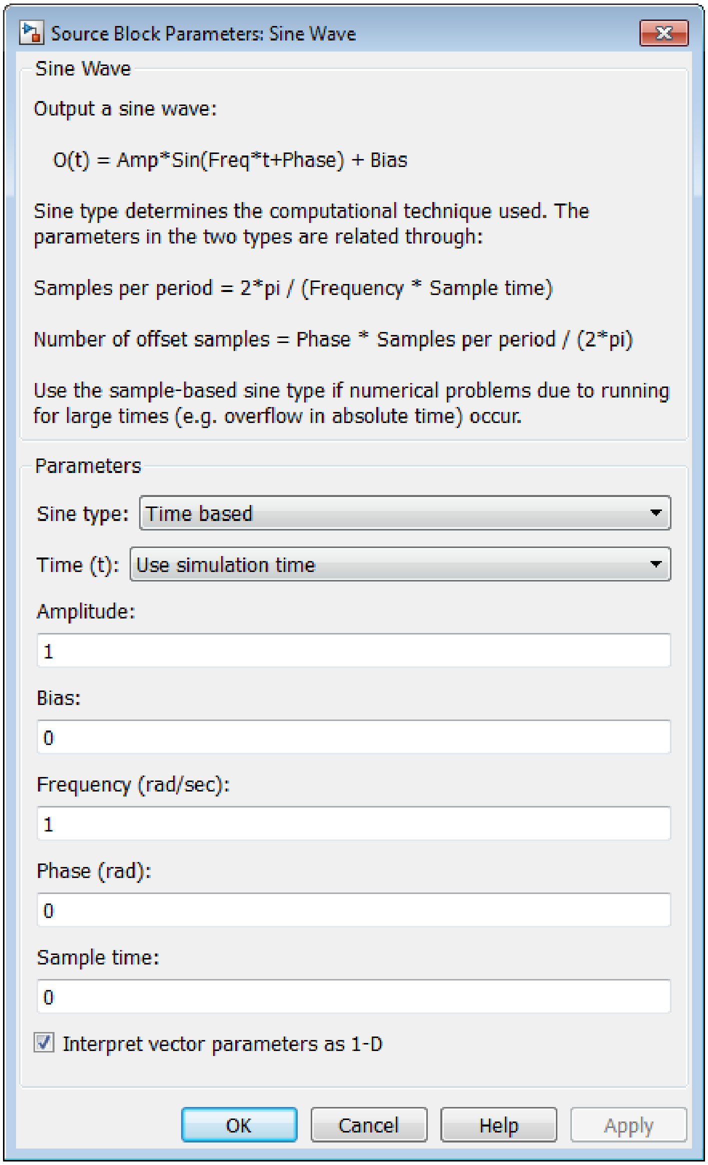
The numeric parameters of this block must have the same dimensions after scalar expansion.

- If **Interpret vector parameters as 1-D** is not selected, the block outputs a signal of the same dimensions and dimensionality as the parameters.
- If **Interpret vector parameters as 1-D** is selected and the numeric parameters are row or column vectors, the block outputs a vector signal. Otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support

The Sine Wave block accepts and outputs real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Sine Wave

Output a sine wave:

O(t) = Amp*Sin(Freq*t+Phase) + Bias

Sine type determines the computational technique used. The parameters in the two types are related through:

Samples per period = 2*pi / (Frequency * Sample time)

Number of offset samples = Phase * Samples per period / (2*pi)

Use the sample-based sine type if numerical problems due to running for large times (e.g. overflow in absolute time) occur.

Parameters

Sine type: Time based

Time (t): Use simulation time

Amplitude: 1

Bias: 0

Frequency (rad/sec): 1

Phase (rad): 0

Sample time: 0

☑ Interpret vector parameters as 1-D

Parameters

and Dialog Box

Sine type
Specify the type of sine wave that this block generates, either time- or sample-based. Some parameters in the dialog box appear depending on whether you select time-based or sample-based.

Time
Specify whether to use simulation time as the source of values for the time variable or an external source. If you specify an external time source, the block displays an input port for the time source.

Amplitude
Specify the amplitude of the signal. The default is `1`.

Bias
Specify the constant value added to the sine to produce the output of this block.

Frequency
Specify the frequency, in radians per second. The default is `1`. This parameter appears only when you set **Sine type** to time-based.

Samples per period
Specify the number of samples per period. This parameter appears only when you set **Sine type** to sample-based.

Phase
Specify the phase shift, in radians. The default is `0`. This parameter appears only when you set **Sine type** to time-based.

Number of offset samples
Specify the offset (discrete phase shift) in number of sample times. This parameter appears only when you set **Sine type** to sample-based.

Sample time
Specify the sample period. The default is `0`. If the sine type is sample-based, the sample time must be greater than 0. See "Specify Sample Time" in the online documentation for more information.

Interpret vector parameters as 1-D

If selected, column or row matrix values for numeric parameters result in a vector output signal. Otherwise, the block outputs a signal of the same dimensionality as the parameters. If you do not select this check box, the block always outputs a signal of the same dimensionality as the numeric parameters. See "Determining the Output Dimensions of Source Blocks" in the Simulink documentation. This parameter is not available when an external signal specifies time. In this case, if numeric parameters are column or row matrix values, the output is a 1-D vector.

Examples

The following Simulink examples show how to use the Sine Wave block:

- `sldemo_househeat`
- `sldemo_tonegen_fixpt`
- `sldemo_VariableTransportDelay`
- `sldemo_zeroxing`

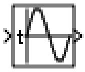
Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes
Zero-Crossing Detection	No

Purpose Generate sine wave, using external signal as time source

Library Math Operations

Description This block is the same as the Sine Wave block that appears in the Sources library. If you select `Use simulation time` for the **Time** parameter in the block dialog box, you get the Sine Wave block. See the documentation for the Sine Wave block for more information.



Slider Gain

Purpose

Vary scalar gain using slider

Library

Math Operations

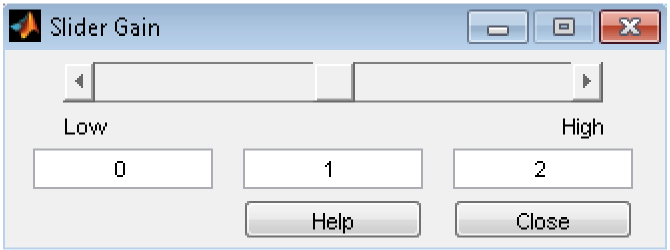
Description

Use the Slider Gain block to vary a scalar gain during a simulation using a slider. The block accepts one input and generates one output.

Data Type Support

Data type support for the Slider Gain block is the same as that for the Gain block (see Gain).

Parameters and Dialog Box



Low

Specify the lower limit of the slider range. The default is `0`.

High

Specify the upper limit of the slider range. The default is `2`.

The edit fields indicate (from left to right) the lower limit, the current value, and the upper limit. You can change the gain in two ways: by manipulating the slider, or by entering a new value in the current value field. You can change the range of gain values by changing the lower and upper limits. Close the dialog box by clicking the **Close** button.

If you click the left or right arrow of the slider, the current value changes by about 1% of the slider range. If you click the rectangular area to either side of the slider's indicator, the current value changes by about 10% of the slider range.

To apply a vector or matrix gain to the block input, consider using the Gain block.

Examples

The following example models show how to use the Slider Gain block:

- `aero_six_dof`

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	Yes, of the gain
States	0
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

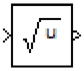
Gain

Sqrt, Signed Sqrt, Reciprocal Sqrt

Purpose Calculate square root, signed square root, or reciprocal of square root

Library Math Operations

Description You can select one of the following functions from the **Function** parameter list.



Function	Description	Mathematical Expression	MATLAB Equivalent
`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	$\mathrm{sign}(u)*\|u\|^{0.5}$	—
`rSqrt`	Reciprocal of the square root of the input	$u^{-0.5}$	—

The block output is the result of applying the function to the input. Each function supports:

- Scalar operations
- Element-wise vector and matrix operations

Data Type Support

The block accepts input signals of the following data types:

Function	Input Data Types	Restrictions
`sqrt`	• Floating point • Built-in integer • Fixed point	None
`signedSqrt`	• Floating point • Built-in integer • Fixed point	When the input is an integer or fixed-point type, the output must be floating point.
`rSqrt`	• Floating point • Built-in integer • Fixed point	None

The block accepts real and complex inputs of the following types:

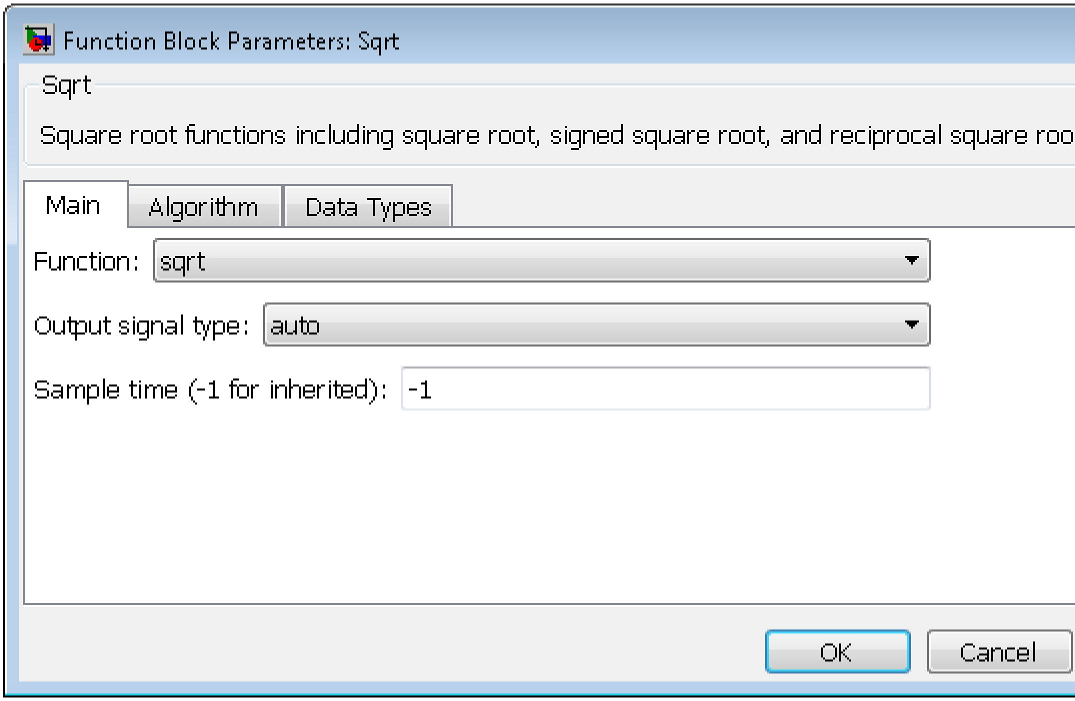
Function	Types of Real Inputs	Types of Complex Inputs
`sqrt`	Any, except for fixed-point inputs that are negative or have nontrivial slope and nonzero bias	Any, except for fixed-point inputs
`signedSqrt`		None
`rSqrt`		None

The block output:

- Uses the data type that you specify for **Output data type**
- Is real or complex, depending on your selection for **Output signal type**

Parameters and Dialog Box

The **Main** pane of the block dialog box appears as follows:



Function Block Parameters: Sqrt

Sqrt

Square root functions including square root, signed square root, and reciprocal square roo

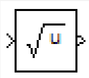
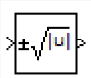
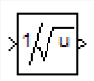
Function: sqrt

Output signal type: auto

Sample time (-1 for inherited): -1

Function

Specify the mathematical function. The block icon changes to match the function you select.

Function	**Block Icon**
`sqrt`	
`signedSqrt`	
`rSqrt`	

Output signal type

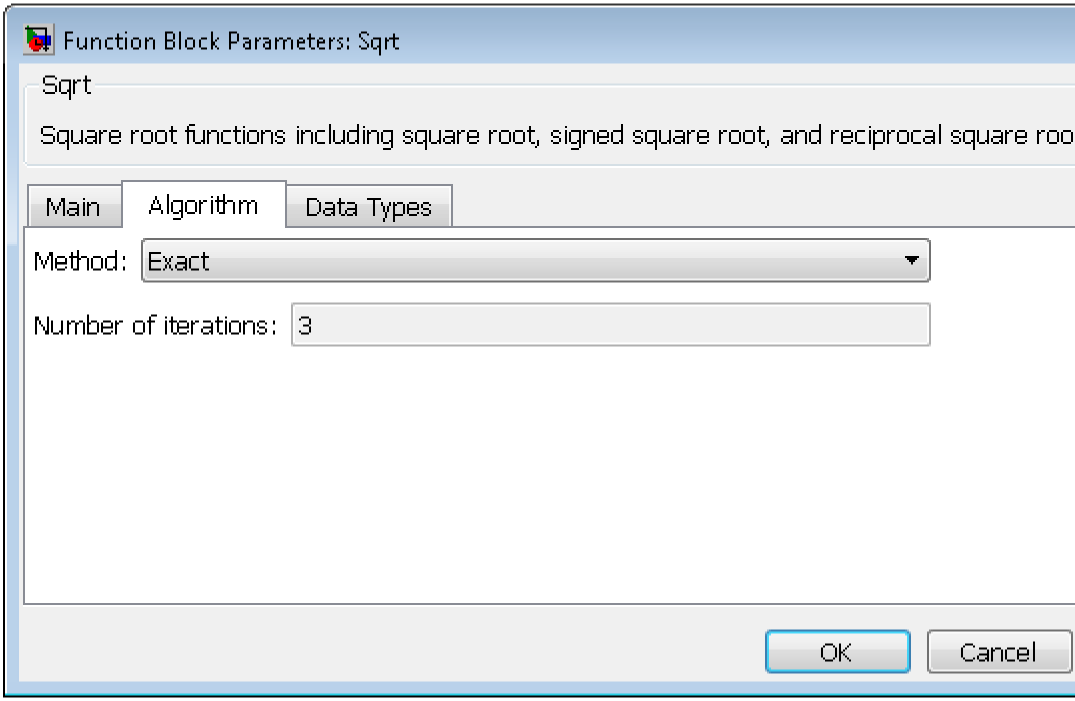
Specify the output signal type of the block as `auto`, `real`, or `complex`.

Function	**Input Signal Type**	**Output Signal Type**		
		Auto	**Real**	**Complex**
`sqrt`	`real`	`real` for nonnegative inputs `NaN` for negative inputs	`real` for nonnegative inputs `NaN` for negative inputs	`complex`
	`complex`	`complex`	`error`	`complex`
`signedSqrt`	`real`	`real`	`real`	`complex`
	`complex`	`error`	`error`	`error`
`rSqrt`	`real`	`real`	`real`	`error`
	`complex`	`error`	`error`	`error`

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

The **Algorithm** pane of the block dialog box appears as follows:



Function Block Parameters: Sqrt
Sqrt
Square root functions including square root, signed square root, and reciprocal square roo
Main
Algorithm
Data Types

Method: Exact

Number of iterations: 3

OK
Cancel

Note The parameters in the **Algorithm** pane are available only when you set **Function** to `rSqrt` on the **Main** pane.

Method

Specify the method for computing the reciprocal of a square root.

Method	Data Types Supported	When to Use This Method
`Exact`	Floating point If you use a fixed-point or built-in integer type, an upcast to a floating-point type occurs.	You do not want an approximation. **Note** The input or output must be floating point.
`Newton-Raphson`	Floating-point, fixed-point, and built-in integer types	You want a fast, approximate calculation.

The `Exact` method provides results that are consistent with MATLAB computations.

Note The algorithms for `sqrt` and `signedSqrt` are always of `Exact` type, no matter what selection appears on the block dialog box.

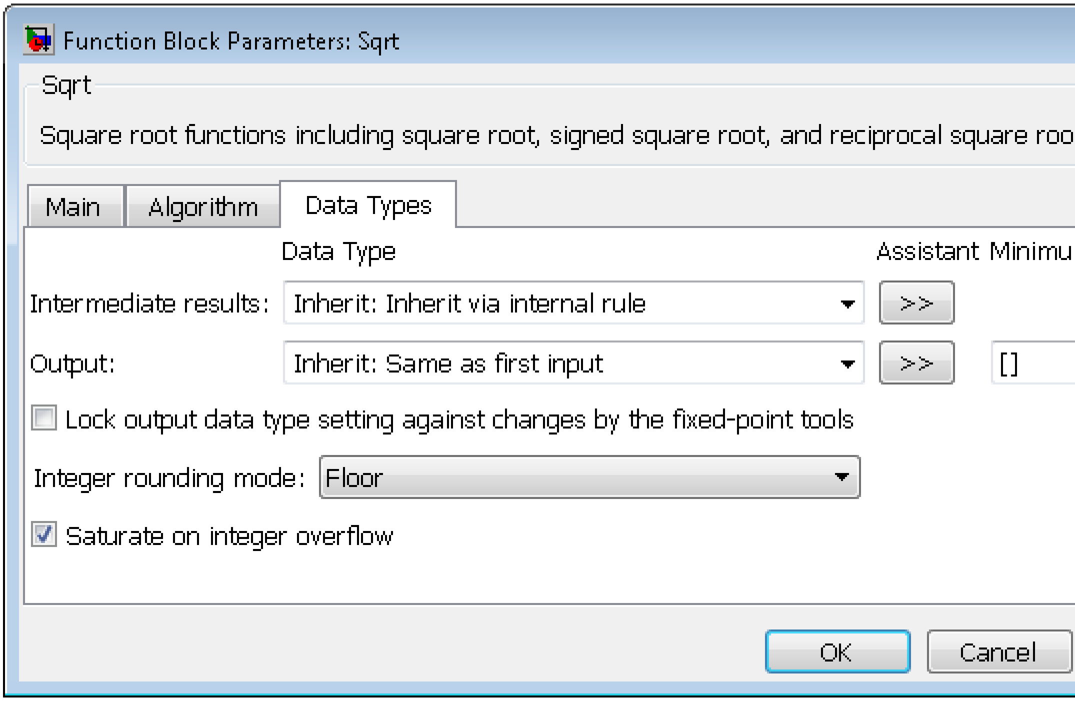
Number of iterations

Specify the number of iterations to perform the Newton-Raphson algorithm. The default value is 3.

This parameter is not available when you select `Exact` for **Method**.

Note If you enter 0, the block output is the initial guess of the Newton-Raphson algorithm.

The **Data Types** pane of the block dialog box appears as follows:



Function Block Parameters: Sqrt

Sqrt

Square root functions including square root, signed square root, and reciprocal square roo

Intermediate results: Inherit: Inherit via internal rule

Output: Inherit: Same as first input []

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☑ Saturate on integer overflow

Intermediate results data type

Specify the data type for intermediate results (available only when you set **Function** to `sqrt` or `rSqrt` on the **Main** pane). You can set the data type to:

- A rule that inherits a data type, for example, `Inherit:Inherit via internal rule`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Follow these guidelines on setting an intermediate data type explicitly for the square root function, `sqrt`:

Input and Output Data Types	Intermediate Data Type
Input or output is double.	Use double.
Input or output is single, and any non-single data type is *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.

Output data type

Specify the output data type. You can set the data type to:

- A rule that inherits a data type, for example, `Inherit:Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in the *Simulink User's Guide* for more information.

Output minimum

Specify the minimum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Use Lock Output Data Type Setting".

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see "Rounding". in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit	Overflows saturate to either the minimum or maximum value	An overflow associated with a signed 8-bit

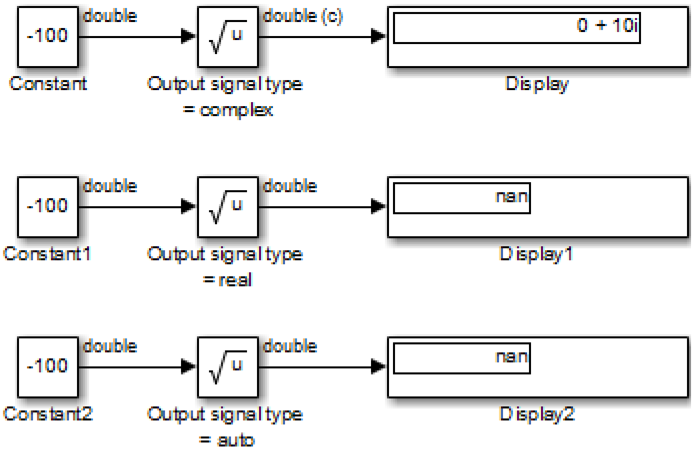
Action	Reasons for Taking This Action	What Happens for Overflows	Example
	saturation protection in the generated code.	that the data type can represent.	integer can saturate to –128 or 127.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking 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.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

sqrt Function

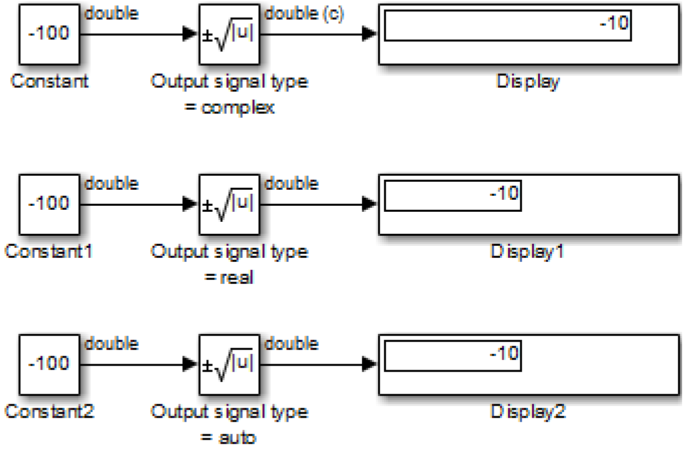
Suppose that you have the following model:


-100
double
Constant
√u
Output signal type = complex
double (c)
0 + 10i
Display
-100
double
Constant1
√u
Output signal type = real
double
nan
Display1
-100
double
Constant2
√u
Output signal type = auto
double
nan
Display2

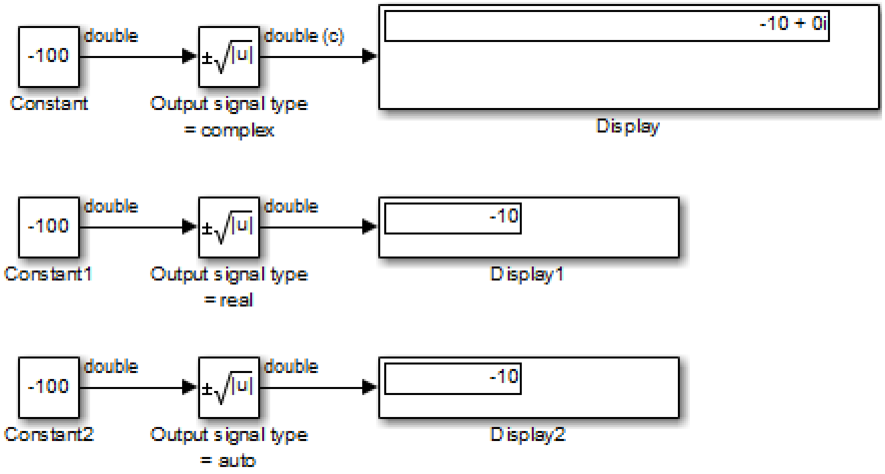
When the input to the Sqrt block is negative and the **Output signal type** is `auto` or `real`, the `sqrt` function outputs `NaN`. However, setting **Output signal type** to `complex` produces the correct answer.

signedSqrt Function

Suppose that you have the following model:

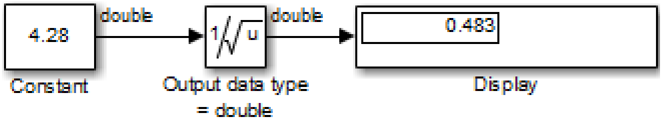

-100
double
±√|u|
double (c)
-10
Constant
Output signal type
= complex
Display
-100
double
±√|u|
double
-10
Constant1
Output signal type
= real
Display1
-100
double
±√|u|
double
-10
Constant2
Output signal type
= auto
Display2

When the input to the Sqrt block is negative, the block output is the same for any **Output signal type** setting. If you change the first Display block format from `short` to `decimal (Stored Integer)`, you see the value of the imaginary part for the complex output.


-100
double
Constant
±√|u|
Output signal type = complex
double (c)
-10 + 0i
Display
-100
double
Constant1
±√|u|
Output signal type = real
double
-10
Display1
-100
double
Constant2
±√|u|
Output signal type = auto
double
-10
Display2

rSqrt Function with Floating-Point Inputs

Suppose that you have the following model:


4.28
double
Constant
1/√u
Output data type = double
double
0.483
Display

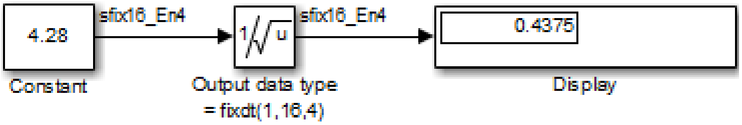
In the Sqrt block dialog box, assume that the following parameter settings apply:

Parameter	Setting
Method	`Newton-Raphson`
Number of iterations	`1`
Intermediate results data type	`Inherit:Inherit from input`

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0004 of the final value (0.4834).

rSqrt Function with Fixed-Point Inputs

Suppose that you have the following model:



In the Sqrt block dialog box, assume that the following parameter settings apply:

Parameter	Setting
Method	`Newton-Raphson`
Number of iterations	`1`
Intermediate results data type	`Inherit:Inherit from input`

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0459 of the final value (0.4834).

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Math Function, Trigonometric Function

Purpose

Remove singleton dimensions from multidimensional signal

Library

Math Operations

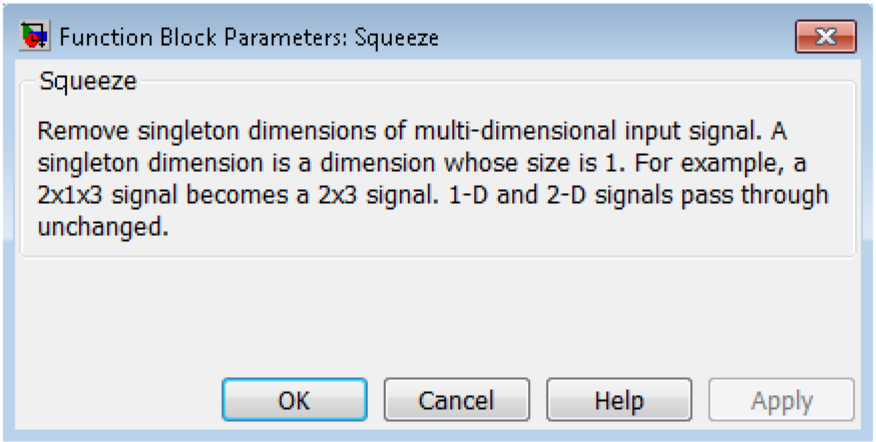
Description

The Squeeze block removes singleton dimensions from its multidimensional input signal. A singleton dimension is any dimension whose size is one. The Squeeze block operates only on signals whose number of dimensions is greater than two. Scalar, one-dimensional (vector), and two-dimensional (matrix) signals pass through the Squeeze block unchanged.

Data Type Support

The Squeeze block accepts input signals of any dimension and of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

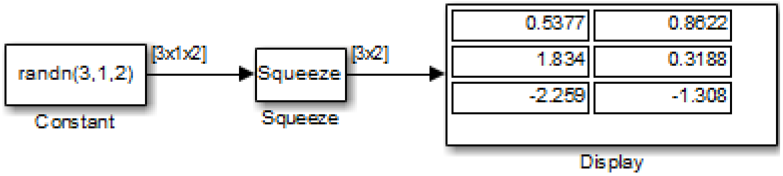


Squeeze

Remove singleton dimensions of multi-dimensional input signal. A singleton dimension is a dimension whose size is 1. For example, a 2x1x3 signal becomes a 2x3 signal. 1-D and 2-D signals pass through unchanged.

Examples

In the following model, the Squeeze block converts a multidimensional array of size 3-by-1-by-2 into a 3-by-2 signal:


randn(3,1,2)
Constant
[3x1x2]
Squeeze
Squeeze
[3x2]
0.5377
0.8622
1.834
0.3188
-2.259
-1.308
Display

Because the Constant block supplies a signal with random values to the Squeeze block, the values in the Display block vary from simulation to simulation.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	N/A
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

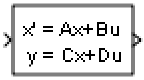
See Also

Reshape

Purpose Implement linear state-space system

Library Continuous

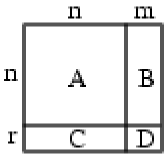
Description The State-Space block implements a system whose behavior you define as



$$\dot{x} = Ax + Bu$$
$$y = Cx + Du,$$

where x is the state vector, u is the input vector, and y is the output 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.

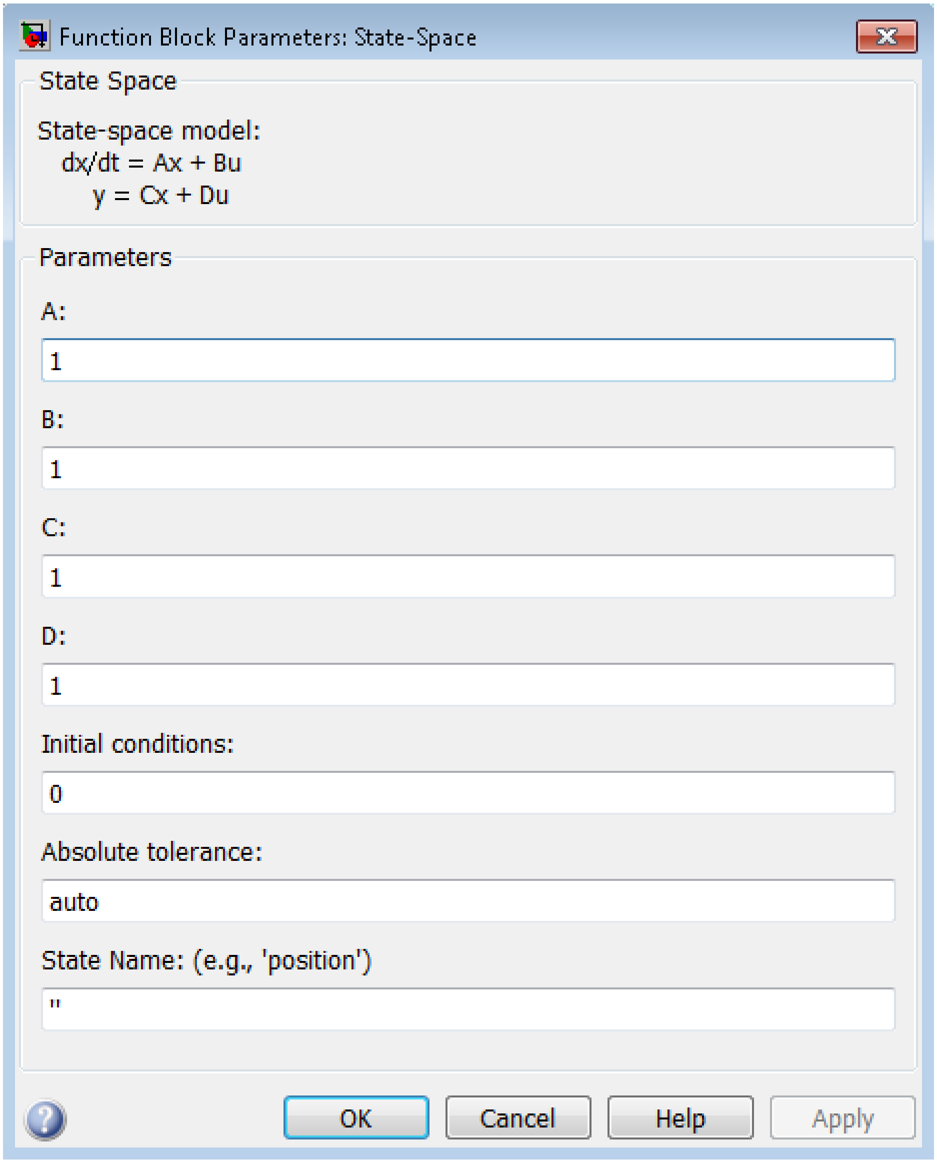


The block accepts one input and generates one output. The input vector width depends on the number of columns in the **B** and **D** matrices. The output vector width depends on the number of rows in the **C** and **D** matrices.

Simulink software converts a matrix containing zeros to a sparse matrix for efficient multiplication.

Data Type Support A State-Space block accepts and outputs real signals of type `double`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.


Function Block Parameters: State-Space
State Space
Parameters
OK
Cancel
Help
Apply

State-space model:
dx/dt = Ax + Bu
y = Cx + Du

A: 1

B: 1

C: 1

D: 1

Initial conditions: 0

Absolute tolerance: auto

State Name: (e.g., 'position') ''

Parameters and Dialog Box

A

Specify the n-by-n matrix coefficient, where n is the number of states.

Settings

Default: 1

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

B

Specify the n-by-m matrix coefficient, where n is the number of states and m is the number of inputs.

Settings

Default: `1`

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

C

Specify the r-by-n matrix coefficient, where r is the number of outputs and n is the number of states.

Settings

Default: 1

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

D

Specify the r-by-m matrix coefficient, where r is the number of outputs and m is the number of inputs.

Settings

Default: 1

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Initial conditions

Specify the initial state vector.

Settings

Default: `0`

The initial conditions of this block cannot be `inf` or `NaN`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, `1`, a real scalar, or a real 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

State Name (e.g., 'position')

Assign a unique name to each state.

Settings

Default: `' '`

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 string, cell array, or structure.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Examples

The following Simulink examples show how to use the State-Space block:

- `sldemo_dblcart1`

- `aero_vibrati`

Characteristics

Direct Feedthrough	Only if **D** ≠ 0
Sample Time	Continuous
Scalar Expansion	Yes, of the initial conditions
States	Depends on the size of **A**
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also

Discrete State-Space

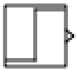
Purpose

Generate step function

Library

Sources

Description



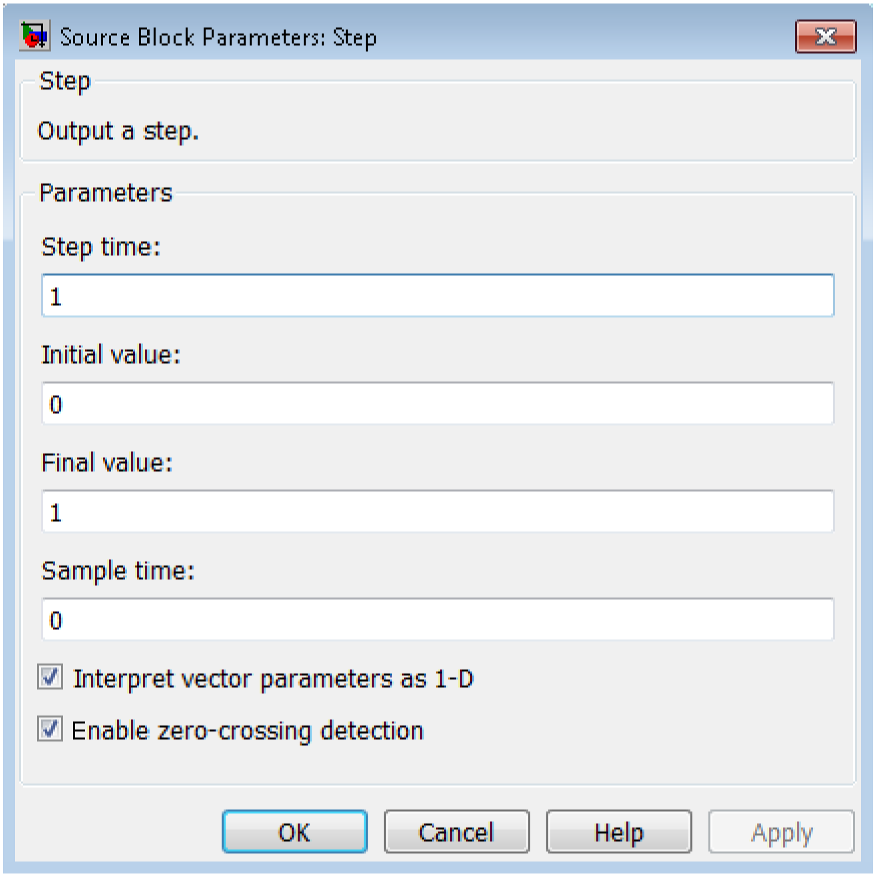
The Step block provides a step between two definable levels at a specified time. If the simulation time is less than the **Step time** parameter value, the block's output is the **Initial value** parameter value. For simulation time greater than or equal to the **Step time**, the output is the **Final value** parameter value.

The numeric block parameters must be of the same dimensions after scalar expansion. If the **Interpret vector parameters as 1-D** option is off, the block outputs a signal of the same dimensions and dimensionality as the parameters. If the **Interpret vector parameters as 1-D** option is on and the numeric parameters are row or column vectors (that is, single row or column 2-D arrays), the block outputs a vector (1-D array) signal. Otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support

The Step block outputs real signals of type `double`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.


Source Block Parameters: Step
OK
Cancel
Help
Apply

Step

Output a step.

Parameters

Step time: 1

Initial value: 0

Final value: 1

Sample time: 0

☑ Interpret vector parameters as 1-D

☑ Enable zero-crossing detection

Parameters and Dialog Box

Step time
Specify the time, in seconds, when the output jumps from the **Initial value** parameter to the **Final value** parameter. The default is `1` second.

Initial value
Specify the block output until the simulation time reaches the **Step time** parameter. The default is `0`.

Final value

Specify the block output when the simulation time reaches and exceeds the **Step time** parameter. The default is `1`.

Sample time

Specify the sample rate of step. See “Specify Sample Time” in the online documentation for more information.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the Step block’s numeric parameters result in a vector output signal; otherwise, the block outputs a signal of the same dimensionality as the parameters. If this option is not selected, the block always outputs a signal of the same dimensionality as the block’s numeric parameters. See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Step block:

- `sldemo_doublebounce`
- `sldemo_enginewc`

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled.

See Also

Ramp

Stop Simulation

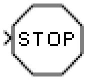
Purpose

Stop simulation when input is nonzero

Library

Sinks

Description



The Stop Simulation block stops the simulation when the input is nonzero. The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop.

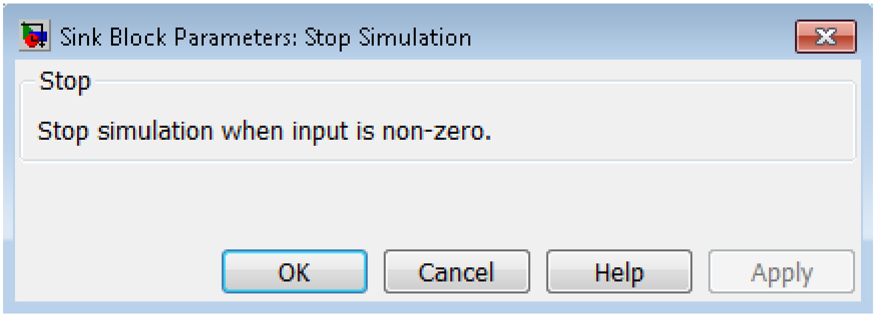
When you use the Stop Simulation block in a For Iterator subsystem, the stop action occurs after execution of *all* the iterations in the subsystem during a time step. The stop action does not interrupt execution until the start of the next time step.

You cannot use the Stop Simulation block to pause the simulation. To create a block that pauses the simulation, see “Creating Pause Blocks” in the Simulink documentation.

Data Type Support

The Stop Simulation block accepts real signals of type `double` or `Boolean`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Sink Block Parameters: Stop Simulation

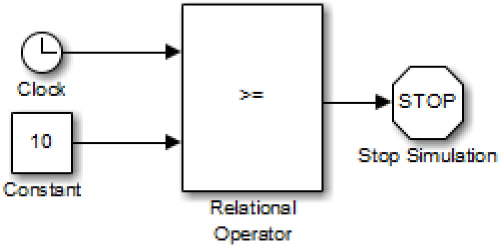
Stop

Stop simulation when input is non-zero.

Examples

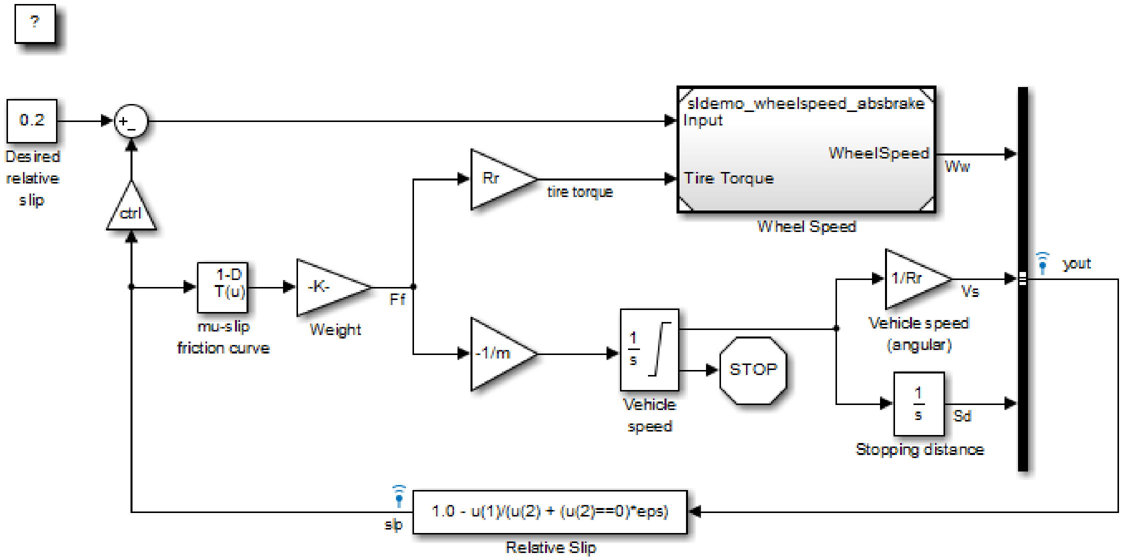
Usage with the Relational Operator Block

You can use the Stop Simulation block with the Relational Operator block to control when a simulation stops. For example, the following model stops simulation when the simulation time reaches 10.


Clock
10
Constant
>=
Relational Operator
STOP
Stop Simulation

Usage with the Integrator Block

You can use the Stop Simulation block with the Integrator block to control when a simulation stops. For example, the `sldemo_absbrake` model stops simulation when the saturation port of the Integrator block outputs a value of 1 or –1.


?
0.2
Desired relative slip
+
-
ctrl
1-D T(u)
mu-slip friction curve
-K-
Weight
Ff
Rr
tire torque
-1/m
1/s
Vehicle speed
STOP
sldemo_wheelspeed_absbrake
Input
WheelSpeed
Tire Torque
Wheel Speed
Ww
1/Rr
Vs
Vehicle speed (angular)
1/s
Sd
Stopping distance
yout
1.0 - u(1)/(u(2) + (u(2)==0)*eps)
Relative Slip
slp

Characteristics

Sample Time	Inherited from driving block
Dimensionalized	Yes
Zero-Crossing Detection	No

Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

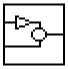
Purpose

Represent system within another system

Library

Ports & Subsystems

Description



A subsystem block contains a subset of blocks or code within an overall model or system. The subsystem block can represent a virtual subsystem or a nonvirtual subsystem.

In nonvirtual subsystems, you can control when the contents of the subsystem are evaluated. Nonvirtual subsystems are executed as a single unit (atomic execution). You can create conditionally executed nonvirtual subsystems that execute only when a transition occurs on a triggering, function-call, action, or enabling input (see “Conditional Subsystems”).

A subsystem is virtual if the block is neither conditionally executed nor atomic.

Tip To determine if a subsystem is virtual, use the `get_param` function for the Boolean block parameter `IsSubsystemVirtual`.

An Atomic Subsystem block is a subsystem block in which **Treat as atomic unit** is selected by default.

A CodeReuse Subsystem block is a subsystem block in which **Treat as atomic unit** is selected and **Function packaging** is set to `Reusable function`, specifying the function code generation format for the subsystem. (see “Function packaging” on page 1-1647 for details).

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's 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.

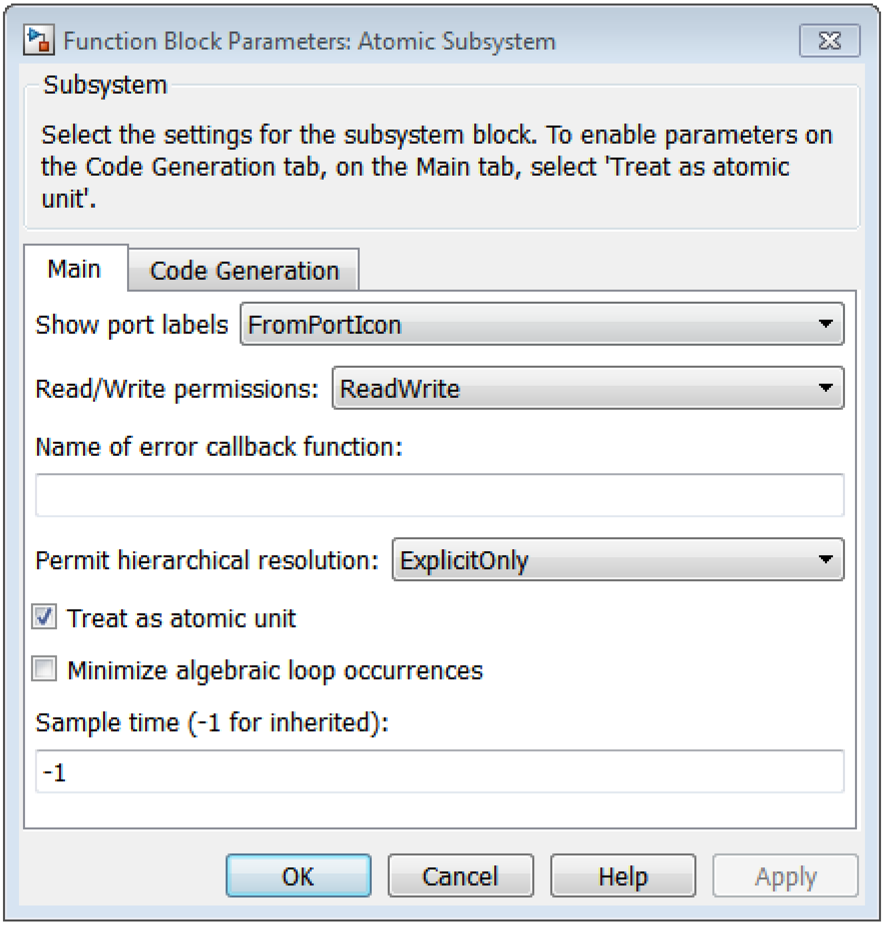
See "Create a Subsystem" for more information.

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 and Dialog Box



Function Block Parameters: Atomic Subsystem

Subsystem

Select the settings for the subsystem block. To enable parameters on the Code Generation tab, on the Main tab, select 'Treat as atomic unit'.

Main
Code Generation

Show port labels: FromPortIcon

Read/Write permissions: ReadWrite

Name of error callback function:

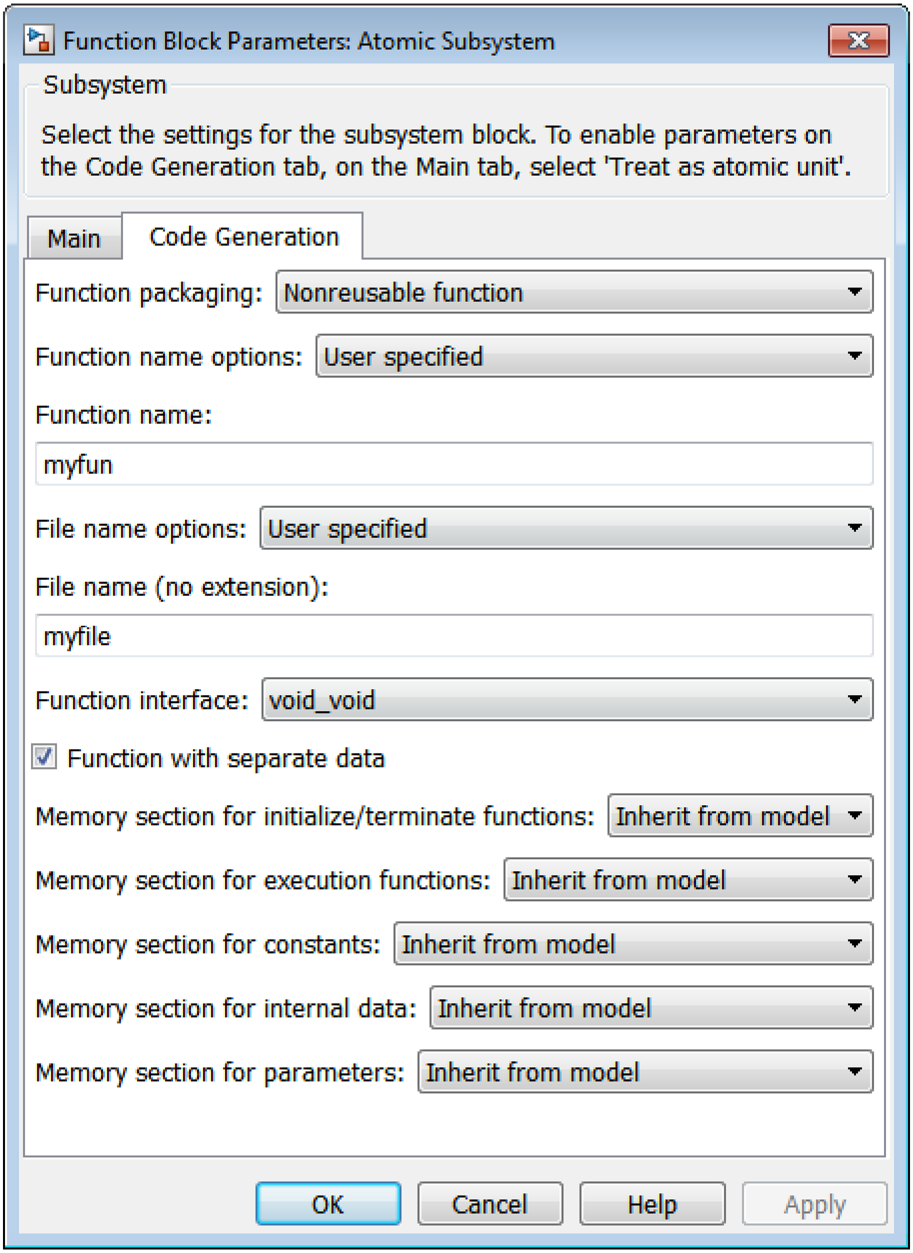
Permit hierarchical resolution: ExplicitOnly

☑ Treat as atomic unit

☐ Minimize algebraic loop occurrences

Sample time (-1 for inherited): -1

OK
Cancel
Help
Apply



Function Block Parameters: Atomic Subsystem

Subsystem

Select the settings for the subsystem block. To enable parameters on the Code Generation tab, on the Main tab, select 'Treat as atomic unit'.

Main
Code Generation

Function packaging: Nonreusable function

Function name options: User specified

Function name:

myfun

File name options: User specified

File name (no extension):

myfile

Function interface: void_void

☑ Function with separate data

Memory section for initialize/terminate functions: Inherit from model

Memory section for execution functions: Inherit from model

Memory section for constants: Inherit from model

Memory section for internal data: Inherit from model

Memory section for parameters: Inherit from model

OK
Cancel
Help
Apply

Note Parameters on the Code Generation tab require a Simulink Coder or Embedded Coder license. For more information, see the parameter sections.

Show port labels

Cause Simulink software to display labels for the subsystem's ports on the subsystem's icon.

Settings

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Read/Write permissions

Control user access to the contents of the subsystem.

Settings

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 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.

Settings

Default: `' '`

Simulink software passes two arguments to the function: the handle of the subsystem and a string that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Permit hierarchical resolution

Specify whether to resolve names of workspace variables referenced by this subsystem.

Settings

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 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.

Settings

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.

Dependencies

This parameter enables:

- “Minimize algebraic loop occurrences” on page 1-1643.
- “Sample time (-1 for inherited)” on page 1-1646
- “Function packaging” on page 1-1647 (requires a Simulink Coder license)

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Minimize algebraic loop occurrences

Try to eliminate any artificial algebraic loops that include the atomic subsystem

Settings

Default: Off

☑ 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.

Dependency

"Treat as atomic unit" on page 1-1642 enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Propagate execution context across subsystem boundary

Enable execution context propagation across the boundary of this subsystem.

Settings

Default: Off

☑ On

Enables execution context propagation across this subsystem's boundary.

☐ Off

Does not enable execution context propagation across this subsystem's boundary.

Dependency

Conditional execution of the subsystem enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Warn if function-call inputs are context-specific

Simulink displays a warning if it has to compute any of this function-call subsystem's inputs directly or indirectly during execution of a function-call.

Settings

Default: Off

☑ On

Simulink displays a warning if it has to compute any of this function-call subsystem's inputs directly or indirectly during execution of a function-call.

☐ Off

Simulink does not display a warning if it has to compute any of this function-call subsystem's inputs directly or indirectly during execution of a function-call.

Dependency

Use of a function-call subsystem enables this parameter.

The option is effective only when the **Context-dependent inputs** diagnostic on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box is set to `Use local settings`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Specify whether all blocks in this subsystem must run at the same rate or can run at different rates.

Settings

Default: `-1`

- `-1`

  Specify the inherited sample time. Use this sample time if the blocks in the subsystem can run at different rates.

- `[Ts 0]`

  Specify periodic sample time.

Tips

- 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 subsystem's **Sample time** parameter.
- If any of the blocks in the subsystem specify a different sample time (other than `-1` or `inf`), Simulink software 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 software displays an error when you update or simulate the model.

Dependency

"Treat as atomic unit" on page 1-1642 enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Function packaging

Specify the code format to be generated for an atomic (nonvirtual) subsystem.

Settings

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 1-1658 parameter setting. You can name the generated function and file using parameters "Function name" on page 1-1652 and "File name (no extension)" on page 1-1655. 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.

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 without arguments for reach 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.

Dependencies

- This parameter requires a Simulink Coder license.
- "Treat as atomic unit" on page 1-1642 enables this parameter.
- Setting this parameter to `Nonreusable function` or `Reusable function` enables the following parameters:
  - "Function name options" on page 1-1650
  - "File name options" on page 1-1653
  - "Memory section for initialize/terminate functions" on page 1-1659 (requires a license for Embedded Coder software and an ERT-based system target file)
  - "Memory section for execution functions" on page 1-1661 (requires a license for Embedded Coder software and an ERT-based system target file)

- Setting this parameter to `Nonreusable function` enables “Function with separate data” on page 1-1656 (requires a license for Embedded Coder software and an ERT-based system target file).

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Function name options

Specify how Simulink Coder software is to name the function it generates for the subsystem.

Settings

Default: `Auto`

`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.

Note When a subsystem is a library block and the subsystem parameter "Function packaging" on page 1-1647 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`

This option enables the **Function name** field. Enter any legal C or C++ function name, which must be unique.

Dependencies

- This parameter requires a Simulink Coder license.

- Setting "Function packaging" on page 1-1647 to `Nonreusable function` or `Reusable function` enables this parameter.
- Setting this parameter to `User specified` enables the "Function name" on page 1-1652 parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Function name

Specify a unique, valid C or C++ function name for subsystem code.

Settings

Default: `'  '`

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.

Dependencies

- This parameter requires a Simulink Coder license.
- Setting “Function name options” on page 1-1650 to `User specified` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

File name options

Specify how Simulink Coder software names the separate file for the function it generates for the subsystem.

Settings

Default: `Auto`

`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`).

Dependencies

- This parameter requires a Simulink Coder license.
- Setting “Function packaging” on page 1-1647 to `Nonreusable function` or `Reusable function` enables this parameter.
- Setting this parameter to `User specified` enables the “File name (no extension)” on page 1-1655 parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

File name (no extension)

Specify how Simulink Coder software is to name the file for the function it generates for the subsystem.

Settings

Default: `' '`

- The filename that you specify does not have to be unique. However, avoid giving non-unique names that result in cyclic dependencies (for example, `sys_a.h` includes `sys_b.h`, `sys_b.h` includes `sys_c.h`, and `sys_c.h` includes `sys_a.h`).

Dependencies

- This parameter requires a Simulink Coder license.
- Setting “File name options” on page 1-1653 to `User specified` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Function with separate data

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.

Settings

Default: Off

☑ 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.

☐ 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.

Dependencies

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- Setting “Function packaging” on page 1-1647 to `Nonreusable function` enables this parameter.
- Selecting this check box enables these parameters:
  - “Memory section for constants” on page 1-1663
  - “Memory section for internal data” on page 1-1664
  - “Memory section for parameters” on page 1-1665

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Function interface

For this subsystem, specify whether the generated function uses arguments.

Settings

Default: `void_void`

`void_void`

Generate a function without arguments and passes 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)
```

Dependencies

- This parameter requires an Embedded Coder license and an ERT-based system target file.
- Setting “Function packaging” on page 1-1647 to `Nonreusable function` enables this parameter.

Command-Line Information

For the command-line information, see “Block-Specific Parameters” on page 6-109.

Memory section for initialize/terminate functions

Indicate how the Embedded Coder software is to apply memory sections to the subsystem's initialization and termination functions.

Settings

Default: `Inherit from model`

`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 possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See "About Memory Sections", "Configure Memory Sections", and "Code Generation Pane: Memory Sections" in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.
- These options can be useful for overriding the model's memory section settings for the given subsystem.

Dependencies

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- Setting "Function packaging" on page 1-1647 to `Nonreusable function` or `Reusable function` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Memory section for execution functions

Indicate how the Embedded Coder software is to apply memory sections to the subsystem's execution functions.

Settings

Default: `Inherit from model`

`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 possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See "About Memory Sections", "Configure Memory Sections", and "Code Generation Pane: Memory Sections" in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.
- These options can be useful for overriding the model's memory section settings for the given subsystem.

Dependencies

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- Setting "Function packaging" on page 1-1647 to `Nonreusable function` or `Reusable function` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Memory section for constants

Indicate how the Embedded Coder software is to apply memory sections to the subsystem's data.

Settings

Default: `Inherit from model`

`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

- 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 "Configure Memory Sections" in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.

Dependencies

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- Setting "Function packaging" on page 1-1647 to `Nonreusable function` and selecting the "Function with separate data" on page 1-1656 check box enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Memory section for internal data

Indicate how the Embedded Coder software is to apply memory sections to the subsystem's data.

Settings

Default: `Inherit from model`

`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

- 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 "Configure Memory Sections" in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.

Dependencies

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- Setting "Function packaging" on page 1-1647 to `Nonreusable function` and selecting the "Function with separate data" on page 1-1656 check box enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Memory section for parameters

Indicate how the Embedded Coder software is to apply memory sections to the subsystem's data.

Settings

Default: `Inherit from model`

`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

- 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 "Configure Memory Sections" in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.

Dependencies

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- Setting "Function packaging" on page 1-1647 to `Nonreusable function` and selecting the "Function with separate data" on page 1-1656 check box enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Characteristics

Sample Time	Depends on the blocks in the subsystem
Dimensionalized	Depends on the blocks in the subsystem
Multidimensionalized	Depends on the blocks in the subsystem
Virtual	Yes, if the read-only property `IsSubsystemVirtual` is `on`
Zero Crossing	Yes, for enable and trigger ports if present

Purpose

Add or subtract inputs

Library

Math Operations

Description

The Sum block performs addition or subtraction on its inputs. This block can add or subtract scalar, vector, or matrix inputs. It can also collapse the elements of a signal.

You specify the operations of the block with the **List of signs** parameter. Plus (+), minus (-), and spacer (|) characters indicate the operations to be performed on the inputs:

- If there are two or more inputs, then the number of + and - characters must equal the number of inputs. For example, “+-+” requires three inputs and configures the block to subtract the second (middle) input from the first (top) input, and then add the third (bottom) input.
- All nonscalar inputs must have the same dimensions. Scalar inputs will be expanded to have the same dimensions as the other inputs.
- A spacer character creates extra space between ports on the block’s icon.
- For a round Sum block, the first input port is the port closest to the 12 o’clock position going in a counterclockwise direction around the block. Similarly, other input ports appear in counterclockwise order around the block.
- If only addition of all inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of “+” characters.
- If only one input port is required, a single “+” or “-” collapses the element via the specified operation.

The Sum block first converts the input data type(s) to its accumulator data type, then performs the specified operations. The block converts the result to its output data type using the specified rounding and overflow modes.

Calculation of Block Output

Output calculation for the Sum block depends on the number of block inputs and the sign of input ports:

If the Sum block has...	And...	The formula for output calculation is...	Where...
One input port	The input port sign is +	y = e[0] + e[1] + e[2] ... + e[m]	`e[i]` is the i^{th} element of input u
	The input port sign is –	y = 0.0 – e[0] – e[1] – e[2] ... – 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 i^{th} input port
	The k^{th} input port is the first port where the sign is +	y = u[k] – u[0] – u[1] – u[2] – u[k–1] (+/–) u[k+1] ... (+/–) u[n]	

Data Type Support

The Sum block accepts real or complex signals of the following data types:

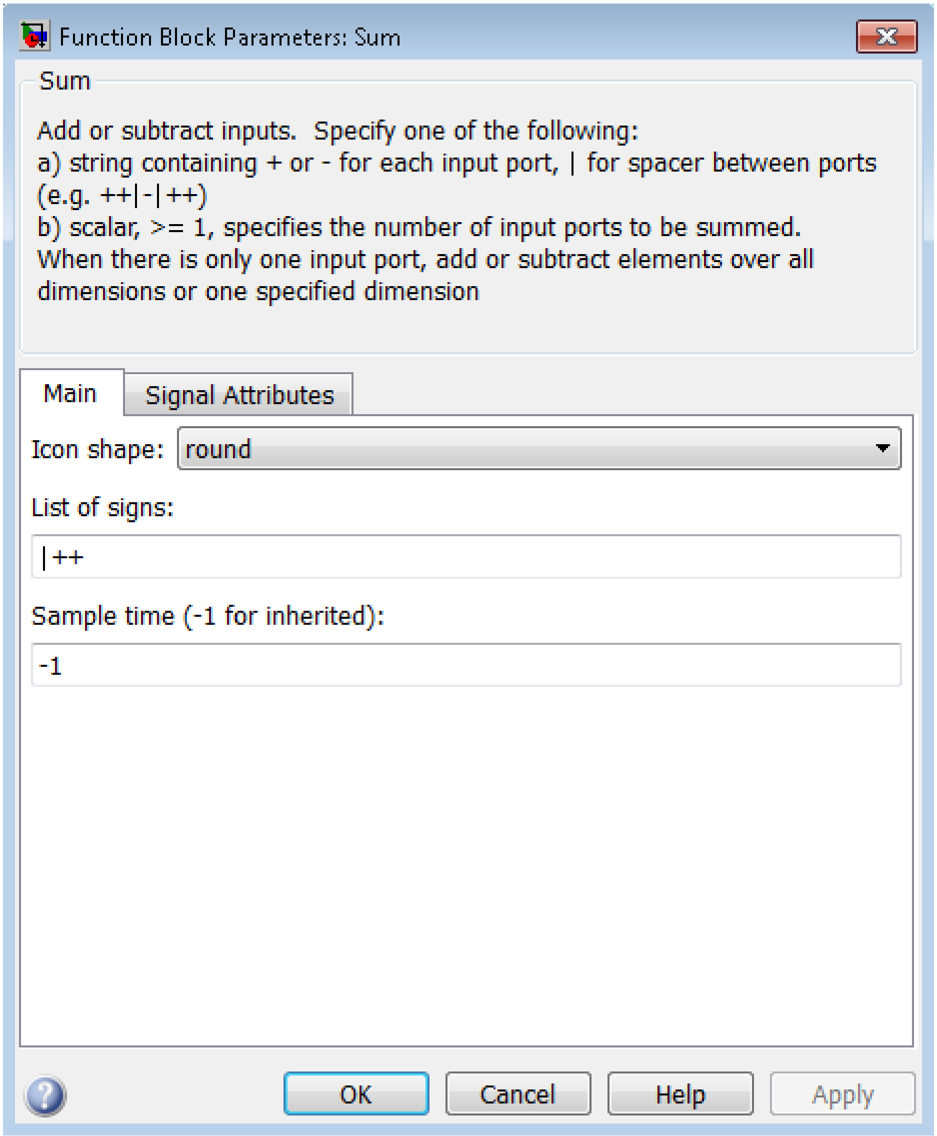
- Floating point
- Built-in integer

- Fixed point
- Boolean

The inputs can be of different data types, unless you select the **Require all inputs to have the same data type** parameter. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Sum block dialog box appears as follows:



Function Block Parameters: Sum

Sum

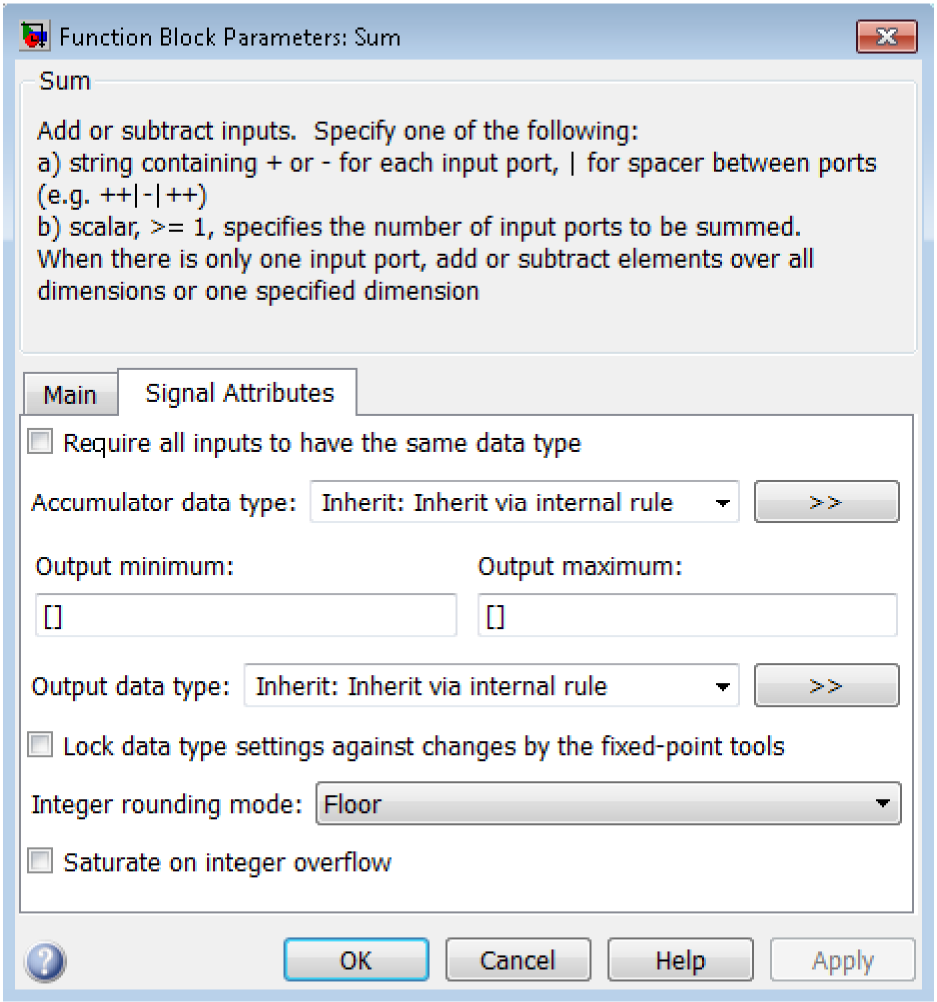
Add or subtract inputs. Specify one of the following:
a) string containing + or - for each input port, | for spacer between ports (e.g. ++|-|++)
b) scalar, >= 1, specifies the number of input ports to be summed.
When there is only one input port, add or subtract elements over all dimensions or one specified dimension

Icon shape: round

List of signs: |++

Sample time (-1 for inherited): -1

The **Signal Attributes** pane of the Sum block dialog box appears as follows:



Sum

Add or subtract inputs. Specify one of the following:
a) string containing + or - for each input port, | for spacer between ports (e.g. ++|-|++)
b) scalar, >= 1, specifies the number of input ports to be summed.
When there is only one input port, add or subtract elements over all dimensions or one specified dimension

☐ Require all inputs to have the same data type

Accumulator data type: Inherit: Inherit via internal rule

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

☐ Lock data type settings against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate on integer overflow

Show data type assistant

Display the **Data Type Assistant**.

Settings

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Specify Block Output Data Types”.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Icon shape

Designate the icon shape of the block.

Settings

Default: `round`

`rectangular`
Designate the icon shape of the block as rectangular.

`round`
Designate the icon shape of the block as round.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

List of signs

Enter plus (+) and minus (-) characters.

Settings

Default: `|++`

- Addition is the default operation, so if you only want to add the inputs, enter the number of input ports.
- For a single vector input, "+" or "-" will collapse the vector using the specified operation.
- Enter as many plus (+) and minus (-) characters as there are inputs.

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.

Dependencies

Entering only one element enables the **Sum over** parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sum over

Select dimension over which to perform the sum over operation.

Settings

Default: `All dimensions`

`All dimensions`
: Sum all input elements, yielding a scalar.

`Specified dimension`
: Display the **Dimension** parameter, where you specify the dimension over which to perform the operation.

Dependencies

Selecting `Specified dimension` enables the **Dimension** parameter.

List of signs (when it has only one element) enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Dimension

Specify the dimension over which to perform the operation.

Settings

Default: `1`

The block follows the same summation rules as the MATLAB `sum` function.

Suppose that you have a 2-by-3 matrix U.

- Setting **Dimension** to `1` results in the output Y being computed as:

$$Y = \sum_{i=1}^{2} U(i, j)$$

- Setting **Dimension** to `2` results in the output Y being computed as:

$$Y = \sum_{j=1}^{3} U(i, j)$$

If the specified dimension is greater than the dimension of the input, an error message appears.

Dependencies

Setting **Sum over** to `Specified dimension` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in "Types of Sample Time".

See also "Specify Sample Time" in the online documentation for more information.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Settings

Default: Off

☑ On
  Require that all inputs have the same data type.

☐ Off
  Do not require that all inputs have the same data type.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Lock data type settings against changes by the fixed-point tools

Select to lock data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑On

Locks all data type settings for this block.

☐Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change data type settings for this block.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
: Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`
: Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`
: Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`
: Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`
: Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`
: Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`
: Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see “Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Accumulator data type

Specify the accumulator data type.

Settings

Default: `Inherit:  Inherit via internal rule`

`Inherit:  Inherit via internal rule`
: Use internal rule to determine accumulator data type.

`Inherit:  Same as first input`
: Use data type of first input signal.

`double`
: Accumulator data type is `double`.

`single`
: Accumulator data type is `single`.

`int8`
: Accumulator data type is `int8`.

`uint8`
: Accumulator data type is `uint8`.

`int16`
: Accumulator data type is `int16`.

`uint16`
: Accumulator data type is `uint16`.

`int32`
: Accumulator data type is `int32`.

`uint32`
: Accumulator data type is `uint32`.

`fixdt(1,16,0)`
: Accumulator data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
: Accumulator data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
The name of a data type object, for example `Simulink.NumericType`

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of accumulator data to specify

Settings

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit via internal rule` (default)
- `Same as first input`

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Specifies fixed-point data types.

`Expression`

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Dependency

Clicking the **Show data type assistant** button for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant".

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data to be signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specify the fixed-point data to be signed.

`Unsigned`
: Specify the fixed-point data to be unsigned.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Word length

Specify the bit size of the word that will hold the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Binary point`

`Binary point`
Specify binary point location.

`Slope and bias`
Enter slope and bias.

Dependencies

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` for the accumulator data type enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` for the accumulator data type enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the minimum to perform:

- Parameter range checking (see "Check Parameter Values") for some blocks
- Simulation range checking (see "Signal Ranges" and "Enabling Simulation Range Checking")
- Automatic scaling of fixed-point data types

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enabling Simulation Range Checking”)
- Automatic scaling of fixed-point data types

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Inherit via internal rule`

`Inherit:  Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit:  Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit:  Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

Note The accumulator internal rule favors greater numerical accuracy, possibly at the cost of less efficient generated code. To get the same accuracy for the output, set the output data type to `Inherit:  Inherit same as accumulator`.

`Inherit:  Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as first input`
: Use data type of first input signal.

`Inherit: Same as accumulator`
: Output data type is the same as accumulator data type.

`double`
: Output data type is `double`.

`single`
: Output data type is `single`.

`int8`
: Output data type is `int8`.

`uint8`
: Output data type is `uint8`.

`int16`
: Output data type is `int16`.

`uint16`
: Output data type is `uint16`.

`int32`
: Output data type is `int32`.

`uint32`
: Output data type is `uint32`.

`fixdt(1,16,0)`
: Output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
: Output data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
: Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specify Block Output Data Types”.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule` (default)
- `Inherit via back propagation`
- `Same as first input`
- `Same as accumulator`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specify Data Types Using Data Type Assistant".

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
: Specify the fixed-point data as signed.

`Unsigned`
: Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Binary point`

`Binary point`
Specify binary point location.

`Slope and bias`
Enter slope and bias.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type".

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Examples

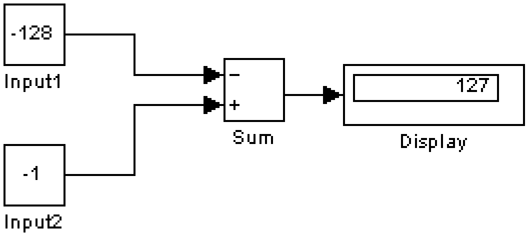
How the Sum Block Reorders Inputs

If you use `-` on the first input port, the Sum block reorders the inputs so that, if possible, the first input uses a `+` operation. For example, in the expression `output = -a-b+c`, the Sum block reorders the inputs 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`.

Tip To explicitly specify a unary minus operation for `output = -a-b+c`, you can use the Unary Minus block in the Math Operations library.

Suppose that you have the following model:



The following block parameters apply:

- Both Constant blocks, `Input1` and `Input 2`, use `int8` for the **Output data type**.
- The Sum block uses `int8` for both **Accumulator data type** and **Output data type**.
- The Sum block has **Saturate on integer overflow** turned `on`.

The Sum block reorders the inputs so that the following operations occur and you get the ideal result of 127.

Step	Block Operation
1	Reorders inputs from `( Input1 + Input2)` to `(Input2  Input1)`.
2	Initializes the accumulator by using the first + input port: `Accumulator = int8(-1) = -1`
3	Continues to accumulate values: `Accumulator = Accumulator   int8(-128) = 127`
4	Calculates the block output: `Output = int8(127) = 127`

If the Sum block does *not* reorder the inputs, the following operations occur instead and you get the nonideal result of 126.

Step	Block Operation
1	Initializes the accumulator by using the first input port: `Accumulator = int8(-(-128)) = 127` Because saturation is on, the initial value of the accumulator saturates at 127 and does not wrap.
2	Continues to accumulate values: `Accumulator = Accumulator + int8(-1) = 126`
3	Calculates the block output: `Output = int8(126) = 126`

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes

States	0
Dimensionalized	Yes
Multidimensionalized	Yes, only along the specified dimension
Zero-Crossing Detection	No

Switch

Purpose

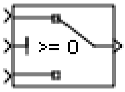
Switch output between first input and third input based on value of second input

Library

Signal Routing

Description

Types of Block Inputs



The Switch block passes through the first input or the third input based on the value of the second input. The first and third inputs are called *data inputs*. The second input is called the *control input*. Specify the condition under which the block passes the first input by using the **Criteria for passing first input** and **Threshold** parameters.

To immediately back propagate a known output data type to the first and third input ports, set the **Output data type** parameter to `Inherit:  Inherit via internal rule` and select the **Require all data port inputs to have the same data type** check box.

Limitations on Data Inputs

The sizes of the two data inputs can be different if you select **Allow different data input sizes** on the block dialog box. However, this block does not support variable-size input signals. Therefore, the size of each input cannot change during simulation.

If the data inputs to the Switch block are buses, the element names of both buses must be the same. Using the same element names ensures that the output bus has the same element names no matter which input bus the block selects. To ensure that your model meets this requirement, use a bus object to define the buses and set the **Element name mismatch** diagnostic to `error`. See “Connectivity Diagnostics Overview” for more information.

Block Icon Appearance

The block icon helps you identify **Criteria for passing first input** and **Threshold** without having to open the block dialog box.

For information about port order for various block orientations, see “How to Rotate a Block” in the Simulink documentation.

Block Behavior for Boolean Control Input

When the control input is a Boolean signal, use one of these combinations of criteria and threshold value:

- `u2 >= Threshold`, where the threshold value equals 1
- `u2 > Threshold`, where the threshold value equals 0
- `u2 ~=0`

Otherwise, the Switch block ignores the threshold and uses the Boolean input for signal routing. For a control input of 1, the block passes the first input, and for a control input of 0, the block passes the third input. In this case, the block icon changes *after compile time* and uses `T` and `F` to label the first and third inputs, respectively.

Data Type Support

The control input can be of any data type that Simulink supports, including fixed-point and enumerated types. The control input cannot be complex. If the control input is enumerated, the **Threshold** parameter must be a value of the same enumerated type.

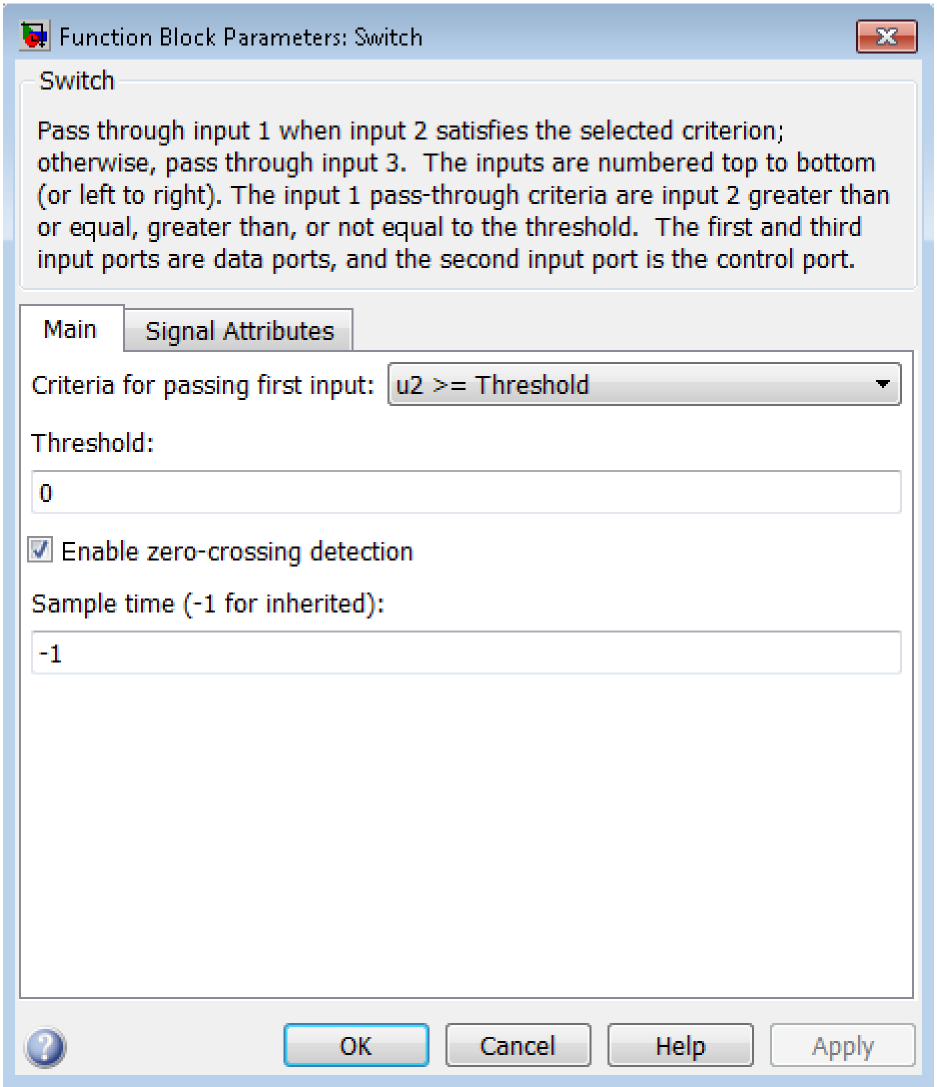
The data inputs can be of any data type that Simulink supports. If either data input is of an enumerated type, the other must be of the same enumerated type.

When the output is of enumerated type, both data inputs should use the same enumerated type as the output.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Switch block dialog box appears as follows:



Function Block Parameters: Switch

Switch

Pass through input 1 when input 2 satisfies the selected criterion; otherwise, pass through input 3. The inputs are numbered top to bottom (or left to right). The input 1 pass-through criteria are input 2 greater than or equal, greater than, or not equal to the threshold. The first and third input ports are data ports, and the second input port is the control port.

Main
Signal Attributes

Criteria for passing first input: u2 >= Threshold

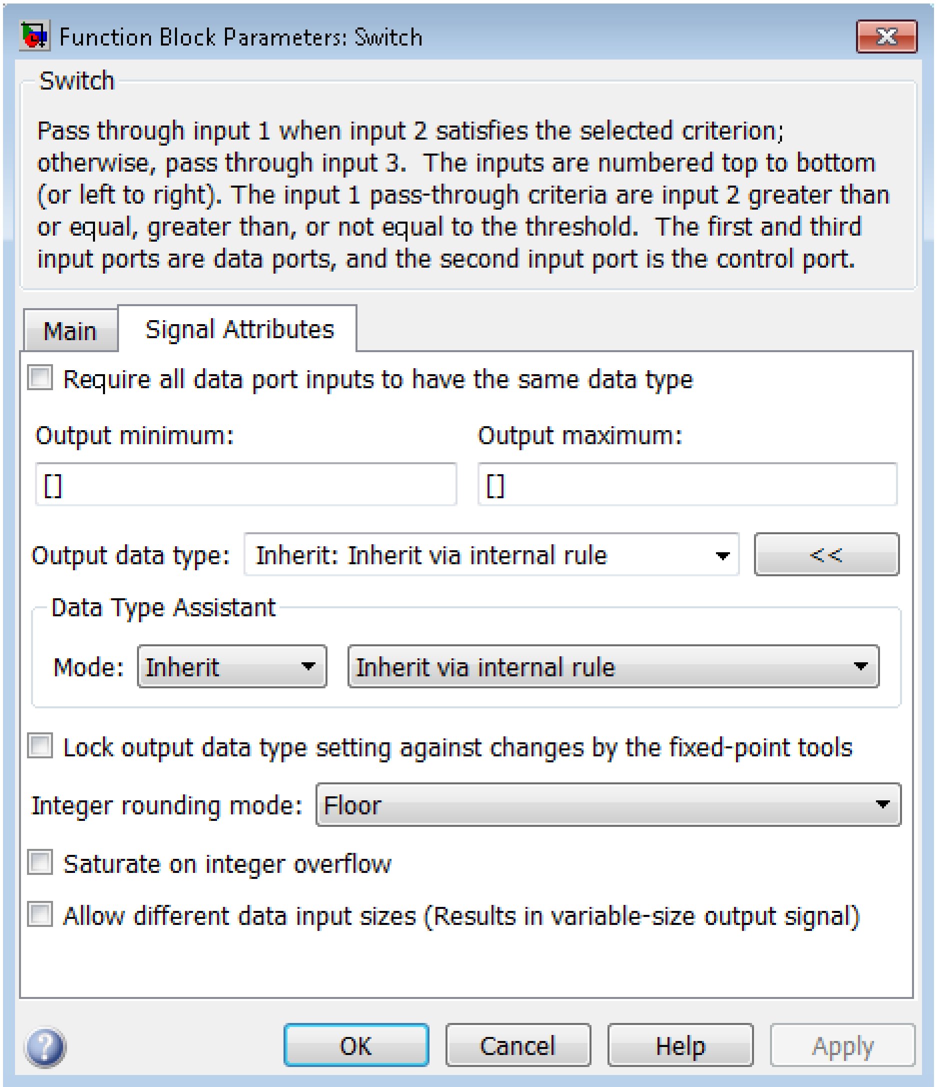
Threshold: 0

☑ Enable zero-crossing detection

Sample time (-1 for inherited): -1

OK
Cancel
Help
Apply

The **Signal Attributes** pane of the Switch block dialog box appears as follows:



Function Block Parameters: Switch
Switch

Pass through input 1 when input 2 satisfies the selected criterion; otherwise, pass through input 3. The inputs are numbered top to bottom (or left to right). The input 1 pass-through criteria are input 2 greater than or equal, greater than, or not equal to the threshold. The first and third input ports are data ports, and the second input port is the control port.

Main
Signal Attributes

☐ Require all data port inputs to have the same data type

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

<<
Data Type Assistant

Mode: Inherit | Inherit via internal rule

☐ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

☐ Saturate on integer overflow

☐ Allow different data input sizes (Results in variable-size output signal)

OK
Cancel
Help
Apply

- “Criteria for passing first input” on page 1-1715
- “Threshold” on page 1-1717
- “Enable zero-crossing detection” on page 1-1718
- “Sample time (-1 for inherited)” on page 1-1974
- “Require all data port inputs to have the same data type” on page 1-1720
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1721
- “Integer rounding mode” on page 1-1722
- “Saturate on integer overflow” on page 1-1724
- “Allow different data input sizes” on page 1-1726
- “Output minimum” on page 1-1727
- “Output maximum” on page 1-1728
- “Output data type” on page 1-1729
- “Mode” on page 1-1731
- “Data type override” on page 1-1926
- “Signedness” on page 1-1927
- “Word length” on page 1-1930
- “Scaling” on page 1-1736
- “Fraction length” on page 1-1931
- “Slope” on page 1-1932
- “Bias” on page 1-1932

Criteria for passing first input

Select the condition under which the block passes the first input. If the control input meets the condition set in the **Criteria for passing first input** parameter, the block passes the first input. Otherwise, the block passes the third input.

Settings

Default: `u2 >= Threshold`

`u2 >= Threshold`
Checks whether the control input is greater than or equal to the threshold value.

`u2 > Threshold`
Checks whether the control input is greater than the threshold value.

`u2 ~=0`
Checks whether the control input is nonzero.

Note The Switch block does not support `u2 ~=0` mode for enumerated data types.

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.

Dependencies

Selecting `u2 ~=0` disables the **Threshold** parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Threshold

Assign the switch threshold that determines which input the block passes to the output.

Settings

Default: `0`

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

Tip

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Settings

Default: On

☑ On

Enable zero-crossing detection.

☐ Off

Do not enable zero-crossing detection.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Require all data port inputs to have the same data type

Require all data inputs to have the same data type.

Settings

Default: Off

☑ On

Requires all data inputs to have the same data type.

☐ Off

Does not require all data inputs to have the same data type.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On

Locks the output data type setting for this block.

☐ Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Use Lock Output Data Type Setting”.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
: Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`
: Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`
: Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`
: Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`
: Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`
: Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`
: Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Rounding” in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

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 is representable by the data type.

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 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 avoid overspecifying how a block handles out-of-range signals. For more information, see “Checking 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Allow different data input sizes

Select this check box to allow input signals with different sizes.

Settings

Default: Off

☑ On

Allows input signals with different sizes, and propagates the input signal size to the output signal. If the two data inputs are variable-size signals, the maximum size of the signals can be equal or different.

☐ Off

Inputs signals must be the same size.

Command-Line Information

Parameter: `AllowDiffInputSize`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Output minimum

Specify the minimum value that the block should output.

Settings

Default: `[]`

The default value is `[]` (unspecified).

Simulink uses this value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Tip

This number must be a finite real double scalar value.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output maximum

Specify the maximum value that the block should output.

Settings

Default: `[]`

The default value is `[]` (unspecified).

Simulink uses this value to perform:

- Parameter range checking (see “Check Parameter Values”) for some blocks
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Tip

This number must be a finite real double scalar value.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the output data type.

Settings

Default: `Inherit:  Inherit via internal rule`

`Inherit:  Inherit via internal rule`

Uses the following rules to determine the output data type.

Data Type of First Input Port	Output Data Type
Has a larger positive range than the third input port	Inherited from the first input port
Has the same positive range as the third input port	Inherited from the third input port
Has a smaller positive range than the third input port	

`Inherit:  Inherit via back propagation`

Uses data type of the driving block.

`double`

Specifies output data type is `double`.

`single`

Specifies output data type is `single`.

`int8`

Specifies output data type is `int8`.

`uint8`

Specifies output data type is `uint8`.

`int16`

Specifies output data type is `int16`.

`uint16`

Specifies output data type is `uint16`.

`int32`
Specifies output data type is `int32`.

`uint32`
Specifies output data type is `uint32`.

`fixdt(1,16,0)`
Specifies output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
Specifies output data type is fixed point `fixdt(1,16,2^0,0)`.

`Enum: <class name>`
Uses an enumerated data type, for example, `Enum: BasicColors`.

`<data type expression>`
Uses a data type object, for example, `Simulink.NumericType`.

Tip

When the output is of enumerated type, both data inputs should use the same enumerated type as the output.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Block Output Data Types” for more information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit via internal rule` (default)
- `Inherit via back propagation`

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Specifies fixed-point data types.

`Enumerated`

Specifies enumerated data types. Selecting `Enumerated` enables you to enter a class name.

`Expression`

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Dependencies

Clicking the **Show data type assistant** button enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`
: Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`
: Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
Specifies the fixed-point data as signed.

`Unsigned`
Specifies the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`

`Binary point`
: Specify binary point location.

`Slope and bias`
: Enter slope and bias.

`Best precision`
: Specify best-precision values.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

See “Specifying a Fixed-Point Data Type” for more information.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

See "Specifying a Fixed-Point Data Type" for more information.

Bus Support

The Switch block is a bus-capable block. The data inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- All the buses must be equivalent (same hierarchy with identical names and attributes for all elements).
- All signals in a nonvirtual bus input to a Switch block must have the same sample time. The requirement holds even if the elements of the associated bus object specify inherited sample times.

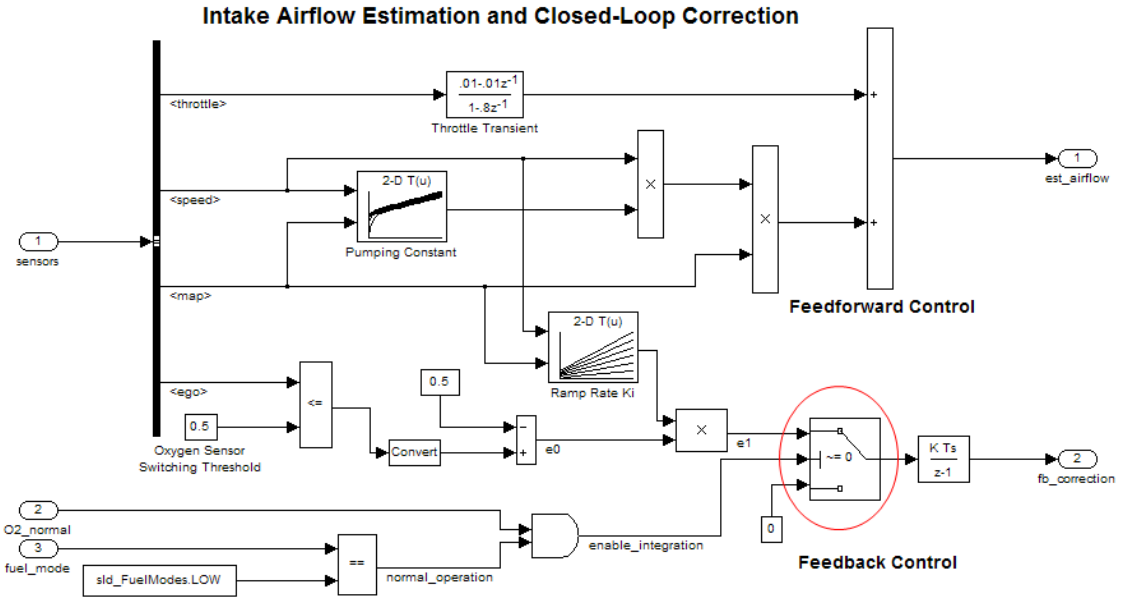
You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See "About Composite Signals" and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Switch block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses". When using an array of buses with a Switch block, set the **Threshold** parameter to a scalar value.

Examples

Use of Boolean Input for the Control Port

In the `sldemo_fuelsys` model, the `fuel_rate_control/airflow_calc` subsystem uses a Switch block:

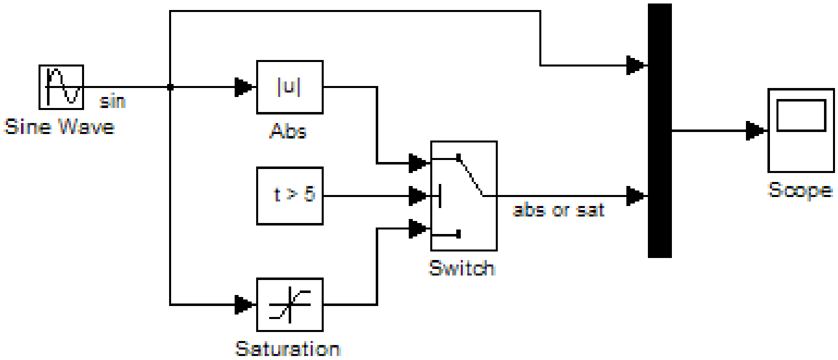

Intake Airflow Estimation and Closed-Loop Correction
<throttle>
.01-.01z-1
1-.8z-1
Throttle Transient
<speed>
2-D T(u)
Pumping Constant
1
sensors
<map>
×
×
+
+
1
est_airflow
Feedforward Control
2-D T(u)
Ramp Rate Ki
<ego>
<=
0.5
0.5
Oxygen Sensor
Switching Threshold
Convert
-
+
e0
×
e1
~= 0
0
K Ts
z-1
2
fb_correction
2
O2_normal
3
fuel_mode
sld_FuelModes.LOW
==
normal_operation
enable_integration
Feedback Control

The value of the control port on the Switch block determines whether or not feedback correction occurs. The control port value depends on the output of the Logical Operator block.

When the Logical Operator block output is...	The control port of the Switch block is...	And feedback control...
TRUE	1	Occurs
FALSE	0	Does not occur

Use of Simulation Time for the Control Port

The `sldemo_zeroxing` model uses a Switch block:



The value of the control port on the Switch block determines when the output changes from the first input to the third input.

When simulation time is...	The Switch block output is...
Less than or equal to 5	The first input from the Abs block
Greater than 5	The third input from the Saturation block

Characteristics

Bus-capable	Yes, with restrictions
Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Dimensionalized	Yes

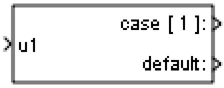
Multidimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled

See Also Multiport Switch

Purpose Implement C-like `switch` control flow statement

Library Ports & Subsystems

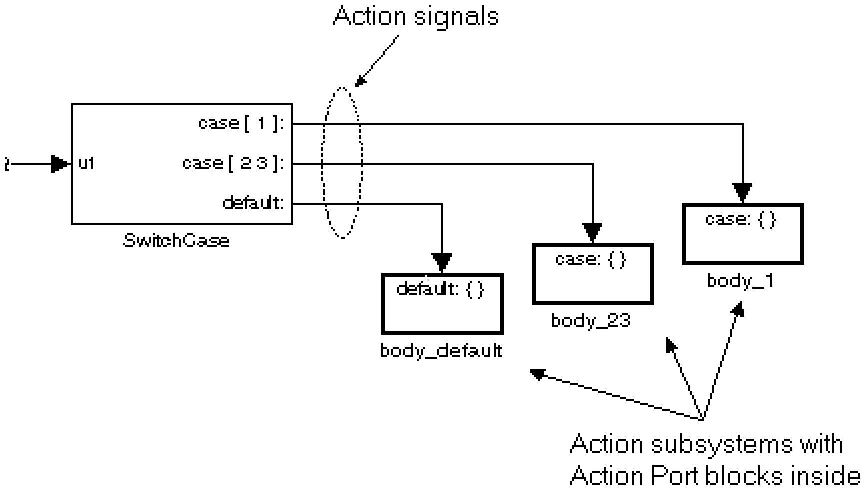
Description



A Switch Case block receives a single input. Each output port is attached to aSwitch Case Action Subsystem. Data outputs are action signals to Switch Case Action subsystems, which you create with Action Port blocks and subsystems.

The Switch Case block uses its input value to select a case condition that determines which subsystem to execute. The cases are evaluated top down starting with the first case. If a case value (in brackets) corresponds to the value of the input, its Switch Case Action Subsystem is executed.

If a `default` case exists, it executes if none of the other cases executes. Providing a `default` case is optional, even if the other case conditions do not exhaust every possible value. The following diagram shows a completed Simulink `switch` control flow statement:


Action signals
case [1]:
u1
case [2 3]:
default:
SwitchCase
default: { }
body_default
case: { }
body_23
case: { }
body_1
Action subsystems with
Action Port blocks inside

Cases for the Switch Case block contain an implied break after their Switch Case Action subsystems are executed. Thus there is no fall-through behavior for the Simulink `switch` control flow statement as found in standard C `switch` statements. The following pseudocode represents generated code for the preceding `switch` control example:

```
switch (u1) {
 case [u1=1]:
  body_1;
  break;
 case [u1=2 or u1=3]:
  body_23;
  break;
 default:
  body_default;
}
```

To construct the Simulink `switch` control flow statement shown in the above example:

1. Place a Switch Case block in the current system and attach the input port labeled `u1` to the source of the data you are evaluating.

2. Open the Switch Case block dialog box and update parameters:

   a. Populate the **Case conditions** field with the individual cases.

   b. To show a default case, select the **Show default case** check box.

3. Create a Switch Case Action Subsystem for each case port you added to the Switch Case block.

   These consist of subsystems with Action Port blocks inside them. When you place the Action Port block inside a subsystem, the subsystem becomes an atomic subsystem with an input port labeled `Action`.

4. Connect each case output port and the default output port of the Switch Case block to the Action port of an Action subsystem.

   Each connected subsystem becomes a case body. This is indicated by the change in label for the Switch Case Action Subsystem block and the Action Port block inside of it to the name `case{}`.

   During simulation of a `switch` control flow statement, the Action signals from the Switch Case block to each Switch Case Action Subsystem turn from solid to dashed.

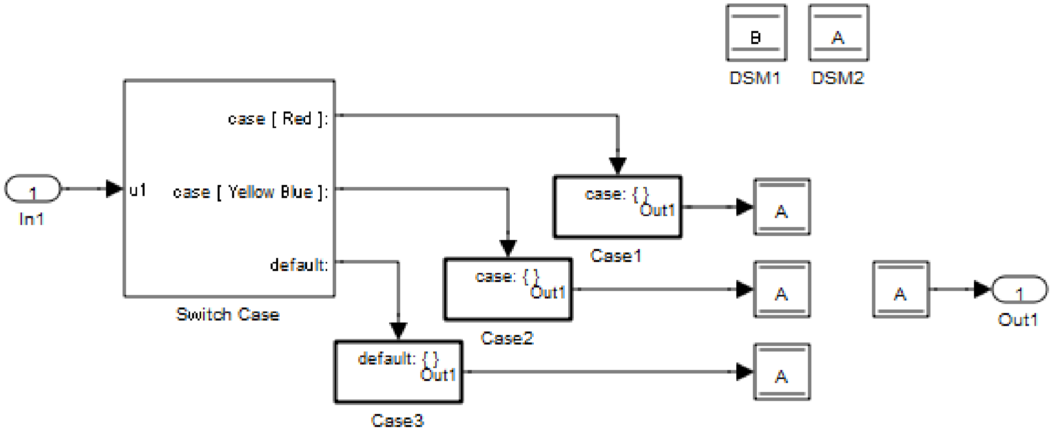
5. In each Switch Case Action Subsystem, enter the Simulink logic appropriate to the case it handles. All blocks in a Switch Case Action Subsystem must run at the same rate as the driving Switch Case block. You can achieve this by setting each block's sample time parameter to be either inherited (`-1`) or the same value as the Switch Case block's sample time.

Enumerated Types in Switch Case Blocks

The Switch Case block supports enumerated data types for the input signal and the case conditions. You specify enumerated case conditions as a cell array of enumerated values in the **Case conditions** field, for example:

```
{BasicColors.Red, [BasicColors.Yellow, BasicColors.Blue]}
```

The Switch Case block icon shows the enumerated names that correspond to the case conditions, for example:


B
A
DSM1
DSM2
case [Red]:
1
In1
u1
case [Yellow Blue]:
default:
Switch Case
case: { }
Out1
Case1
A
case: { }
Out1
Case2
A
A
1
Out1
default: { }
Out1
Case3
A

You can use the `enumeration` function to specify a case condition that includes a case for every value in an enumerated type. When you use enumerated data in a Switch Case block, follow these rules:

- If the input `u1` is of an enumerated type, all case condition values must be of that same enumerated type.
- If any case condition value is of an enumerated type, the input `u1` and all other case condition values must be of that same enumerated type.
- When the case condition values are of an enumerated type, each value that appears as a case condition must have a different underlying integer.

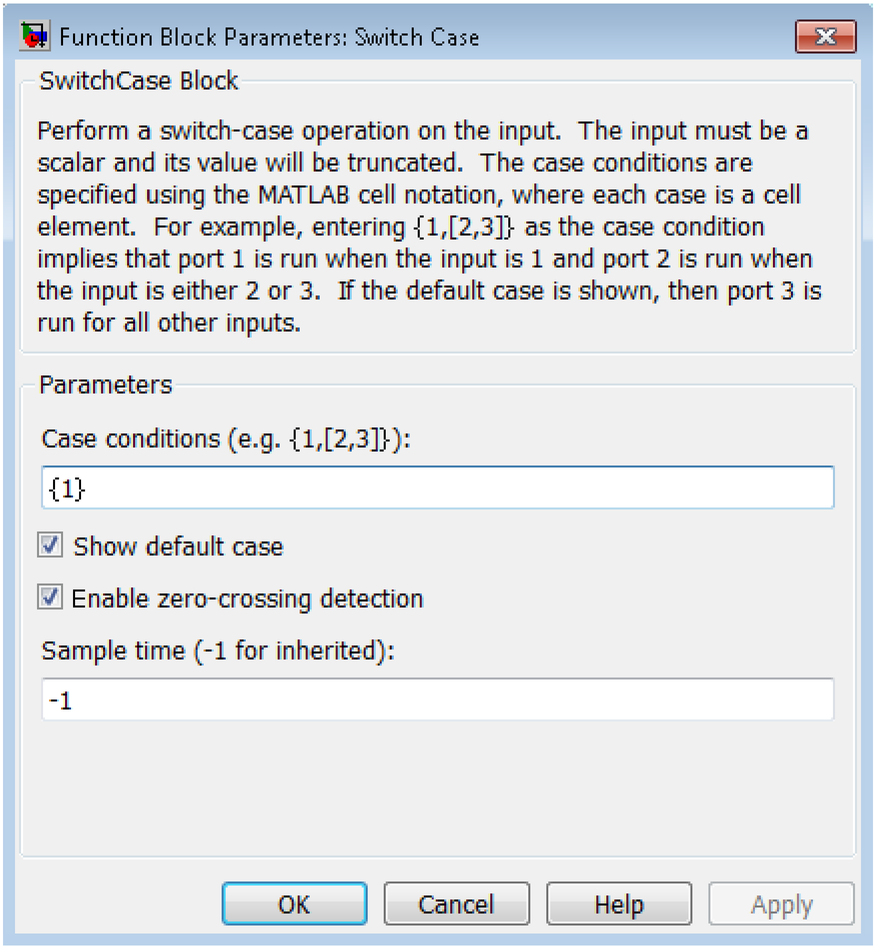
For more information about enumerated data types in Simulink, see "Use Enumerated Data in Simulink Models". The generated code for a Switch Case block that uses enumerated data uses the enumerated values of the specified case conditions, rather than their underlying integers.

Data Type Support

The input to the port labeled `u1` of a Switch Case block can be:

- A scalar value having a built-in data type that Simulink supports. The block does not support Boolean or fixed-point data types and truncates the numeric inputs to 32-bit signed integers.
- A scalar value of any enumerated data type, as described in "Enumerated Types in Switch Case Blocks" on page 1-1745.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



SwitchCase Block

Perform a switch-case operation on the input. The input must be a scalar and its value will be truncated. The case conditions are specified using the MATLAB cell notation, where each case is a cell element. For example, entering {1,[2,3]} as the case condition implies that port 1 is run when the input is 1 and port 2 is run when the input is either 2 or 3. If the default case is shown, then port 3 is run for all other inputs.

Parameters

Case conditions (e.g. {1,[2,3]}):

{1}

☑ Show default case

☑ Enable zero-crossing detection

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Case conditions

Specify the case conditions using MATLAB cell notation. For example, entering `{1,[7,9,4]}` specifies that output port case[1] is run when the input value is `1`, and output port case[7 9 4] is run when the input value is `7`, `9`, or `4`.

You can use colon notation to specify a range of integer case conditions. For example, entering `{[1:5]}` specifies that output port case[1 2 3 4 5] is run when the input value is `1`, `2`, `3`, `4`, or `5`.

Depending on block size, cases with long lists of conditions are displayed in shortened form in the Switch Case block, using a terminating ellipsis (...).

You can use the `enumeration` function to specify a case condition that includes a case for every value in an enumerated type.

Show default case

If you select this check box, the default output port appears as the last case on the Switch Case block, allowing you to specify a default case. This case executes when the input value does not match any of the case values specified in the **Case conditions** field. With **Show default case** selected, a default output port always appears, even if the preceding cases exhaust all possibilities for the input value.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

Characteristics

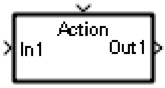
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	No
Zero-Crossing Detection	Yes, if enabled

Switch Case Action Subsystem

Purpose Represent subsystem whose execution is triggered by Switch Case block

Library Ports & Subsystems

Description



This block is a Subsystem block that is preconfigured to serve as a starting point for creating a subsystem whose execution is triggered by a Switch Case block.

Note All blocks in a Switch Case Action Subsystem must run at the same rate as the driving Switch Case block. You can achieve this by setting each block's sample time parameter to be either inherited (`-1`) or the same value as the Switch Case block's sample time.

For more information, see "Action Subsystem, If Action Subsystem, Switch Case Action Subsystem", Switch Case block and "Control Flow Logic" in the "Creating a Model" chapter of the Simulink documentation.

Purpose

Delay scalar signal multiple sample periods and output all delayed versions

Library

Discrete

Description

The Tapped Delay block delays an input by the specified number of sample periods and outputs all the delayed versions. Use this block to discretize a signal in time or resample a signal at a different rate.

The block accepts one scalar input and generates an output vector that contains each delay. Specify the order of the delays in the output vector with the **Order output vector starting with** parameter:

- `Oldest` orders the output vector starting with the oldest delay version and ending with the newest delay version.
- `Newest` orders the output vector starting with the newest delay version and ending with the oldest delay version.

Specify the output vector for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior.

Specify the time between samples with the **Sample time** parameter. Specify the number of delays with the **Number of delays** parameter. A value of `-1` instructs the block to inherit the number of delays by back propagation. Each delay is equivalent to the z^{-1} discrete-time operator, which the Unit Delay block represents.

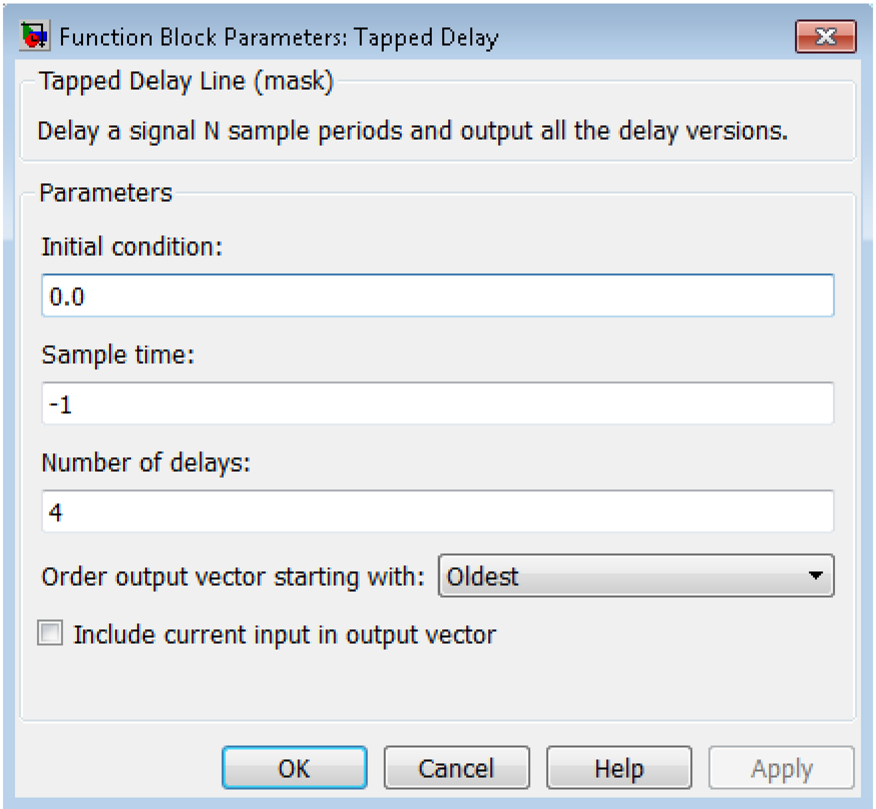
Data Type Support

The Tapped Delay block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box


Function Block Parameters: Tapped Delay
Tapped Delay Line (mask)

Delay a signal N sample periods and output all the delay versions.

Parameters

Initial condition: 0.0

Sample time: -1

Number of delays: 4

Order output vector starting with: Oldest

☐ Include current input in output vector

OK
Cancel
Help
Apply

Initial condition

Specify the initial output of the simulation. The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation. Simulink software does not allow you to set the initial condition of this block to `inf` or `NaN`.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online Simulink documentation for more information.

Number of delays

Specify the number of discrete-time operators.

Order output vector starting with

Specify whether to output the oldest delay version first, or the newest delay version first.

Include current input in output vector

Select to include the current input in the output vector.

Characteristics

Direct Feedthrough	Yes, when **Include current input in output vector** check box is selected. No, otherwise.
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of initial conditions

See Also

Unit Delay

Terminator

Purpose

Terminate unconnected output port

Library

Sinks

Description

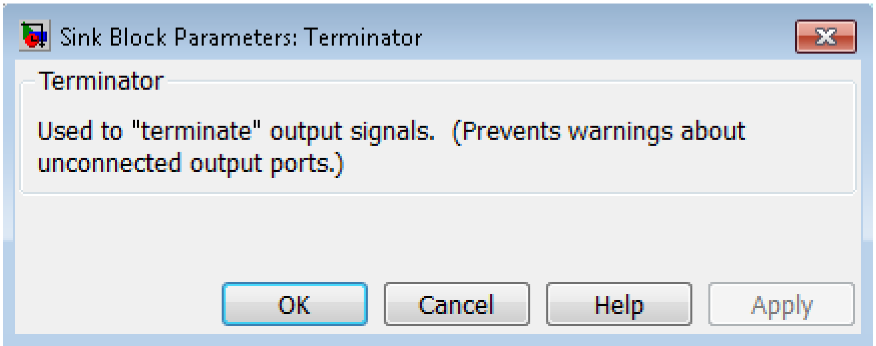
Use the Terminator block to cap blocks whose output ports do not connect to other blocks. If you run a simulation with blocks having unconnected output ports, Simulink issues warning messages. Using Terminator blocks to cap those blocks helps prevent warning messages.

Data Type Support

The Terminator block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Sink Block Parameters: Terminator

Terminator

Used to "terminate" output signals. (Prevents warnings about unconnected output ports.)

Examples

The following Simulink examples show how to use the Terminator block:

- `sldemo_bounce`

- `sldemo_fuelsys`
- `aero_six_dof`

Characteristics

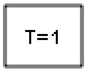
Sample Time	Inherited from the driving block
Dimensionalized	Yes
Multidimensionalized	Yes
Virtual	Yes For more information, see “Virtual Blocks” in the Simulink documentation.
Zero-Crossing Detection	No

Timed-Based Linearization

Purpose Generate linear models in base workspace at specific times

Library Model-Wide Utilities

Description This block calls `linmod` or `dlinmod` to create a linear model for the system when the simulation clock reaches the time specified by the **Linearization time** parameter. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.



The block sets the following model parameters to the indicated values:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`

The name of the structure used to save the snapshots is the name of the model appended by `_Timed_Based_Linearization`, for example, `vdp_Timed_Based_Linearization`. The structure has the following fields:

Field	Description
`a`	The A matrix of the linearization
`b`	The B matrix of the linearization
`c`	The C matrix of the linearization
`d`	The D matrix of the linearization
`StateName`	Names of the model's states
`OutputName`	Names of the model's output ports
`InputName`	Names of the model's input ports

Field	Description
`OperPoint`	A structure that specifies the operating point of the linearization. The structure specifies the operating point time (`OperPoint.t`). The states (`OperPoint.x`) and inputs (`OperPoint.u`) fields are not used.
`Ts`	The sample time of the linearization for a discrete linearization

Use the Trigger-Based Linearization block if you need to generate linear models conditionally.

You can use state and simulation time logging to extract the model states and inputs at operating points. For example, suppose that you want to get the states of the f14 example model at linearization times of 2 seconds and 5 seconds.

1. Open the model and drag an instance of this block from the Model-Wide Utilities library and drop the instance into the model.
2. Open the block's parameter dialog box and set the **Linearization time** to 2 and 5.
3. Open the model's **Model Configuration Parameters** dialog box.
4. Select the **Data Import/Export** pane.
5. Check **States** and **Time** on the **Save to Workspace** control panel
6. Select OK to confirm the selections and close the dialog box.
7. Simulate the model.

   At the end of the simulation, the following variables appear in the MATLAB workspace: `f14_Timed_Based_Linearization`, `tout`, and `xout`.

8 Get the indices to the operating point times by entering the following at the MATLAB command line:

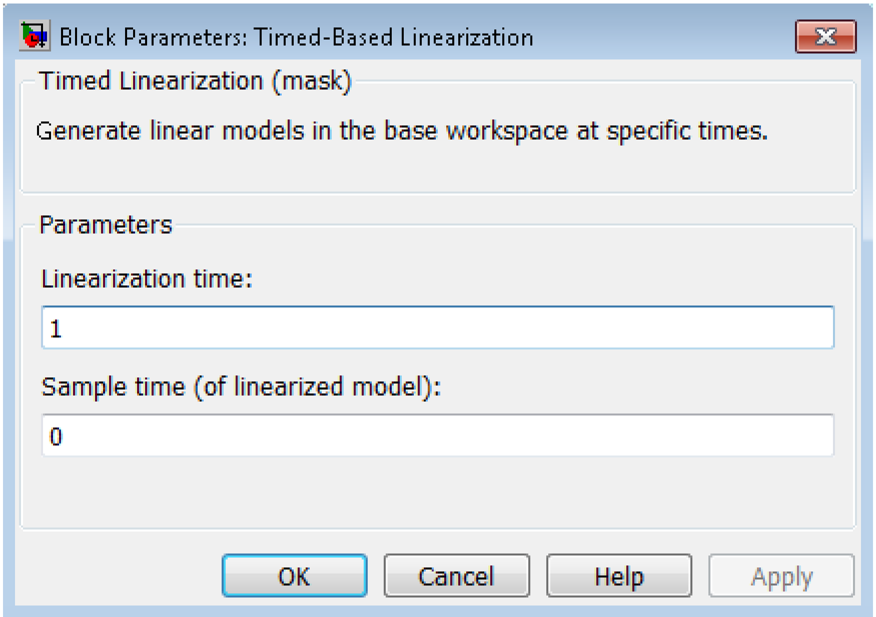
```
ind1 = find(f14_Timed_Based_Linearization(1).OperPoint.t==tout);
ind2 = find(f14_Timed_Based_Linearization(1).OperPoint.t==tout);
```

9 Get the state vectors at the operating points.

```
x1 = xout(ind1,:);
x2 = xout(ind2,:);
```

Data Type Support

Not applicable.


Block Parameters: Timed-Based Linearization
Timed Linearization (mask)
Parameters
OK
Cancel
Help
Apply

Generate linear models in the base workspace at specific times.

Linearization time:

1

Sample time (of linearized model):

0

Parameters and Dialog Box

Linearization time
Time at which you want the block to generate a linear model. Enter a vector of times if you want the block to generate linear models at more than one time step.

Sample time (of linearized model)
Specify a sample time to create discrete-time linearizations of the model (see "Discrete-Time System Linearization" on page 2-28).

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	No

See Also

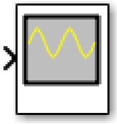
Trigger-Based Linearization

Time Scope

Purpose

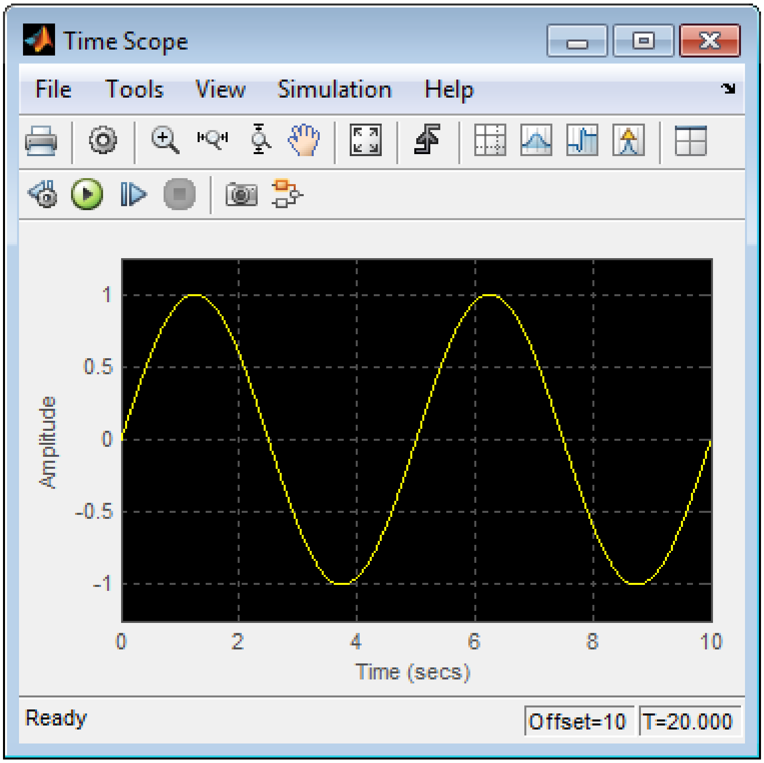
Display time-domain signals

Description



The Time Scope block displays signals in the time domain. The Time Scope block accepts input signals with the following characteristics:

- Continuous or discrete sample time
- Real- or complex-valued
- Fixed or variable size dimensions
- Floating- or fixed-point data type
- N-dimensional
- Simulink enumerations


Time Scope
File
Tools
View
Simulation
Help
Amplitude
1
0.5
0
-0.5
-1
0
2
4
6
8
10
Time (secs)
Ready
Offset=10
T=20.000

You can use the Time Scope block in models running in Normal or Accelerator simulation modes. You can also use the Time Scope block in models running in Rapid Accelerator or External simulation modes, with some limitations. See the "Supported Simulation Modes" on page 1-1856 section for more information.

You can use the Time Scope block inside of all subsystems and conditional subsystems. *Conditional subsystems* include enabled subsystems, triggered subsystems, enabled and triggered subsystems, and function-call subsystems. See "About Conditional Subsystems" in the Simulink documentation for more information.

See the following sections for more information on the Time Scope:

- “Displaying Multiple Signals” on page 1-1762
- “Signal Display” on page 1-1768
- “Toolbar” on page 1-1772
- “Simulation Toolbar” on page 1-1777
- “Measurements Panels” on page 1-1779
- “Configuration Properties Dialog Box” on page 1-1821
- “Style Dialog Box” on page 1-1834
- “Simulation Stepping Options” on page 1-1837
- “Core — General UI Options” on page 1-1839
- “Visuals — Time Domain Properties” on page 1-1842
- “Tools — Plot Navigation Options” on page 1-1851
- “Sources — Simulink Properties” on page 1-1854
- “Supported Data Types” on page 1-1855
- “Supported Simulation Modes” on page 1-1856

Displaying Multiple Signals

Multiple Signal Input

You can configure the Time Scope block to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have different dimensions, sample rates, and data types. Each signal can be either real or complex valued. You can set the number of input ports on the Time Scope block in the following ways:

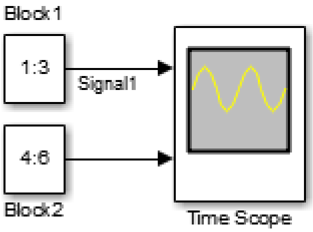
- Right-click the Time Scope block in your model to bring up the context menu. Point your cursor to the **Signals & Ports > Number of Input Ports** item on the context menu. You can then select the number of input ports for the Time Scope block. If the desired number of input signals is 1, 2, or 3, then click on the appropriate value. To configure a Time Scope block to have more than three input ports, select **More**, and enter a number for the **Number of input ports** parameter.

- Open the Time Scope window by double-clicking the Time Scope block in your model. To specify how many input ports the Time Scope block should have, in the scope menu, select **File > Number of Input Ports**.

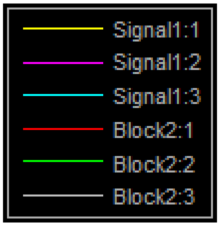
An input signal may contain multiple channels, depending on its dimensions. Multiple channels of data are always shown as different-colored lines on the same display.

Multiple Signal Names

By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal named `Signal1` would have the following default names in the channel legend: `Signal1:1`, `Signal1:2`. In the following example, there are two 3-channel input signals to the scope block, one signal named `Signal1` and one unnamed signal coming from a block named `Block2`.



To see all the signal names, run the simulation and show the legend. To show the legend, select **View > Configuration Properties**, click the **Display** tab, and select the **Show Legend** check box. The following legend appears in the display.

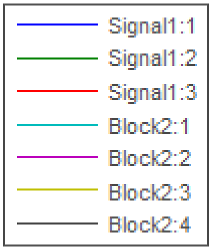


Multiple Signal Colors

By default, the scope has a black axes background and chooses line colors for each channel in the same manner as the Simulink Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the following order:

1. Yellow
2. Magenta
3. Cyan
4. Red
5. Green
6. Gray

If there are more than 6 channels, then the scope repeats this order to assign line colors to the remaining channels. To choose line colors for each channel in the same manner as the MATLAB `plot` function, change the axes background color to any color except black. To change the axes background color to white, select **View > Style**, click the Axes background color button (), and select white from the color palette. Run the simulation again. The following legend appears in the display.



When the scope axes background is not black, it assigns each channel of each input signal a line color in the following order:

1. Blue

2. Dark Green

3. Red

4. Cyan

5. Purple

6. Dark Yellow

7. Black

If there are more than 7 channels, then the scope repeats this order to assign line colors to the remaining channels. To manually modify any line color, select **View > Style** to open the Style dialog box. Next to **Properties for line**, select the signal name whose color you want to change. Then, next to **Line**, click the Line color button (✎) and select any color from the palette.

Multiple Time Offsets

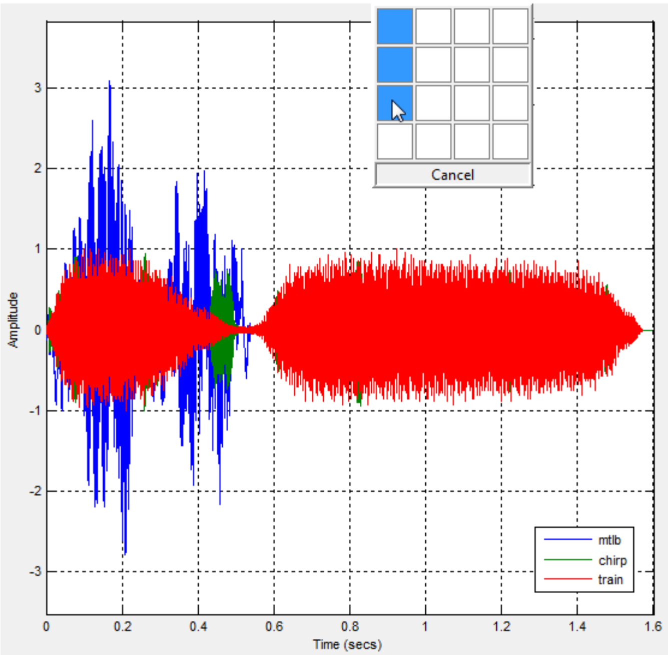
You can offset all channels of an input signal by the same number of seconds or offset each channel independently. To offset all channels equally, select **View > Configuration Properties**, and specify a scalar value for the **Time display offset** parameter on the **Main** pane.

To offset each channel independently, specify a vector of offset values. When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

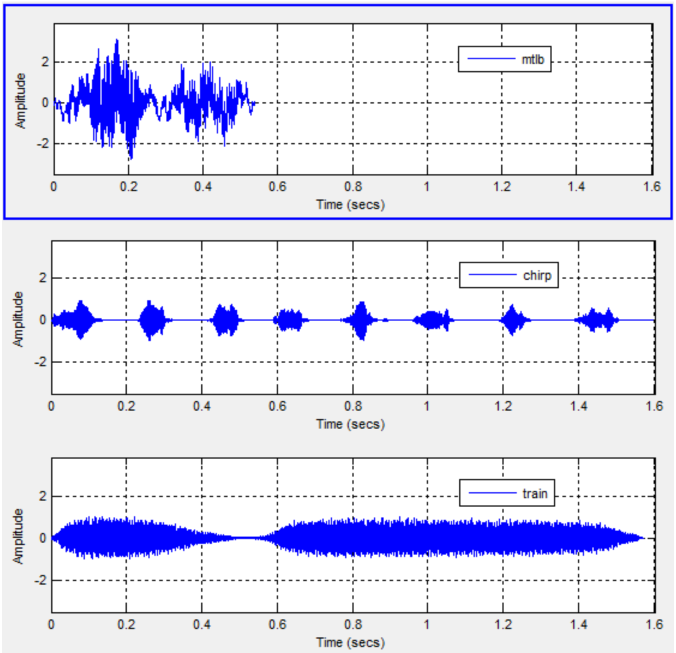
- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.
- When *N* is less than the number of input channels, the scope applies the values you specify in the offset vector to the first *N* input channels. The scope does not offset the remaining channels.
- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

Multiple Displays

You can display multiple channels of data on different displays in the scope window. In the scope toolbar, select **View > Layout**, or select the Layout button (⊞). This feature allows you to tile the window into a number of separate displays, up to a grid of 4 rows and 4 columns. For example, if there are three inputs to the scope, you can display the signals in separate displays by selecting row 3, column 1, as shown in the following figure.


Cancel
3
2
1
0
-1
-2
-3
Amplitude
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
Time (secs)
mtlb
chirp
train

After you select row 3, column 1, the scope window is partitioned into three separate displays, as shown in the following figure.



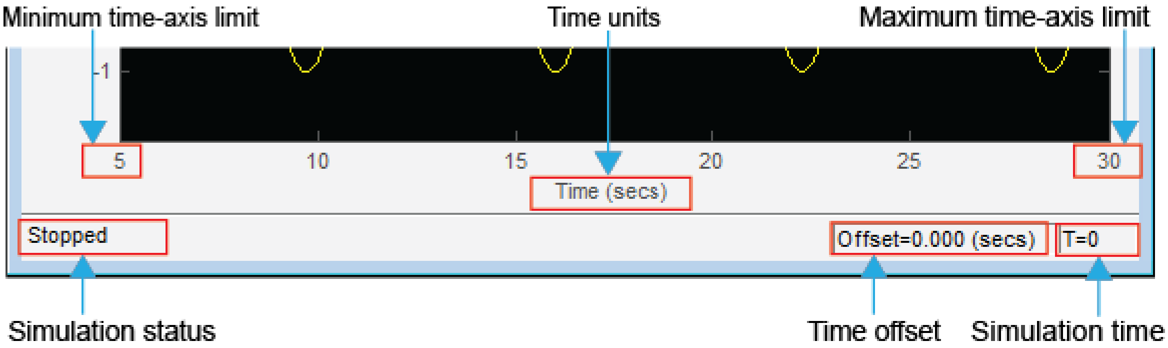
When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The scope dialog boxes reference the active display.

Signal Display

Time Scope uses the simulation start time and stop time in order to determine the default time range. However, you can define the length of simulation time for which the Time Scope displays data. To change the signal display settings, select **View > Configuration Properties** to bring up the Configuration Properties dialog box. Then, modify the values for the **Time span** and **Time display offset** parameters on the **Time** tab. For example, if you set the **Time span** to `20` seconds and the **Time display offset** to `0`, the scope displays 20 seconds' worth of simulation data at a time. The values on the *time*-axis of the Time Scope display remain the same throughout simulation.

To communicate the simulation time that corresponds to the current display, the scope uses the **Time units**, **Time offset**, and **Simulation**

time indicators on the scope window. The following figure highlights these and other important aspects of the Time Scope window.


Minimum time-axis limit
Time units
Maximum time-axis limit
-1
5
10
15
20
25
30
Time (secs)
Stopped
Offset=0.000 (secs)
T=0
Simulation status
Time offset
Simulation time

- **Minimum time-axis limit** — The Time Scope sets the minimum *time*-axis limit using the value of the **Time display offset** parameter on the **Main** tab of the Visuals—Time Domain Properties dialog box. If you specify a vector of values for the **Time display offset** parameter, the scope uses the smallest of those values to set the minimum *time*-axis limit.

- **Maximum time-axis limit** — The Time Scope sets the maximum *time*-axis limit by summing the value of **Time display offset** parameter with the value of the **Time span** parameter. If you specify a vector of values for the **Time display offset** parameter, the scope sets the maximum *time*-axis limit by summing the largest of those values with the value of the **Time span** parameter.

- **Simulation status** — Provides the current status of the model simulation. The status can be one of the following conditions:
  - `Initializing`
  - `Ready`
  - `Running`
  - `Paused`

The **Simulation status** is part of the **Status Bar** in the Time Scope window. You can choose to hide or display the entire **Status Bar**. From the Time Scope menu, select **View > Status Bar**.

- **Time units** — The units used to describe the *time*-axis. The Time Scope sets the time units using the value of the **Time Units** parameter on the **Time** tab of the Configuration Properties dialog box. By default, this parameter is set to `Metric (based on Time Span)` and displays in metric units such as milliseconds, microseconds, minutes, days, etc. You can change it to `Seconds` to always display the *time*-axis values in units of seconds. You can change it to `None` to not display any units on the *time*-axis. When you set this parameter to `None`, then Time Scope shows only the word `Time` on the *time*-axis.

  To hide both the word `Time` and the values on the *time*-axis, set the **Show time-axis labels** parameter to `None`. To hide both the word `Time` and the values on the *time*-axis in all displays except the bottom ones in each column of displays, set this parameter to `Bottom Displays Only`. This behavior differs from the Simulink Scope block, which always shows the values but never shows a label on the *x*-axis.

- **Time offset** — The **Time offset** value helps you determine the simulation times for which the scope is displaying data. The value is always in the range `0`≤ `Time offset`≤ **Simulation time**. Therefore, add the Time offset to the fixed time span values on the *time*-axis to get the overall simulation time.

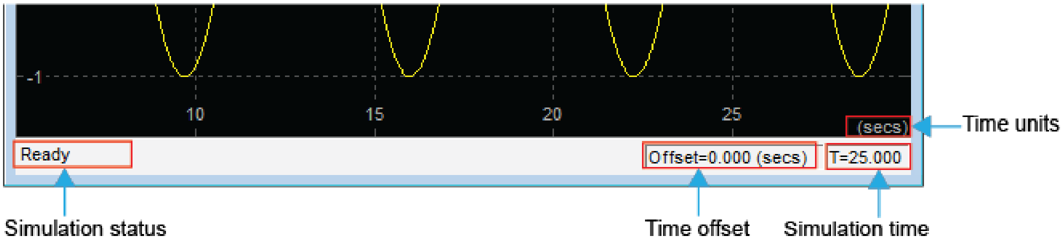
  For example, if you set the **Time span** to `20` seconds, and you see a **Time offset** of `0 (secs)` on the scope window. This value indicates that the scope is displaying data for the first `0` to `20` seconds of simulation time. If the **Time offset** changes to `20 (secs)`, the scope displays data for simulation times from `20` seconds to `40` seconds. The scope continues to update the **Time offset** value until the simulation is complete.

- **Simulation time** — When the model is running or simulation has been paused, the scope displays the current simulation time. If the model simulation completes or is stopped, the scope displays the time at which the simulation stopped. The **Simulation time** is part of the **Status Bar** in the Time Scope window. You can choose to hide or display the entire **Status Bar**. From the Time Scope menu, select **View > Status Bar** .

Note In some situations, the Time Scope block simulation time can be different from the Simulink simulation time. For multirate input signals, which have different sample times, and input signals originating from conditionally-executed subsystems, such as Triggered and Enabled subsystems, separate Time Scope blocks may report different simulation times. The Time Scope block reports the simulation time as the time corresponding to the last point in the display.

Axes Maximization

When the scope is in maximized axes mode, the following figure highlights the important indicators on the scope window.



To toggle this mode, in the scope menu, select **View > Configuration Properties**. In the **Main** pane, locate the **Maximize axes** parameter.

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable.

The default setting is `Auto`.

Reduce Updates to Improve Performance

By default, the scope updates the displays periodically at a rate not exceeding 20 hertz. If you would like the scope to update on every simulation time step, you can disable the **Reduce Updates to Improve Performance** option. However, as a recommended practice, leave this option enabled because doing so can significantly improve the speed of the simulation.

In the Time Scope menu, select **Simulation > Reduce Updates to Improve Performance** to clear the check box. Alternatively, use the **Ctrl+R** shortcut to toggle this setting.

Toolbar

The Time Scope toolbar contains the following buttons.

Print and Configuration Properties Buttons

Button	Menu Location	Shortcut Keys	Description
🖨	**File > Print**	**Ctrl+P**	Print the current scope window. Printing is available only when the scope display is not changing. You can enable printing by placing the scope in snapshot mode, pausing or stopping model simulation, or simulating the model in step-forward mode. To print the current scope window to a figure rather than sending it to your printer, select **File > Print to figure**.
⚙	**View > Configuration Properties**	N/A	Open the **Configuration Properties** dialog box. See the “Configuration Properties Dialog Box” on page 1-1821 section for more information.

Axes Control Buttons

🔍	**Tools > Zoom In**	N/A	When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw a rectangular area of interest inside the scope window.
↔	**Tools > Zoom X**	N/A	When this tool is active, you can zoom in on the *x*-axis. To do so, click inside the scope window, or click and drag your cursor along the *x*-axis over your area of interest.

🔍	**Tools > Zoom Y**	N/A	When this tool is active, you can zoom in on the *y*-axis. To do so, click inside the scope window, or click and drag your cursor along the *y*-axis over your area of interest.
✋	**Tools > Pan**	N/A	When this tool is active, you can pan on the scope window. To do so, click in the center of your area of interest and drag your cursor to the left, right, up, or down, to move the position of the display.
⛶	**Tools > Scale Axes Limits**	**Ctrl+A**	Click this button to scale the axes in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation.

			• **Scale Axes Limits after 10 Updates** — When you select this option, the scope scales the axes after 10 updates. The scope does not scale the axes again during the simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Measurements Buttons

	Tools > Triggers	N/A	Open or close the **Triggers** panel. This panel allows you to pause the display only when certain events occur. You can use the Triggers panel when you want to align or search for interesting events. Triggers can be configured to both select and align specific regions of interest in the display area of the scope. See the “Triggers Panel” on page 1-1781 section for more information.
⊞	**Tools > Measurements > Cursor Measurements**	N/A	Open or close the **Cursor Measurements** panel. This panel puts screen cursors on all the displays. See the “Cursor Measurements Panel” on page 1-1800 section for more information.

	Tools > Measurements > Signal Statistics	N/A	Open or close the **Signal Statistics** panel. This panel displays the maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur. See the “Signal Statistics Panel” on page 1-1802 section for more information.
	Tools > Measurements > Bilevel Measurements	N/A	Open or close the **Bilevel Measurements** panel. This panel displays information about a selected signal’s transitions, overshoots or undershoots, and cycles. See the “Bilevel Measurements Panel” on page 1-1804 section for more information.
	Tools > Measurements > Peak Finder	N/A	Open or close the **Peak Finder** panel. This panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified. See the “Peak Finder Panel” on page 1-1818 section for more information.

Other Buttons

⊞	**View > Layout**	N/A	Arrange the layout of displays in the Time Scope. This feature allows you to tile your screen into a number of separate displays, up to a grid of 4 rows and 4 columns. You may find multiple displays useful when the Time Scope takes multiple input signals. The default display is 1 row and 1 column. See the "Multiple Displays" on page 1-1766 section for more information.

You can control whether this toolbar appears in the Time Scope window. From the Time Scope menu, select **View > Toolbar**.

Simulation Toolbar

The Simulation Toolbar contains the following buttons.

Button	Menu Location	Shortcut Keys	Description
⚙	**Simulation > Simulation Stepping Options**	N/A	Open the **Simulation Stepping Options** dialog box. This button appears only when you have previous stepping disabled. See the "Simulation Stepping Options" on page 1-1837 section for more information.
◁‖	**Simulation > Previous Step**	N/A	Advance the model simulation backward by one time step. This button appears only when you have previous stepping enabled and the model simulation is paused.

Button	Menu Location	Shortcut Keys	Description
▶	**Simulation > Run**	**Ctrl+T**, **p**, **Space**	Start the model simulation. This button appears only when the model simulation is stopped.
▶	**Simulation > Continue**	**p**, **Space**	Continue the model simulation. This button appears only when the model simulation is paused.
⏸	**Simulation > Pause**	**p**, **Space**	Pause the model simulation. This button appears only when the model simulation is running.
⏵	**Simulation > Step Forward**	**Right arrow**, **Page Down**	Advance the model simulation forward by one time step. This button starts the model simulation, allows it to run for one time step, and then pauses it again. The scope window then updates with the latest data.
⏹	**Simulation > Stop**	**Ctrl+T**, **s**	Stop the model simulation. This button appears only when the model simulation is running or paused.
📷	**Simulation > Simulink Snapshot**	N/A	Take a snapshot of the current scope display. This button temporarily freezes the scope display, while allowing simulation to continue running. To unfreeze the scope display and view the

Button	Menu Location	Shortcut Keys	Description
			current simulation data, toggle this button to turn off snapshot mode.
	View > Highlight Simulink Block	**Ctrl+L**	Bring the model window forward, and highlight the Time Scope block whose display you are currently viewing. The Time Scope block that corresponds to the active Time Scope window flashes three times in the model.

You can control whether this toolbar appears in the Time Scope window. From the Time Scope menu, select **View > Simulation Toolbar**.

To see a full listing of the shortcut keys for these simulation controls, from the Time Scope menu, select **Help > Keyboard Command Help**.

Measurements Panels

The Measurements panels are the five panels that appear to the right side of the Time Scope GUI. These panels are labeled **Trace selection**, **Signal statistics**, **Bilevel measurements**, and **Peak finder**.

The Time Domain Measurements panels only appear if the **Measurements** tool is enabled in the Tools—Plot Navigation Options dialog box. To open this dialog box, in the Time Scope menu, select **File > Configuration** and click the **Tools** pane. If you disable the tool by clearing the **Enabled** check box, the Time Domain Measurements tools no longer display in the Time Scope figure. You can reenable the tool at any time by selecting the **Enabled** check box. See the "Tools — Plot Navigation Options" on page 1-1851 section for more information.

Measurements Panel Buttons

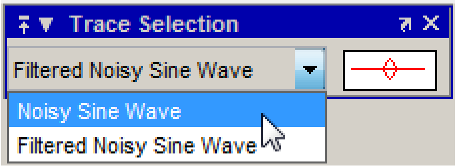
Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

Button	Description
⤒	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
▼	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all of its panes to conserve space. After you click this button, it becomes the expand button ▶.
▶	Expand the current panel. This button appears after you click the collapse button to hide the panes in the current panel. Click this button to display the panes in the current panel and show measurements again. After you click this button, it becomes the collapse button ▼ again.
↗	Undock the current panel. This button lets you move the current panel into a separate window that can be relocated anywhere on your screen. After you click this button, it becomes the dock button ↘ in the new window.
↘	Dock the current panel. This button appears only after you click the undock button. Click this button to put the current panel back into the right side of the Scope window. After you click this button, it becomes the undock button ↗ again.
✕	Close the current panel. This button lets you remove the current panel from the right side of the Scope window.

Some panels have their measurements separated by category into a number of panes. Click the pane expand button ▶ to show each pane that is hidden in the current panel. Click the pane collapse button ▼ to hide each pane that is shown in the current panel.

Trace Selection Panel

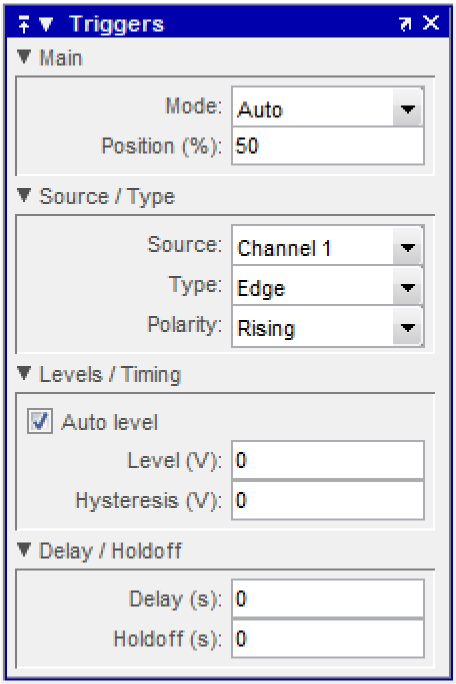
When you use the scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click on any of the other Measurements panels. The Measurements panels display information about only the signal chosen in this panel. Choose the signal name for which you would like to display time domain measurements. See the following figure.



You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools > Measurements > Trace Selection**.

Triggers Panel

The **Triggers** panel allows you to pause the display only when certain events occur. You can use the Triggers panel when you want to align or search for interesting events. You can be configure triggers to both select and align specific regions of interest in the display area of the scope. You can also choose to hide or display the **Triggers** panel. In the scope toolbar, click the Triggers button ( ). Alternatively, in the scope menu, select **Tools > Triggers**.



Mode: Auto
Position (%): 50
Source: Channel 1
Type: Edge
Polarity: Rising
☑ Auto level
Level (V): 0
Hysteresis (V): 0
Delay (s): 0
Holdoff (s): 0

When the **Triggers** panel is displayed, triangle pointers appear at the top and right side of the axes on each display. These markers indicate the time position (▼) and level (◀) at the event. The color of the markers corresponds to the color of the source signal.

Main Pane

The **Main** pane lets you choose how often the display updates and in what position the trigger indicator appears.

- **Mode** — Define how often the display should update.
  - **Auto** — The scope makes acquisitions when either a trigger event occurs or a periodic timer event expires.

  - **Normal** — The scope makes acquisitions only when a trigger event occurs.
  - **Once** — The scope makes acquisitions and updates display only once, for the first trigger event encountered.
  - **Off** — The scope does not make acquisitions. Triggering is disabled. This setting is equivalent to hiding the **Triggers** panel.

  If mode is set to either **Normal** or **Once** and the Triggers panel does not encounter any event, the display remains blank. Set **Mode** to **Auto** if you want the scope to display signal data regularly, in addition to trigger events.

  Note When **Mode** is set to anything except **Off**, the scope disables panning.

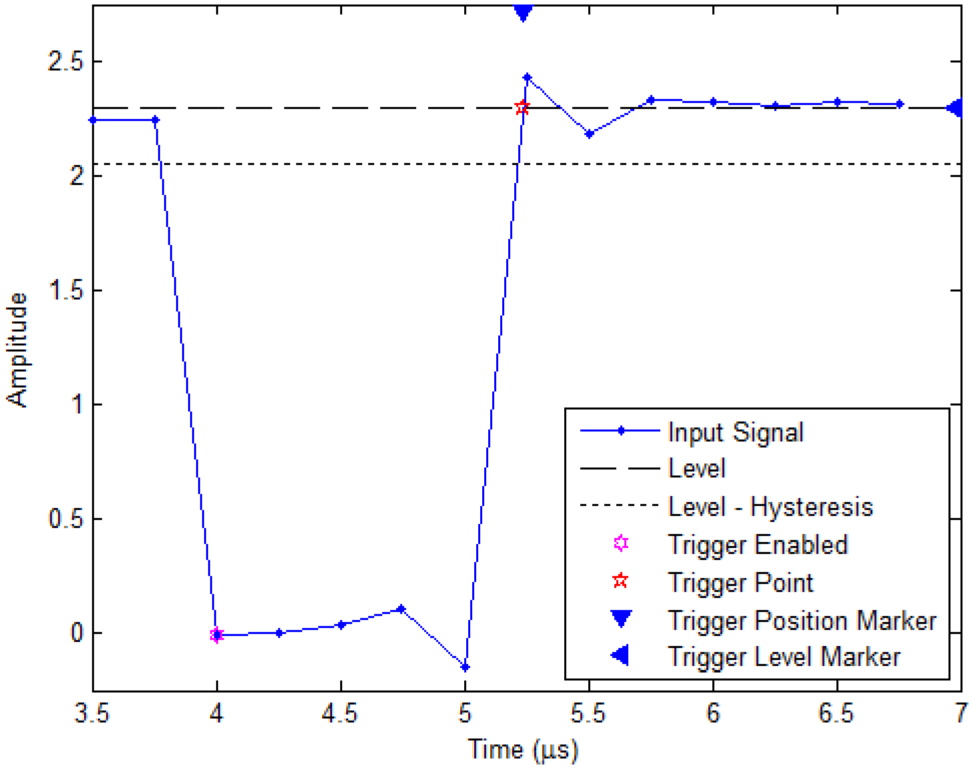
- **Position (%)** — Specify, as a percentage of the total time span within the active display, the horizontal position in which the trigger indicator appears. A position value of `0` corresponds to the minimum *time*-axis value at the far-left side of the display. A position value of `100` corresponds to the maximum *time*-axis value at the far-right side of the display. Drag the trigger position indicator to the left or right to adjust its position.

Source / Type Pane

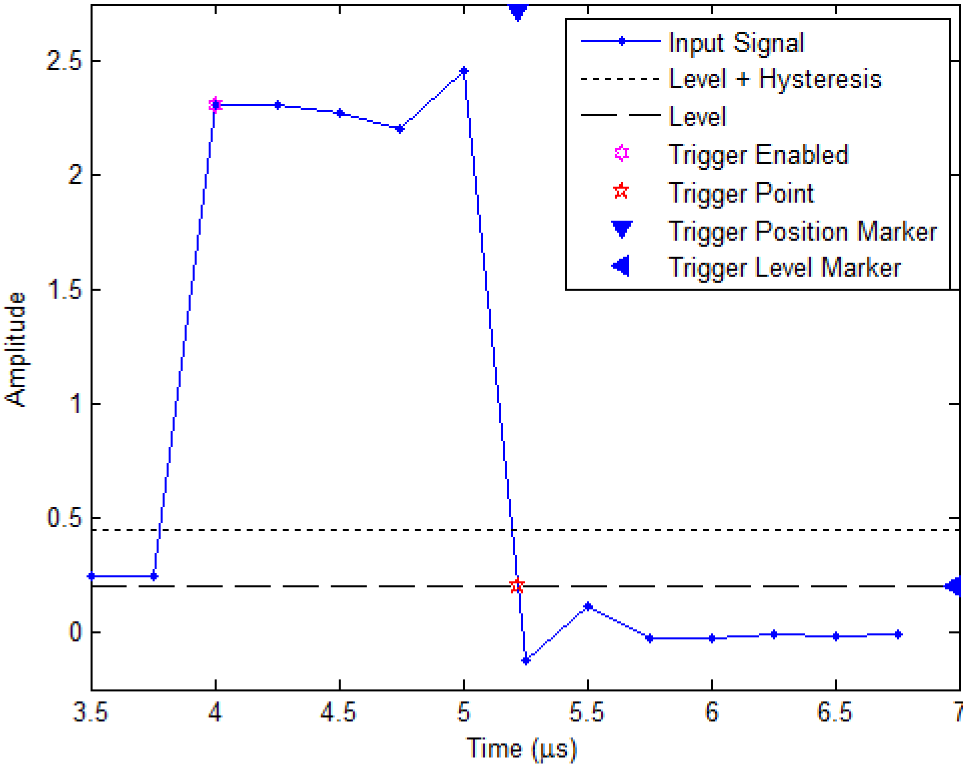
The **Source / Type** pane lets you choose the source of the trigger and the type of events on which to stop.

- **Source** — Assign the trigger source to a particular channel. If the input signal has multiple channels, the scope assigns an index number to identify each channel of that signal. For more information, see "Multiple Signal Input".
- **Type** — Select the type of trigger to use.

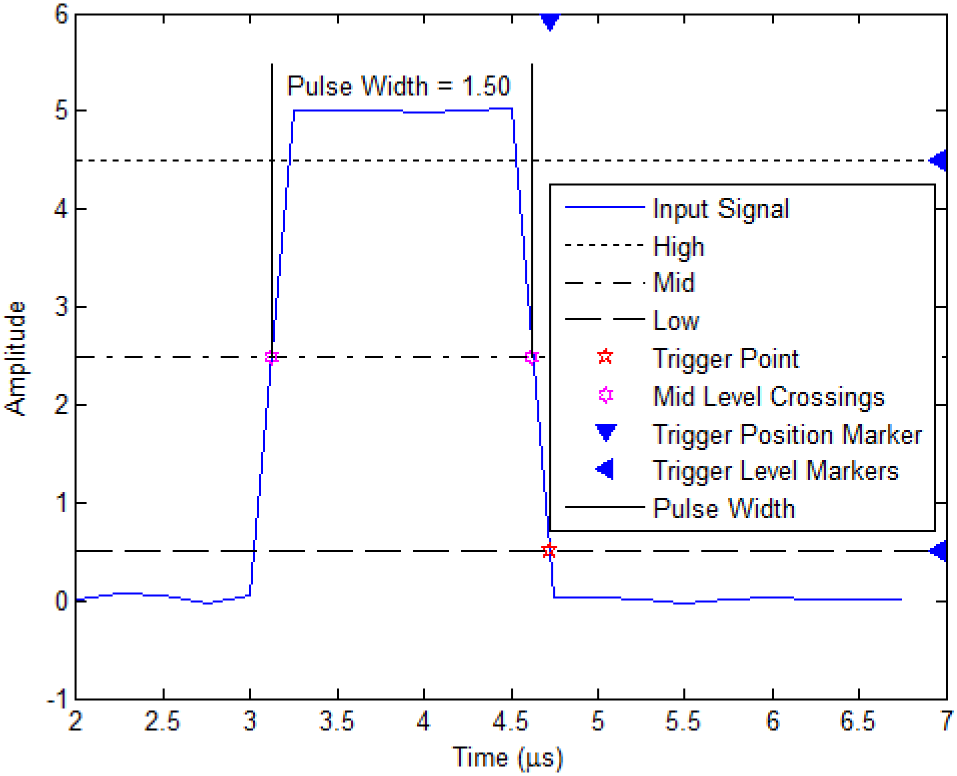
- **Edge** — Trigger when the scope crosses a level threshold. In the case of a rising edge, the scope enables the trigger event when the signal value becomes less than the level threshold minus hysteresis. The scope disables the trigger event when the signal becomes greater than the level threshold for the first time. The scope uses linear interpolation to generate a trigger event at the time when the signal crosses the level threshold, as shown in the following figure.



In the case of a falling edge, the scope enables the trigger event when the signal value becomes greater than the level threshold plus hysteresis. The scope disables the trigger event when the signal becomes less than the level threshold for the first time. The scope uses linear interpolation to generate a trigger event at the time when the signal crosses the level threshold, as shown in the following figure.

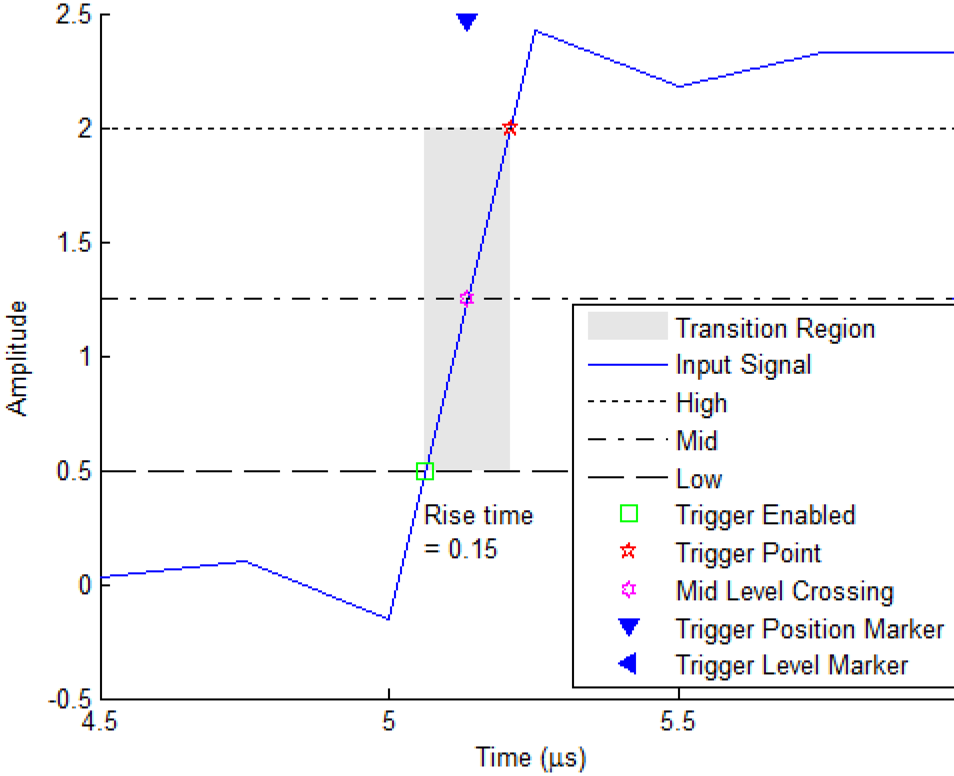


- **Pulse Width** — Trigger when the scope encounters a pulse whose width falls inside or outside specified time limits. You specify the range of valid time limits in the **Levels / Timing** pane. In the case of a positive-polarity pulse, the scope encounters a trigger event when the signal crosses the low threshold for the second time. The scope measures the pulse width as the time between the first and second crossings of the middle threshold, located halfway between the high and low thresholds, as shown in the following figure.

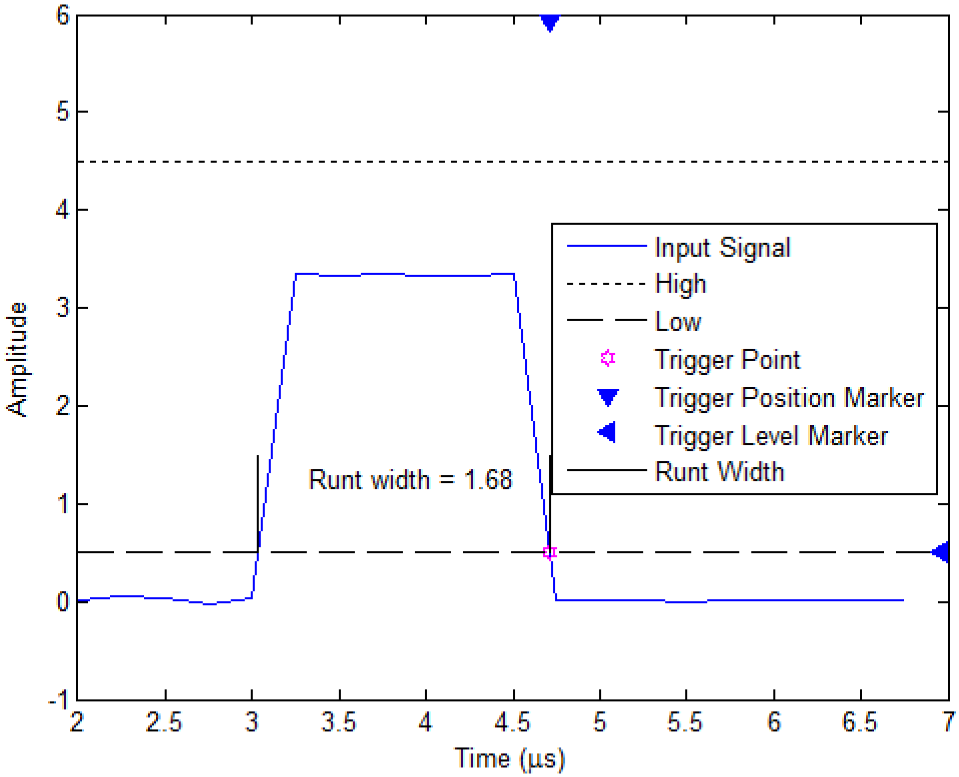


Note A *Glitch*-type trigger looks for a pulse or spike whose duration is less than a specified amount. You can implement a *Glitch* type trigger by using a **Pulse Width** type trigger and manually setting the **Max Width** parameter.

- **Transition** — Trigger on a rising or falling edge that crosses two levels, high and low, inside or outside a specified time interval. You specify the range of valid transition times in the **Levels / Timing** pane. In the case of a rising transition, the scope encounters the trigger event when the signal crosses the high threshold. The transition time is when the signal crosses the middle threshold, located halfway between the high and low thresholds, as shown in the following figure.

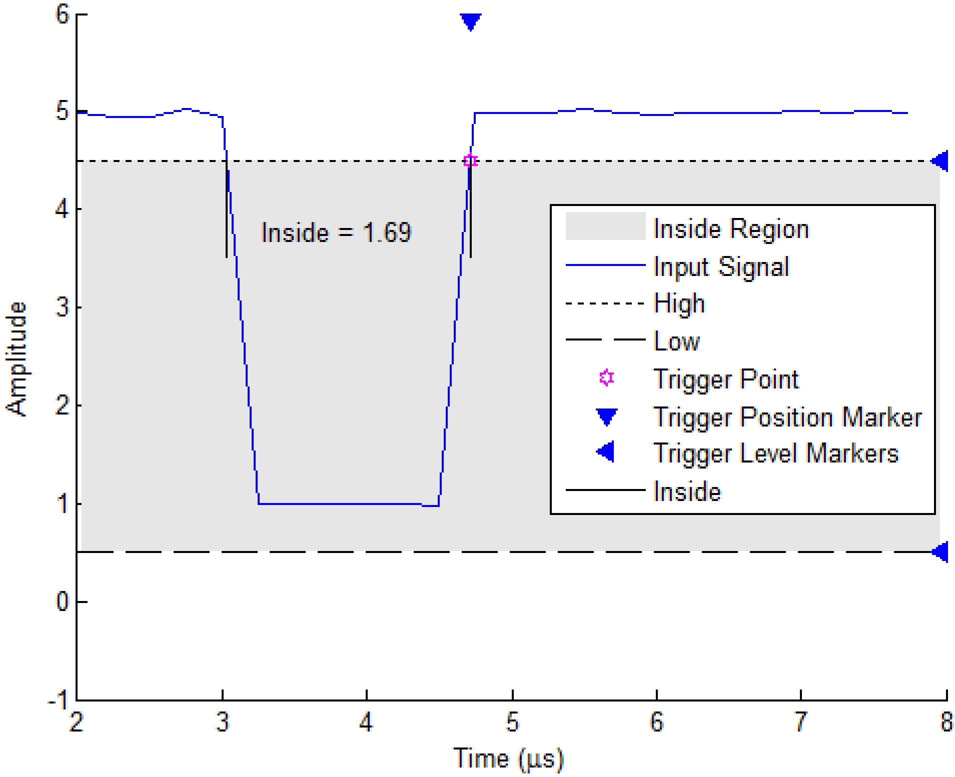


- **Runt** — Trigger on a runt pulse, which crosses one threshold, high or low, but not both. In the case of a positive-polarity runt pulse, the scope encounters a trigger event when the signal crosses the low threshold the second time, without ever crossing the high threshold. The scope measures the runt width as the time between the first and second crossings of the low threshold, as shown in the following figure.

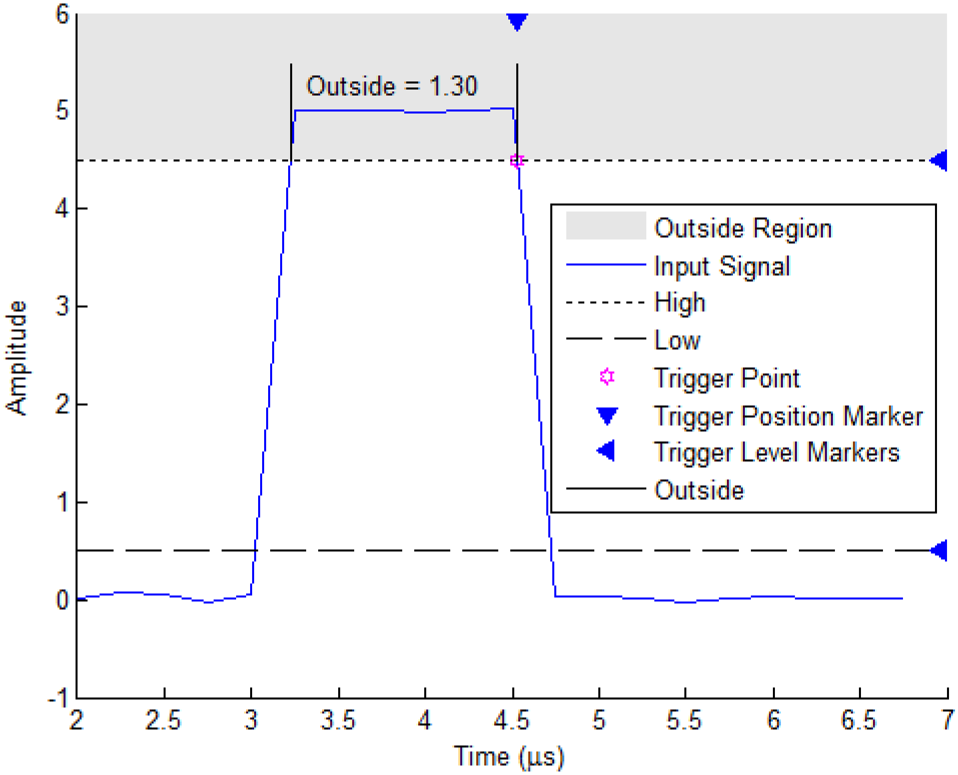


Note You can also replicate a **Runt**-type trigger by using a **Window**-type trigger and setting **Polarity** to **Inside**.

- **Window** — Trigger when the input signal stays within or outside the region defined by the high and low thresholds for a period of time. In the case of an inside window, the scope encounters a trigger event when the signal enters and exits the inside region, as shown in the following figure.

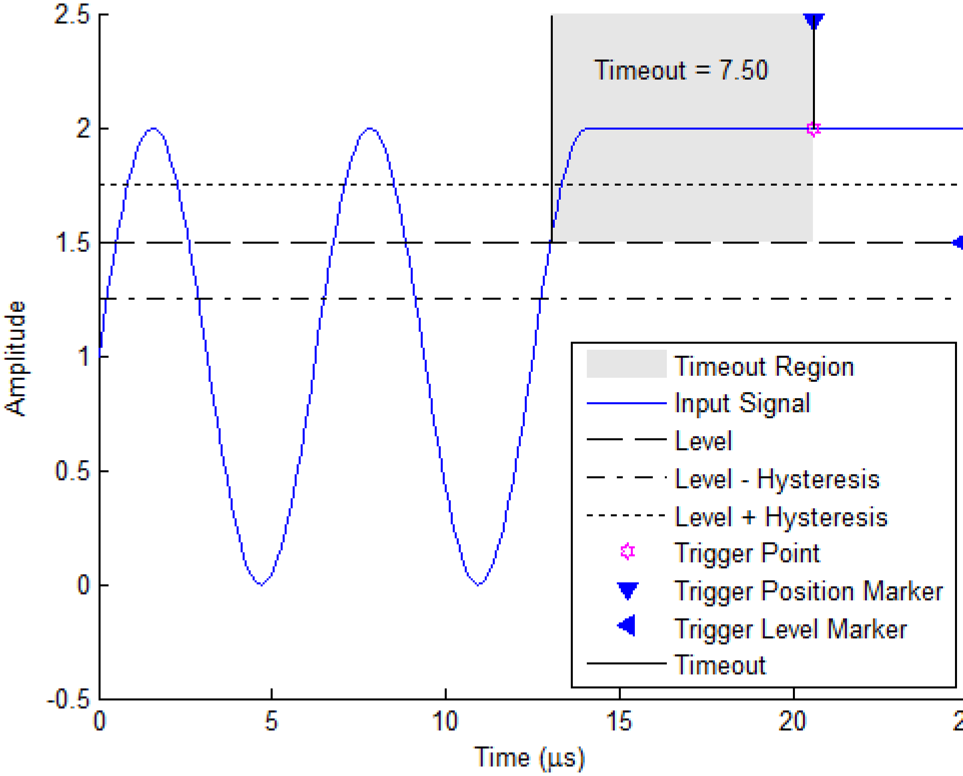


In the case of an outside window, the scope encounters a trigger event when the signal enters and exits the outside region, as shown in the following figure.

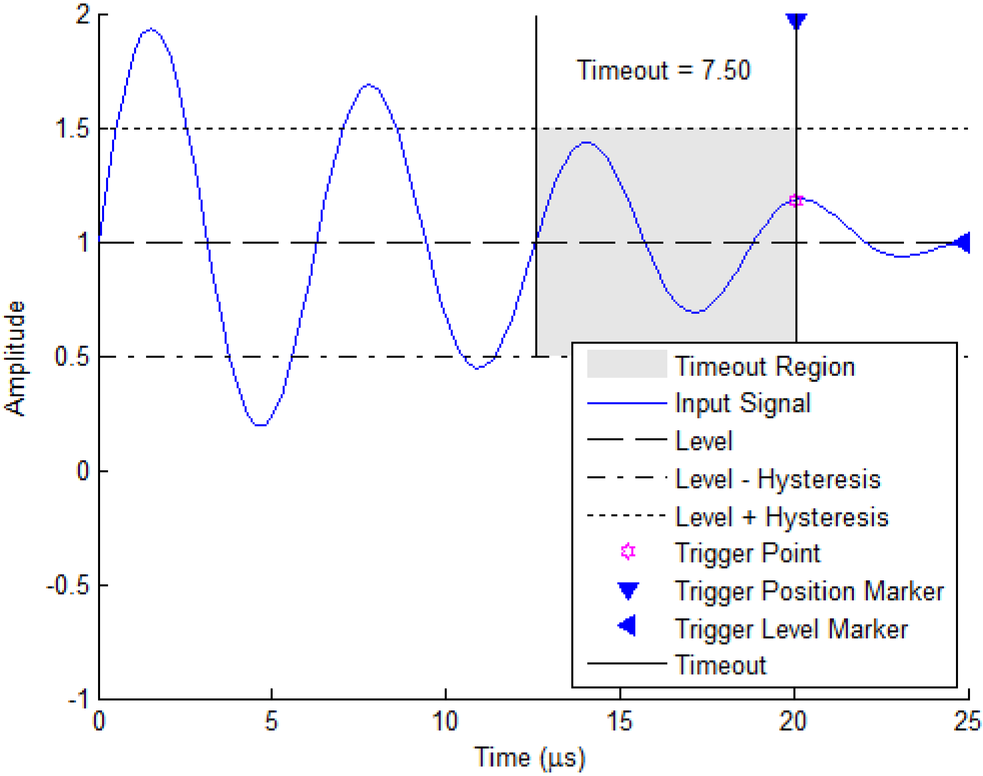


The scope encounters a trigger event when the signal crosses either the high or low threshold the second time.

- **Timeout** — Trigger when the input signal stays above or below a voltage threshold longer than a specified time. In the case of a timeout trigger with polarity set to `Either` and a timeout duration of 7.50 seconds, the scope can encounter the trigger event 7.50 seconds after the signal crosses the level threshold the last time, as shown in the following figure.



Alternatively, the scope can encounter the trigger event when the signal stays within the boundaries defined by the hysteresis for 7.50 seconds after the signal crosses the level threshold, as shown in the following figure.



- **Polarity** — Select the polarity of the trigger type. The option you choose for **Type** directly affects the options available for **Polarity**, as shown in the following table.

Trigger Type	Polarity Options
Edge	Rising, Falling, Either
Pulse Width	Positive, Negative, Either
Transition	Rise Time, Fall Time, Either

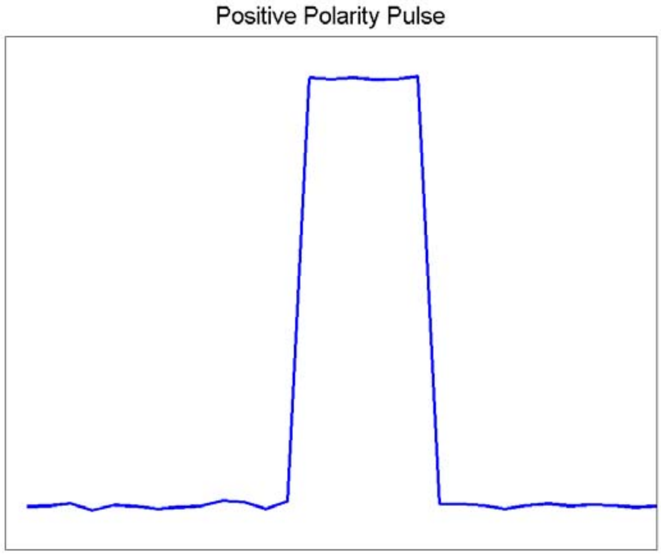
Trigger Type	Polarity Options
Runt	Positive, Negative, Either
Window	Inside, Outside, Either
Timeout	Rising, Falling, Either

When you set **Type** to `Edge`, the polarity options are:

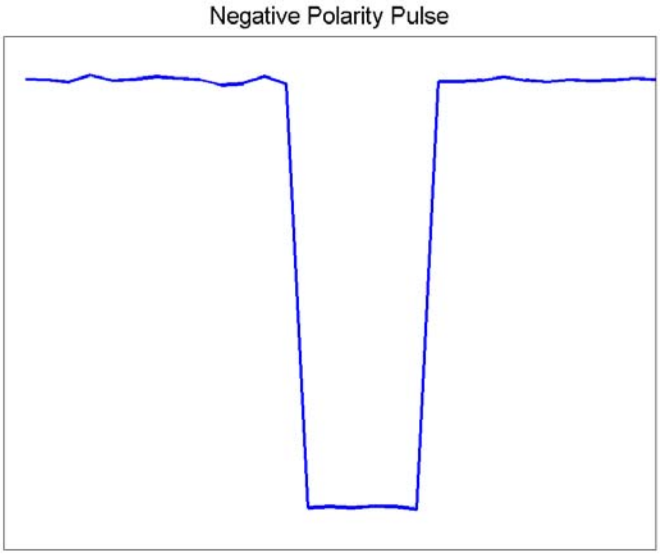
- **Rising** — Trigger on a *rising edge*, a transition from a low-state level to a high-state level.
- **Falling** — Trigger on a *falling edge*, transition from a high-state level to a low-state level.
- **Either** — Trigger on both rising edges and falling edges.

When you set **Type** to `Pulse Width` or `Runt`, the polarity options are:

- **Positive** — Trigger on a positive-polarity pulse, as shown in the following figure.



- **Negative** — Trigger on a negative-polarity pulse, as shown in the following figure.



- **Either** — Trigger on both positive-polarity and negative-polarity pulses.

When you set **Type** to `Transition`, the polarity options are:

- **Rise Time** — Trigger based on how long the signal takes to transition from the low threshold to the high threshold.
- **Fall Time** — Trigger based on how long the signal takes to transition from the high threshold to the low threshold.
- **Either** — Trigger based on how long it takes to make either a rising or falling transition.

When you set **Type** to `Window`, the polarity options are:

- **Inside** — Trigger when the signal stays within the low and high levels for a specified time duration.
- **Outside** — Trigger when the signal stays outside of the low and high levels for a specified time duration.

  - **Either** — Trigger on both inside and outside windows.

  When you set **Type** to `Timeout`, the polarity options are:

  - **Rising** — Trigger when the signal does not cross the reference level from below.
  - **Falling** — Trigger when the signal does not cross the reference level from above.
  - **Either** — Trigger when the signal does not cross the reference level from either direction.

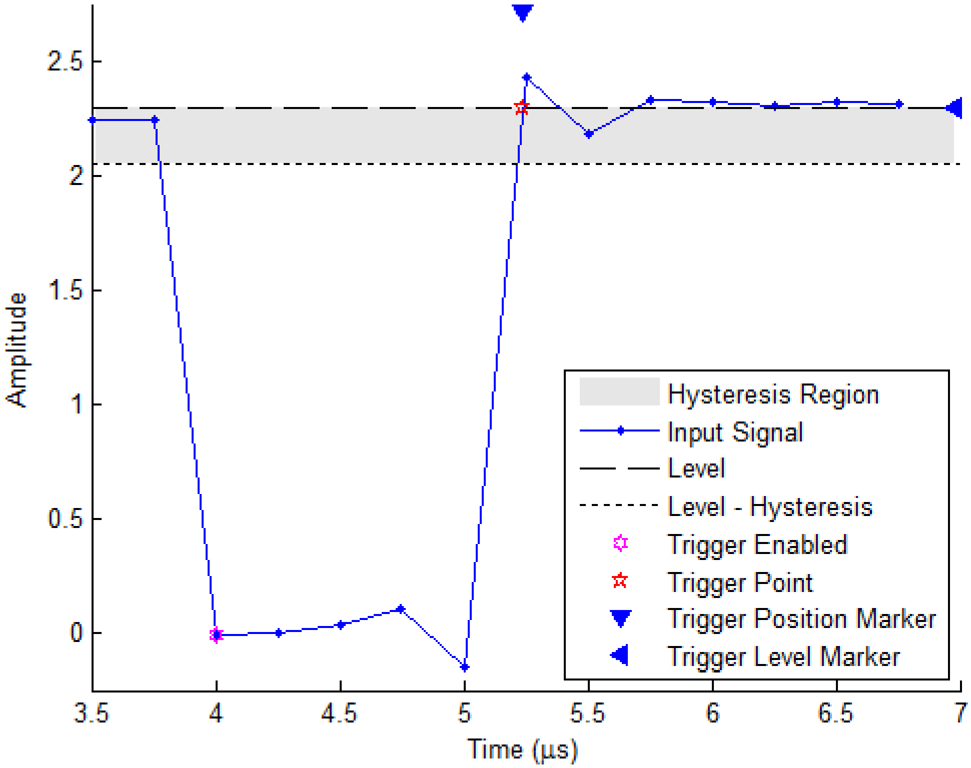
Levels / Timing Pane

The **Levels / Timing** pane enables you to set the trigger level and hysteresis value. The option you choose for **Type** directly affects which level and timing parameters are available, as shown in the following table.

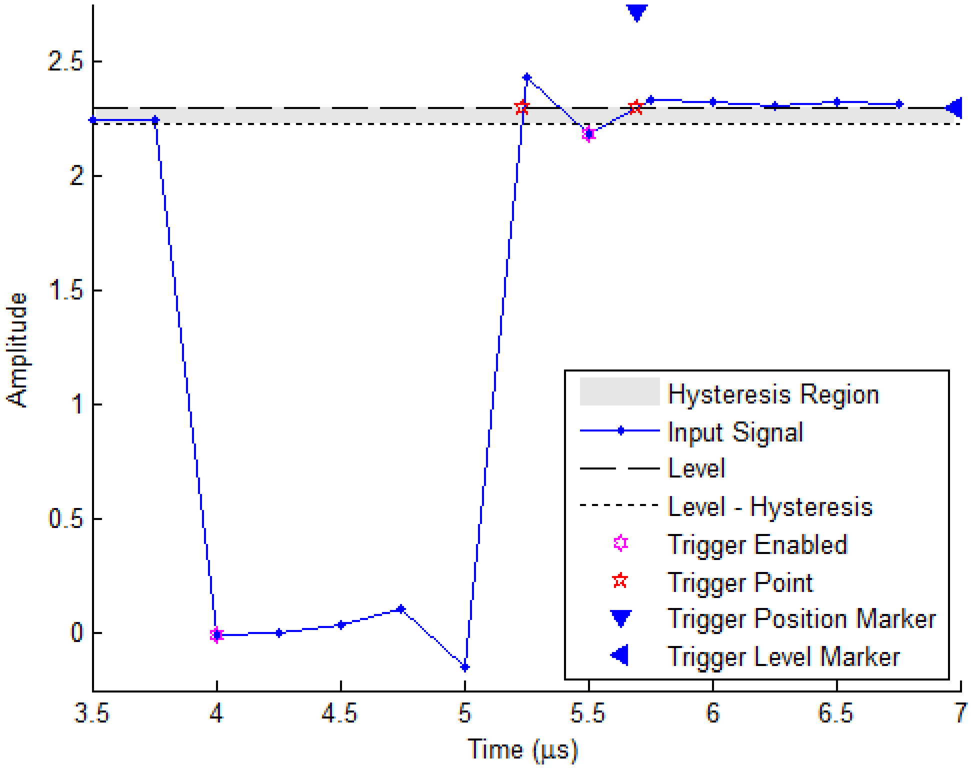
Trigger Type	Level Parameters	Auto-Level Setting	Timing Parameters
Edge	Level, Hysteresis	Level = 50%	n/a
Pulse Width	High, Low	High = 90%, Low = 10%	Min Width, Max Width
Transition	High, Low	High = 90%, Low = 10%	Min Time, Max Time
Runt	High, Low	High = 90%, Low = 10%	Min Width, Max Width
Window	High, Low	High = 90%, Low = 10%	Min Time, Max Time
Timeout	Level, Hysteresis	n/a	Timeout

- **Auto level** — Enable the Triggers panel to automatically choose the level parameters. If you set the trigger type to **Edge**, this option sets the **Level** parameter to 50% of the range of the source signal. If you set the trigger type to **Timeout**, the Triggers panel does not show this option. Setting the trigger type to other menu choices results in **High** and **Low** parameter adjustment. **Auto level** sets the **High** parameter to 90% of the range of the source signal and the **Low** parameter to 10% of the range of the source signal.

- **Level (V)** — Specify, in volts, the trigger level. This parameter is visible when you set **Type** to `Edge` or `Timeout`.
- **Hysteresis (V)** — Specify, in volts, the hysteresis or noise reject value. This parameter is visible when you set **Type** to `Edge` or `Timeout`. If the signal jitters inside this range and briefly crosses the trigger level, the scope does not register an event. In the case of an edge trigger with rising polarity, the scope ignores any times that the signal crosses the trigger level within the hysteresis region, as shown in the following figure.



You can reduce the hysteresis region size by decreasing the hysteresis value. If you set the hysteresis value to 0.07 in this example, then the scope also considers the second rising edge to be a trigger event, as shown in the following figure.



- **High (V)** — Specify, in volts, the value that denotes a positive polarity, or high-state level. This parameter is visible when you set **Type** to `Pulse Width`, `Transition`, `Runt`, or `Window`.
- **Low (V)** — Specify, in volts, the value that denotes a negative polarity, or low-state level. This parameter is visible when you set **Type** to `Pulse Width`, `Transition`, `Runt`, or `Window`.
- **Min Width (s)** — Specify, in seconds, the minimum pulse width. This parameter is visible when you set **Type** to `Pulse Width` or `Runt`.
- **Max Width (s)** — Specify, in seconds, the maximum pulse width. This parameter is visible when you set **Type** to `Pulse Width` or `Runt`.

- **Min Time (s)** — Specify, in seconds, the minimum duration. This parameter is visible when you set **Type** to `Transition` or `Window`.
- **Max Time (s)** — Specify, in seconds, the maximum duration. This parameter is visible when you set **Type** to `Transition` or `Window`.
- **Timeout (s)** — Specify, in seconds, the timeout duration. This parameter is visible when you set **Type** to `Timeout`.

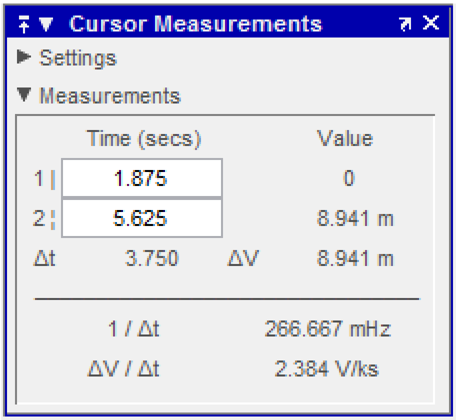
Delay / Holdoff Pane

The **Delay / Holdoff** pane enables you to offset the trigger position by a fixed delay or set the minimum possible time between trigger events.

- **Delay (s)** — Specify, in seconds, the fixed delay time by which to offset the trigger position. This parameter controls the amount of time the scope waits after a trigger event occurs before displaying a signal.
- **Holdoff (s)** — Specify, in seconds, the minimum possible time between trigger events. This amount of time is used to suppress data acquisition after a valid trigger event is encountered. A trigger holdoff prevents repeated occurrences of a trigger from occurring during the portion of a burst that is of interest.

Cursor Measurements Panel

The **Cursor Measurements** panel displays screen cursors. You can choose to hide or display the **Cursor Measurements** panel. In the Scope menu, select **Tools > Measurements > Cursor Measurements**. Alternatively, in the Scope toolbar, click the Cursor Measurements ⊞ button.



	Time (secs)		Value
1 |	1.875		0
2 ¦	5.625		8.941 m
Δt	3.750	ΔV	8.941 m

1 / Δt	266.667 mHz
ΔV / Δt	2.384 V/ks

Settings Pane

The **Settings** pane enables you to modify the type of screen cursors used to calculate time and value measurements.

- **Screen Cursors**— Shows screen cursors.
- **Horizontal**— Shows horizontal screen cursors.
- **Vertical**— Shows vertical screen cursors.
- **Waveform Cursors**— Shows cursors that attach to the input signals.
- **Lock Cursor Spacing**— Locks the time difference between the two cursors.

Measurements Pane

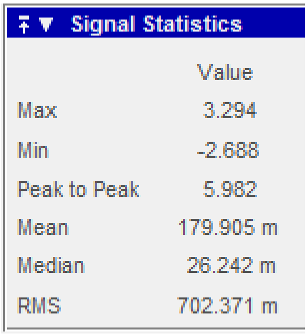
The **Measurements** pane shows the time and value measurements.

- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.
- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.

- **Δt**— Shows the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV**— Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.
- **1/Δt**— Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV/Δt**— Shows the scope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

Signal Statistics Panel

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the *x*-axis indices at which the maximum and minimum values occur. You can choose to hide or display the **Signal Statistics** panel. In the Scope menu, select **Tools > Measurements > Signal Statistics**. Alternatively, in the scope toolbar, click the Signal Statistics button.



Signal Statistics

	Value
Max	3.294
Min	-2.688
Peak to Peak	5.982
Mean	179.905 m
Median	26.242 m
RMS	702.371 m

Signal Statistics Measurements

The **Signal Statistics** panel shows statistics about the portion of the input signal within the *x*-axis and *y*-axis limits of the active display. The statistics shown are:

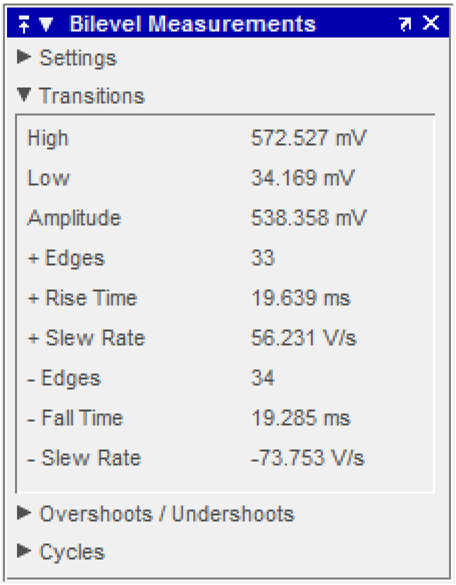
- **Max** — Shows the maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.
- **Min** — Shows the minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.
- **Peak to Peak** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.
- **Mean** — Shows the average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.
- **Median** — Shows the median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.
- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Bilevel Measurements Panel

The **Bilevel Measurements** panel shows information about a selected signal's transitions, overshoots or undershoots, and cycles. You can choose to hide or display the **Bilevel Measurements** panel. In the Scope menu, select **Tools > Measurements > Bilevel Measurements**. Alternatively, in the Scope toolbar, you can select the Bilevel Measurements button.



Bilevel Measurements

▶ Settings

▼ Transitions

High	572.527 mV
Low	34.169 mV
Amplitude	538.358 mV
+ Edges	33
+ Rise Time	19.639 ms
+ Slew Rate	56.231 V/s
- Edges	34
- Fall Time	19.285 ms
- Slew Rate	-73.753 V/s

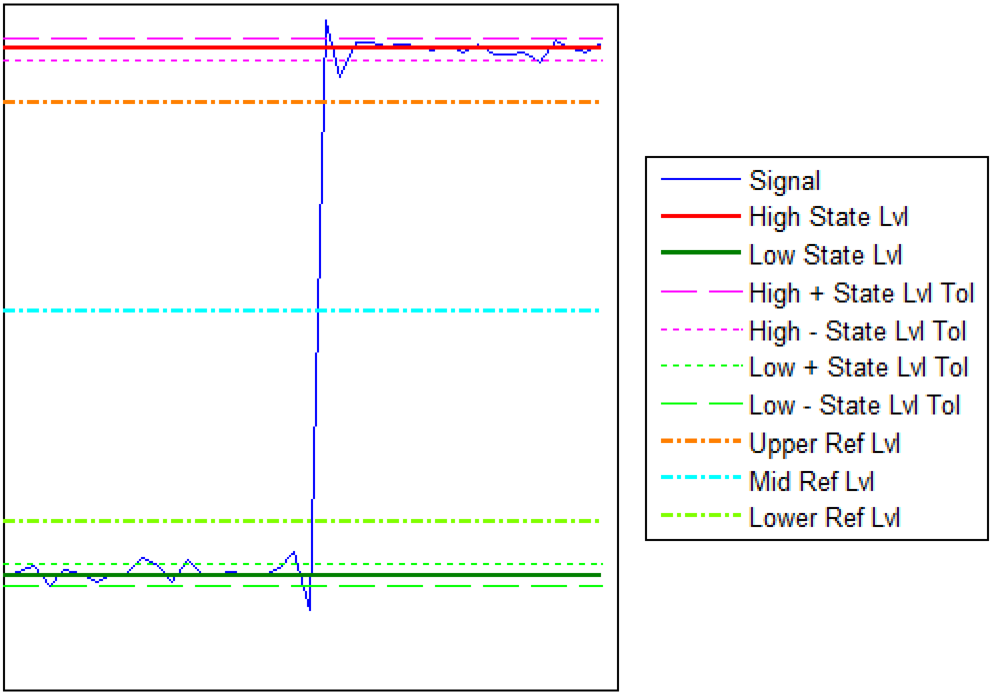
▶ Overshoots / Undershoots

▶ Cycles

The **Bilevel Measurements** panel is separated into four panes, labeled **Settings**, **Transitions**, **Overshoots / Undershoots**, and **Cycles**. You can expand each pane to see the available options.

Settings Pane

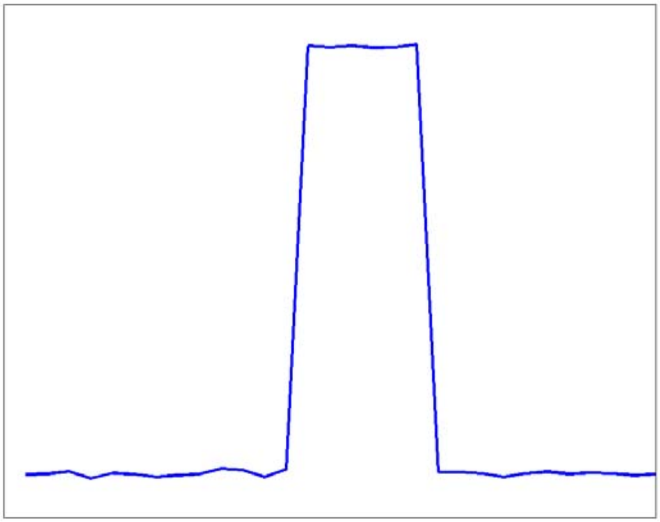
The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level, as shown in the following figure.



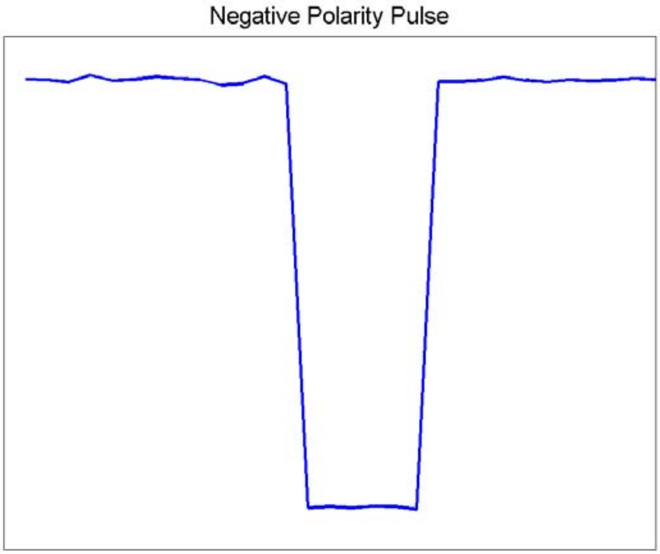
- **Auto State Level** — When this check box is selected, the Bilevel measurements panel autodetects the high- and low- state levels of a bilevel waveform. For more information on the algorithm this option uses, see the Signal Processing Toolbox `statelevels` function reference. When this check box is cleared, you may enter in values for the high- and low- state levels manually.

- **High** — Used to manually specify the value that denotes a positive polarity, or high-state level, as shown in the following figure.

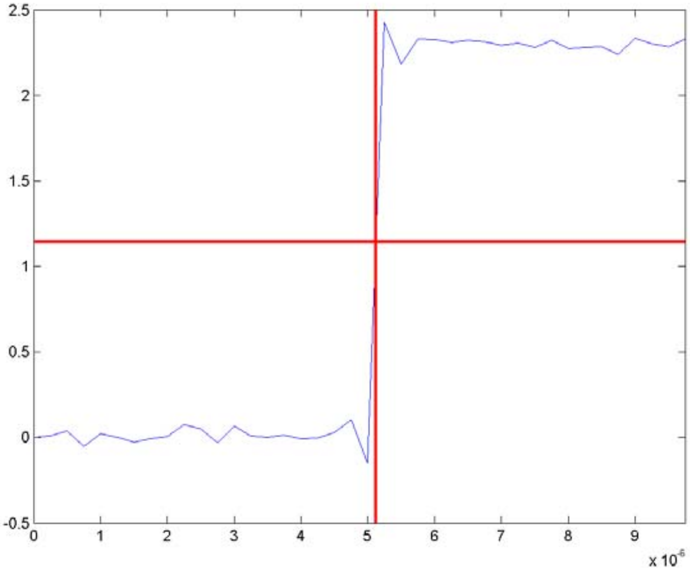
Positive Polarity Pulse



- **Low** — Used to manually specify the value that denotes a negative polarity, or low-state level, as shown in the following figure.



- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. In the following figure, the mid-reference level is shown as the horizontal line, and its corresponding mid-reference level instant is shown as the vertical line.

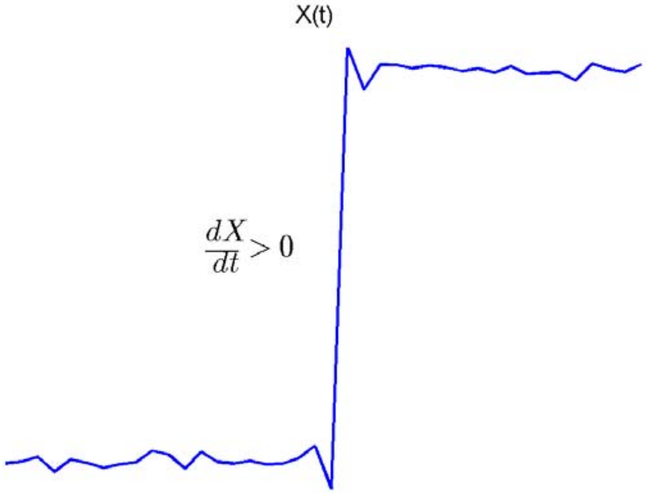


- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, `D`, which you can set when you run the `settlingtime` function. The settling time is displayed in the **Overshoots/Undershoots** pane.

Transitions Pane

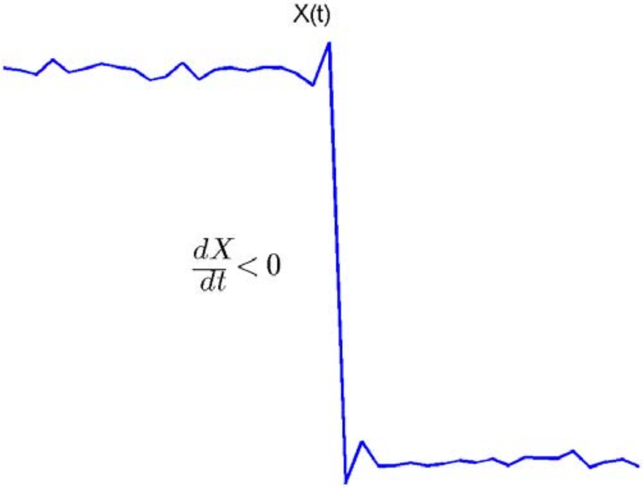
The **Transitions** pane displays calculated measurements associated with the input signal changing between its two possible state level values, high and low.

A positive-going transition, or *rising edge*, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. The following figure shows a positive-going transition.



Whenever there is a plus sign (+) next to a text label, this symbol refers to measurement associated with a rising edge, a transition from a low-state level to a high-state level.

A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. The following figure shows a negative-going transition.



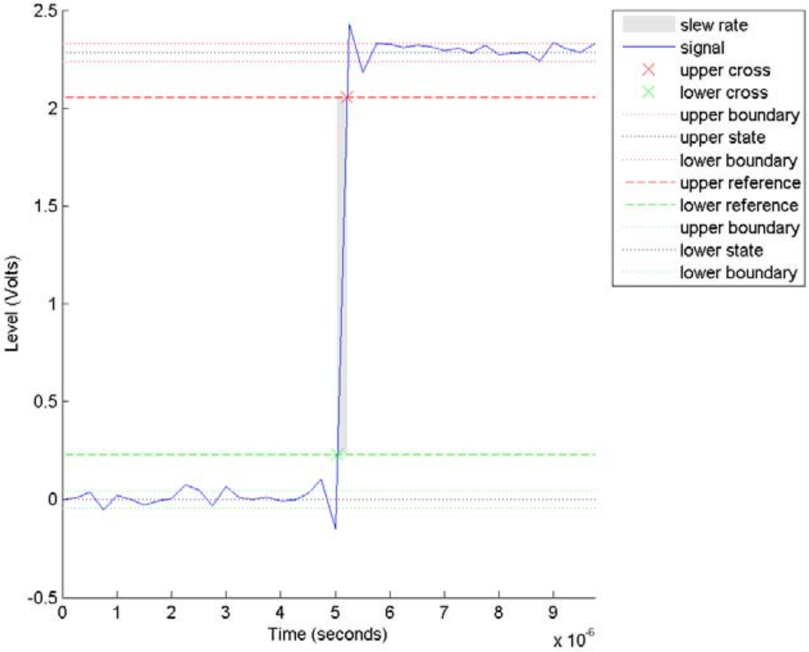
Whenever there is a minus sign (–) next to a text label, this symbol refers to measurement associated with a falling edge, a transition from a high-state level to a low-state level.

The Transition measurements assume that the amplitude of the input signal is in units of volts. You must convert all input signals to volts for the Transition measurements to be valid.

- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.
- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For

more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.

- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.
- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.
- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `risetime` function reference.
- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray in the following figure.



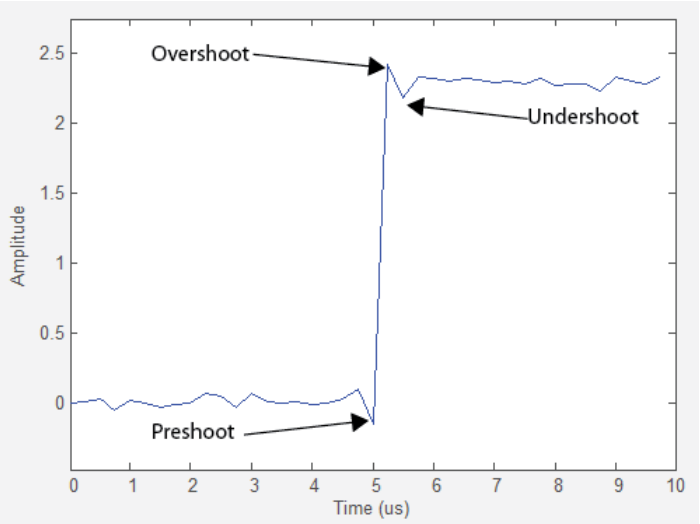
For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.
- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `falltime` function reference.
- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. For more information on the algorithm

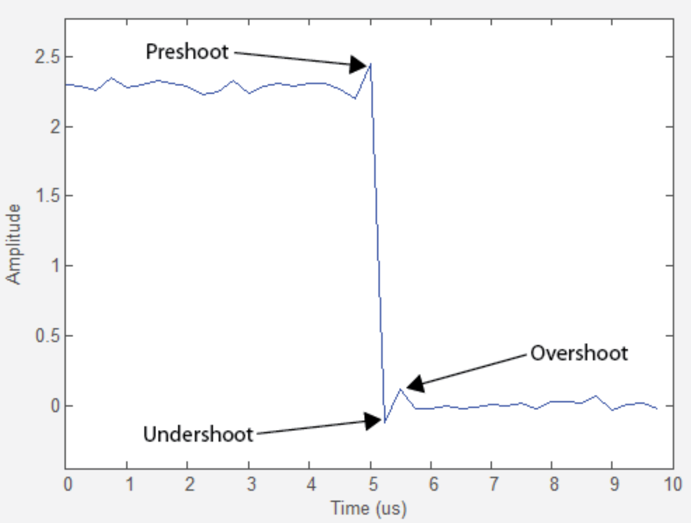
this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

Overshoots/Undershoots

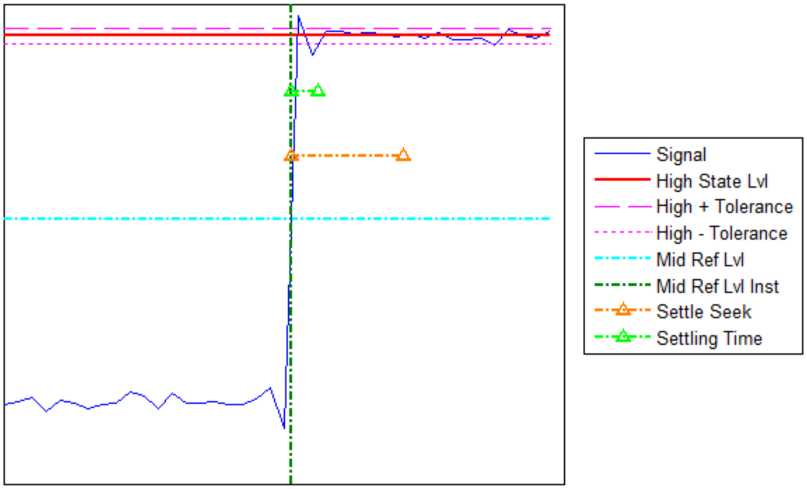
The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. *Overshoot* and *undershoot* refer to the amount that a signal respectively exceeds and falls below its final steady-state value. *Preshoot* refers to the amount prior to a transition that a signal varies from its initial steady-state value. This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.



The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.



- **+ Preshoot** — Average lowest aberration in the region immediately preceding each rising transition.
- **+ Overshoot** — Average highest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- **+ Undershoot** — Average lowest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.
- **+ Settling Time** — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.



You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

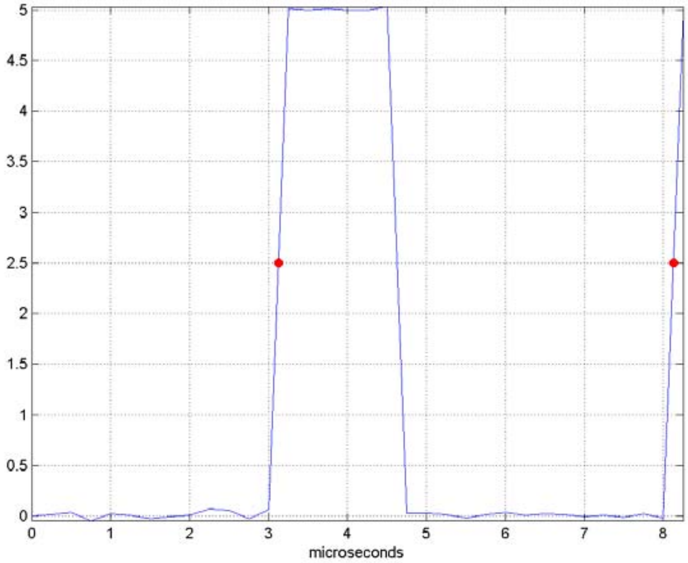
- – **Preshoot** — Average highest aberration in the region immediately preceding each falling transition.
- – **Overshoot** — Average highest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- – **Undershoot** — Average lowest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.
- – **Settling Time** — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can

modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

Cycles

The **Cycles** pane displays calculated measurements pertaining to repetitions or trends in the displayed portion of the input signal.

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots in the following figure.



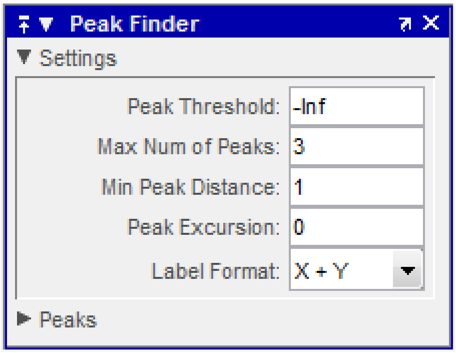
For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulseperiod` function reference.

- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.
- **+ Pulses** — Number of positive-polarity pulses counted.
- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.
- **– Pulses** — Number of negative-polarity pulses counted.
- **– Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- **– Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that particular rising edge. However, this feature does not apply to the **High** and **Low** measurements.

Peak Finder Panel

The **Peak Finder** panel displays maxima and the *x*-axis values at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified. You can choose to hide or display the **Peak Finder** panel. In the scope menu, select **Tools > Measurements > Peak Finder**. Alternatively, in the scope toolbar, select the Peak Finder button.



Peak Threshold: -Inf
Max Num of Peaks: 3
Min Peak Distance: 1
Peak Excursion: 0
Label Format: X + Y

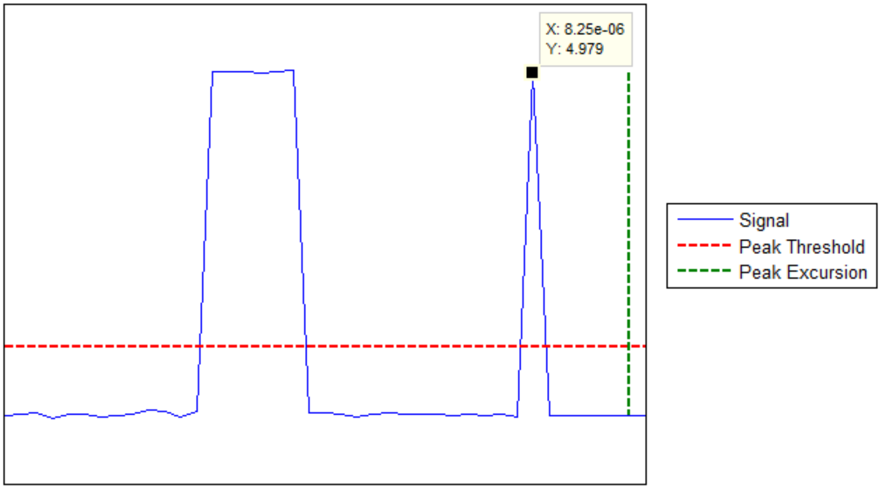
The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.

- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer between 1 and 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.



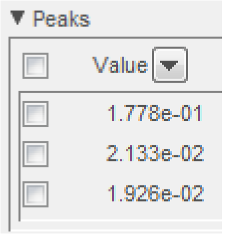
The *peak threshold* is a minimum value necessary for a sample value to be a peak. The *peak excursion* is the minimum difference between a peak sample and the samples to its left and right in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.

- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, you must first expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.
  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

Peaks Pane

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.



▼ Peaks

☐	Value
☐	1.778e-01
☐	2.133e-02
☐	1.926e-02

The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height. Use the sort descending button (▼) to rearrange the category and order by which Peak Finder displays peak values. Click this button again to sort the peaks in ascending order instead. When you do so, the arrow changes direction to become the sort ascending button (▲). A filled sort button indicates that the peak values are currently sorted by this category. If the sort button is not filled (▽), then the peak values are sorted by the other category. The **Max Num of Peaks** parameter still controls the number of peaks listed.

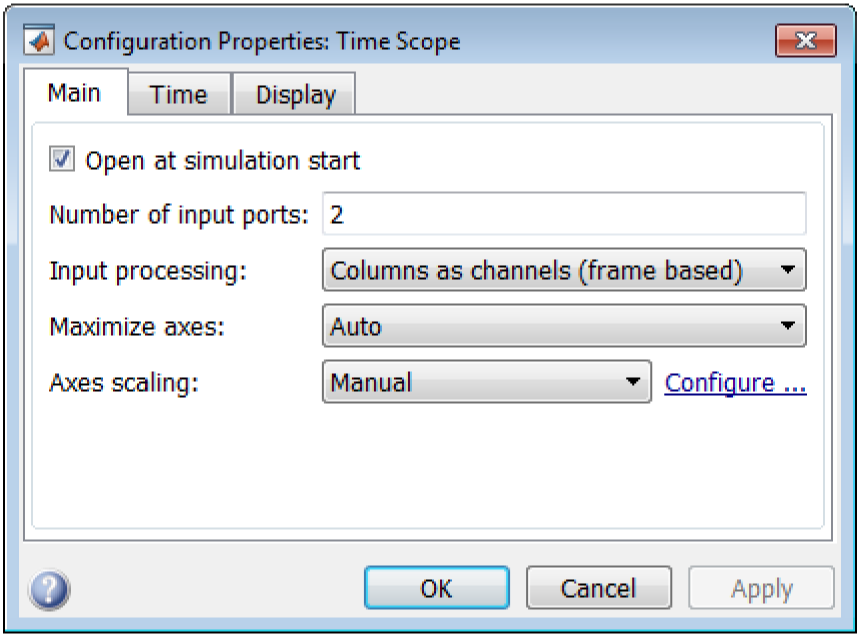
Use the check boxes to control which peaks are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peaks. To show all the peak values on the display, select the check box in the top-left corner of the **Peaks** pane. To hide all the peak values on the display, clear this check box. To show an individual peak, select the check box directly to the left of its **Value** listing. To hide an individual peak, clear the check box directly to the left of its **Value** listing.

Configuration Properties Dialog Box

The Configuration Properties dialog box controls various properties about the Time Scope displays. From the Time Scope menu, select **View > Configuration Properties** to open this dialog box. Alternatively, in the Time Scope toolbar, click the Configuration Properties ⚙ button.

Main Pane

The **Main** pane of the Configuration Properties dialog box appears as follows.



☑ Open at simulation start

Number of input ports: 2

Input processing: Columns as channels (frame based)

Maximize axes: Auto

Axes scaling: Manual

Open at simulation start

Select this check box to ensure that the scope opens when the simulation starts.

Number of input ports

Specify the number of input ports that should appear on the left side of the scope block.

Input processing

Specify whether the Time Scope should treat the input signal as `Columns as channels (frame based)` or `Elements as channels (sample based)`.

Frame-based processing is only available for discrete input signals. For more information about frame-based input channels, see the "What Is Frame-Based Processing?" section in the DSP System Toolbox documentation. For an example that uses the Time Scope block and frame-based input signals, see the Display Time-Domain Data in the Time Scope section in the DSP System Toolbox documentation.

Maximize axes

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable.

The default setting is `Auto`.

Axes scaling

Specify when the scope should automatically scale the axes. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools > Scale axes limits**.
  - Press the **Scale Axes Limits** toolbar button.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.

- `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. Selecting this option shows the **Number of updates** edit box.

This parameter is Tunable.

Note Click the link labeled **Configure** to the right of the **Axes scaling** parameter to see additional axes scaling parameters. After you click this button, its label changes to **Hide**. To hide these additional parameters, click the **Hide** link.

Scale axes limits at stop

Select this check box to scale the axes limits when the simulation stops. The *y*-axis are always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Do not allow Y-axis limits to shrink

When you select this parameter, the *y*-axis limits are only allowed to grow during axes scaling operations. If you clear this check box, the *y*-axis limits may shrink during axes scaling operations.

This parameter appears only when you select `Auto` for the **Axis Scaling** parameter. When you set the **Axes Scaling** parameter to `Manual` or `After N Updates`, the *y*-axis limits are allowed to shrink. Tunable.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this parameter to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable.

Y-axis Align

Specify where the scope should align your data with respect to the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable.

Scale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. Tunable.

X-axis Data range (%)

Set the percentage of the *x*-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to `100`, the Scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this parameter to `30`, the Scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** parameter to specify data placement with respect to the *x*-axis.

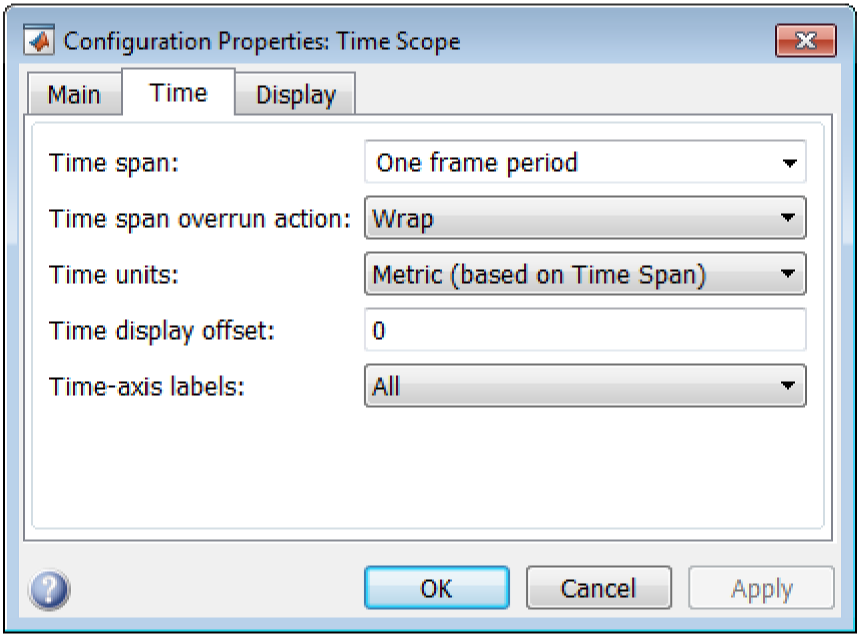
This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

X-axis Align

Specify how the Scope should align your data with respect to the *x*-axis: `Left`, `Center`, or `Right`. This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

Time Pane

The **Time** pane of the Configuration Properties dialog box appears as follows.



Time span: One frame period

Time span overrun action: Wrap

Time units: Metric (based on Time Span)

Time display offset: 0

Time-axis labels: All

Time span

Specify the time span, either by selecting a predefined option or by entering a numeric value in seconds. You can select one of the following options:

- `Auto` — In this mode, Time Scope automatically calculates the appropriate value for time span.

  The Time Scope block calculates the minimum and maximum *time*-axis limits as follows:

  - Minimum *time*-axis limit = Simulation “Start time”
  - Maximum *time*-axis limit = Simulation “Stop time” + max(*FrameRate* * (*FrameSize*–1) / *FrameSize*)

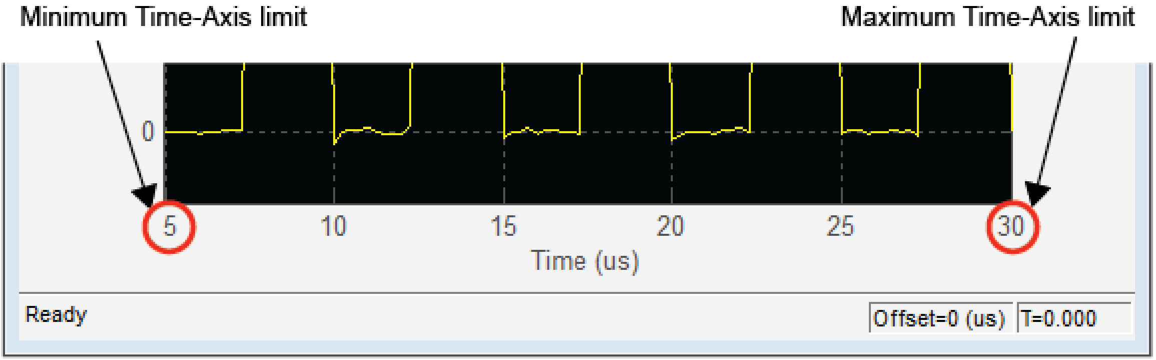
  FrameSize is a vector equal to the number of rows in each input signal. *FrameRate* is the reciprocal of the sample time for each frame. The Time Scope System object calculates the minimum and maximum *time*-axis limits as follows:

  - Minimum *time*-axis limit = min(`TimeDisplayOffset`)
  - Maximum *time*-axis limit = max(`TimeDisplayOffset`) + max(1/`SampleRate`.**FrameSize*)

  where `TimeDisplayOffset` and `SampleRate` are the values of their respective properties. This property is Tunable.

- `One frame period` — In this mode, the Time Scope uses the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to `Columns as channels (frame based)`. This option is not available when you set the **Input processing** parameter to `Elements as channels (sample based)`.
- `<user defined>` — In this mode, you specify the time span by replacing the text `<user defined>` with a numeric value in seconds.

The scope sets the *time*-axis limits using the value of this parameter and the value of the **Time display offset** parameter. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.



This parameter is Tunable.

Time span overrun action

Specify how the scope displays new data beyond the visible time span. You can select one of the following options:

- `Wrap` — In this mode, the scope displays new data until the data reaches the maximum *time*-axis limit. When the data reaches the maximum *time*-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum *time*-axis limit.
- `Scroll` — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

This parameter is Tunable.

The default setting is `Wrap`.

Time units

Specify the units used to describe the *time*-axis. You can select one of the following options:

- `Metric` — In this mode, the scope converts the times on the *time*-axis to the most appropriate measurement units. These can include milliseconds, microseconds, nanoseconds, minutes, days, etc. The scope chooses the appropriate measurement units based on the minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.
- `Seconds` — In this mode, the scope always displays the units on the *time*-axis as seconds.
- `None` — In this mode, the scope does not display any units on the *time*-axis. The scope only shows the word `Time` on the *time*-axis.

This property is Tunable.

The default setting is `Metric`.

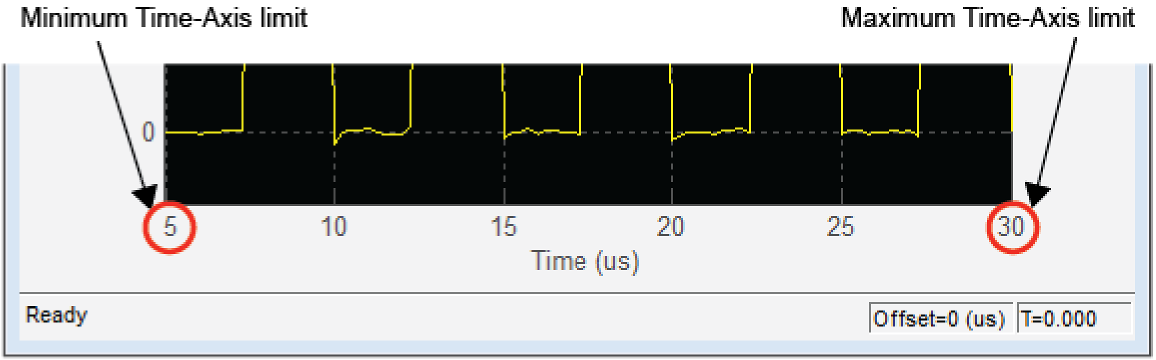
Time display offset

This parameter allows you to offset the values displayed on the *time*-axis by a specified number of seconds. When you specify a scalar value, the scope offsets all channels equally. When you specify a vector of offset values, the scope offsets each channel independently. Tunable.

When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.
- When *N* is less than the number of input channels, the scope applies the values you specify in the offset vector to the first *N* input channels. The scope does not offset the remaining channels.
- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

The scope computes the *time*-axis range using the values of the **Time display offset** and **Time span** parameters. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.



Similarly, when you specify a vector of values, the scope sets the minimum *time*-axis limit using the smallest value in the vector. To set the maximum *time*-axis limit, the scope sums the largest value in the vector with the value of the **Time span** parameter. For more information, see “Signal Display” on page 1-1768.

Show time-axis labels

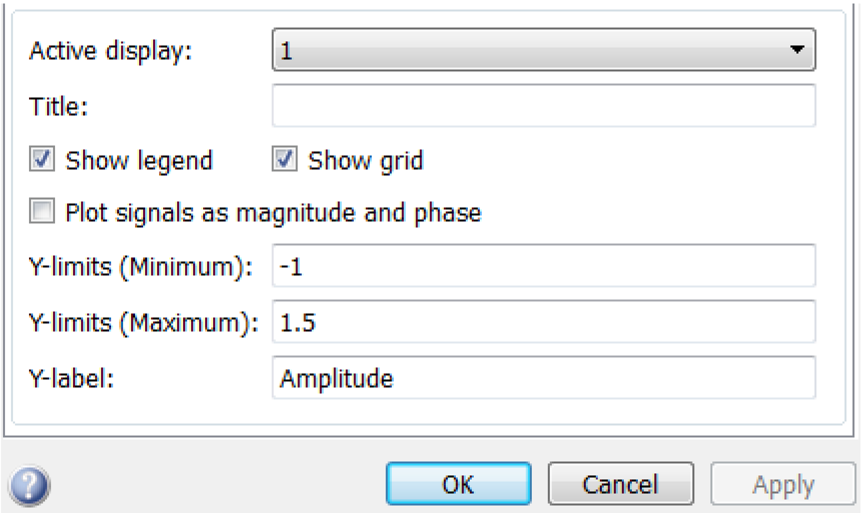
Specify how to display the time units used to describe the *time*-axis. The default setting is `All`. You can select one of the following options:

- `All` — In this mode, the *time*-axis labels appear in all displays.
- `None` — In this mode, the *time*-axis labels do not appear in the displays.
- `Bottom Displays Only` — In this mode, the *time*-axis labels appear in only the bottom row of the displays.

Tunable.

Display Pane

The **Display** pane of the Configuration Properties dialog box appears as follows.



Active display: 1

Title:

☑ Show legend ☑ Show grid

☐ Plot signals as magnitude and phase

Y-limits (Minimum): -1

Y-limits (Maximum): 1.5

Y-label: Amplitude

Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed. This property is Tunable.

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is `1`.

Title

Specify the active display title as a string. By default, the active display has no title. Tunable.

Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn the legend off, clear the **Show legend** check box. Tunable.

You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal.

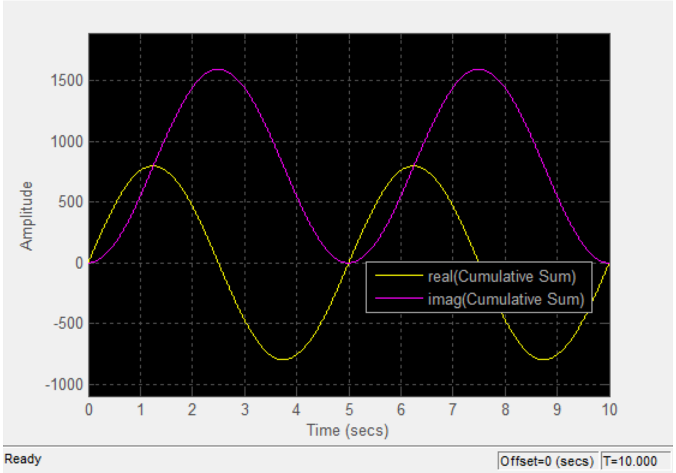
To change the appearance of any channel of any input signal in the scope window, from the menu, select **View > Style**.

Show grid

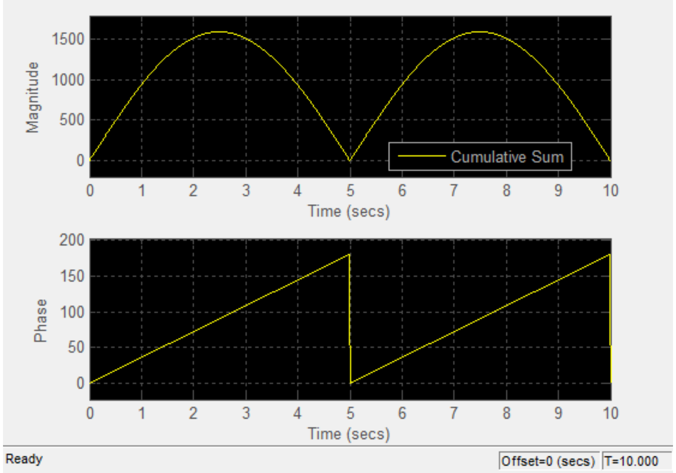
When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box. Tunable.

Plot signals as magnitude and phase

When you select this check box, the scope splits the display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal is complex valued, the scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes, as shown in the following figure.


1500
1000
500
0
-500
-1000
Amplitude
0
1
2
3
4
5
6
7
8
9
10
Time (secs)
real(Cumulative Sum)
imag(Cumulative Sum)
Ready
Offset=0 (secs)
T=10.000

Selecting this check box and clicking the **Apply** or **OK** button changes the display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.


Magnitude
1500
1000
500
0
Cumulative Sum
0 1 2 3 4 5 6 7 8 9 10
Time (secs)
Phase
200
150
100
50
0
0 1 2 3 4 5 6 7 8 9 10
Time (secs)
Ready
Offset=0 (secs)
T=10.000

This feature is particularly useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box always produces the same behavior. The phase is 0 for nonnegative input and 180 degrees for negative input. Tunable.

Y-limits (Minimum)

Specify the minimum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of –180 degrees.

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

Y-label

Specify as a string the text for the scope to display to the left of the *y*-axis. This property is Tunable.

This parameter becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that parameter, the *y*-axis label always appears as `Magnitude` on the top axes and `Phase` on the bottom axes.

Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You are able to change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. From the scope menu, select **View > Style** to open this dialog box.

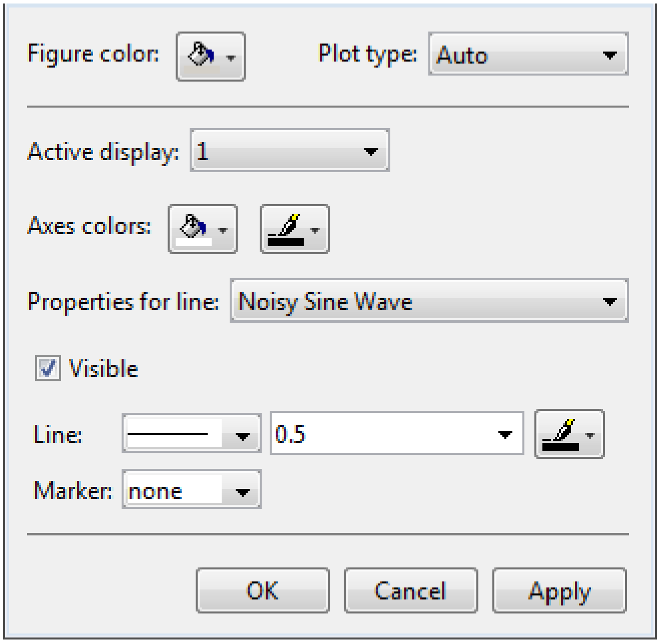


Figure color:

Plot type: Auto

Active display: 1

Axes colors:

Properties for line: Noisy Sine Wave

☑ Visible

Line: 0.5

Marker: none

Properties

The **Style** dialog box allows you to modify the following properties of the scope figure:

Figure color

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Plot type

Specify the type of plot to use. The default setting is `Line`. Valid values for **Plot type** are:

- `Line` — Displays input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB `line` or `plot` function.
- `Stairs` — Displays input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB `stairs` function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.
- `Auto` — Displays input signal as a line graph if it is a continuous signal and displays input signal as a stairstep graph if it is a discrete signal.

This parameter is Tunable.

Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed. This property is Tunable.

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is `1`.

Axes colors

Specify the color that you want to apply to the background of the axes for the active display.

Properties for line

Specify the signal for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected signal on the active display should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected signal on the active display.

Marker

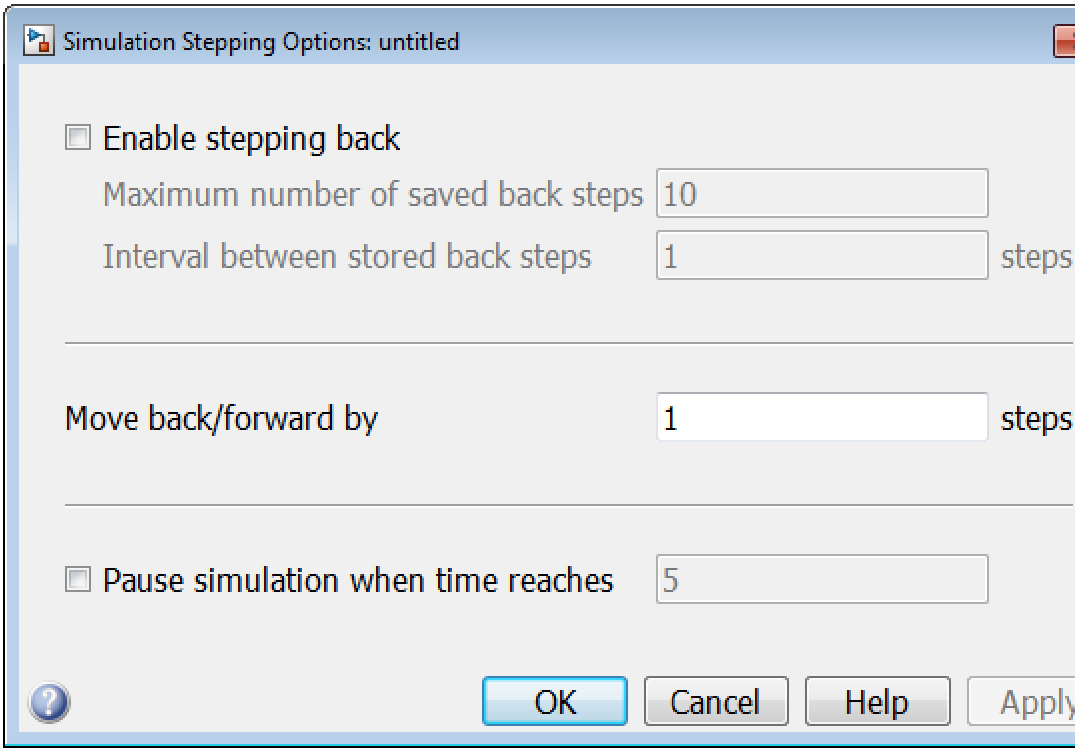
Specify marks for the selected signal on the active display to show at data points. This parameter is similar to the `Marker` property for the MATLAB Handle Graphics® plot objects. You can choose any of the marker symbols from the following table.

Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
•	Point
+	Plus sign

Specifier	Marker Type
*	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

Simulation Stepping Options

The Simulation Stepping Options dialog box lets you control the simulation behavior. You can pause the simulation at a specified time, enable stepping back, or specify options for stepping back. You can also modify the number of steps by which to step forward or backward. To open this dialog box, in the Time Scope menu, select **Simulation > Stepping Options** to open this dialog box. Alternatively, if stepping back is disabled, in the Time Scope toolbar, click the previous step ◀ button.


Simulation Stepping Options: untitled

☐ Enable stepping back

Maximum number of saved back steps: 10

Interval between stored back steps: 1 steps

Move back/forward by: 1 steps

☐ Pause simulation when time reaches: 5

The Simulation Stepping Options dialog box is not unique to Time Scope; it can also be launched from any Simulink model. To open this dialog box from any Simulink model, select **Simulation > Stepping Options**. For more information, see “Simulation Stepping” and “Simulation Stepping Options” in the Simulink documentation.

Enable stepping back

Select this check box to enable the Time Scope to take steps back in time. When selected, Time Scope enables the previous step button (◁‖) on the simulation toolbar.

Maximum number of saved back steps

Specify the maximum number of back steps that the Time Scope saves in memory. To maximize simulation speed, the value for this parameter should be kept small. The default setting is `10`.

Interval between stored back steps

Specify the number of steps between back steps that the Time Scope saves in memory for previous stepping. Set this parameter to a larger number to increase the time span of a back step without increasing the amount of memory used. The default setting is `1`.

Move back/forward by

Specify the number of steps forward or backward that the Time Scope progresses when you click the next step (‖▷) and previous step (◁‖) buttons. The default setting is `1`.

Pause simulation when time reaches

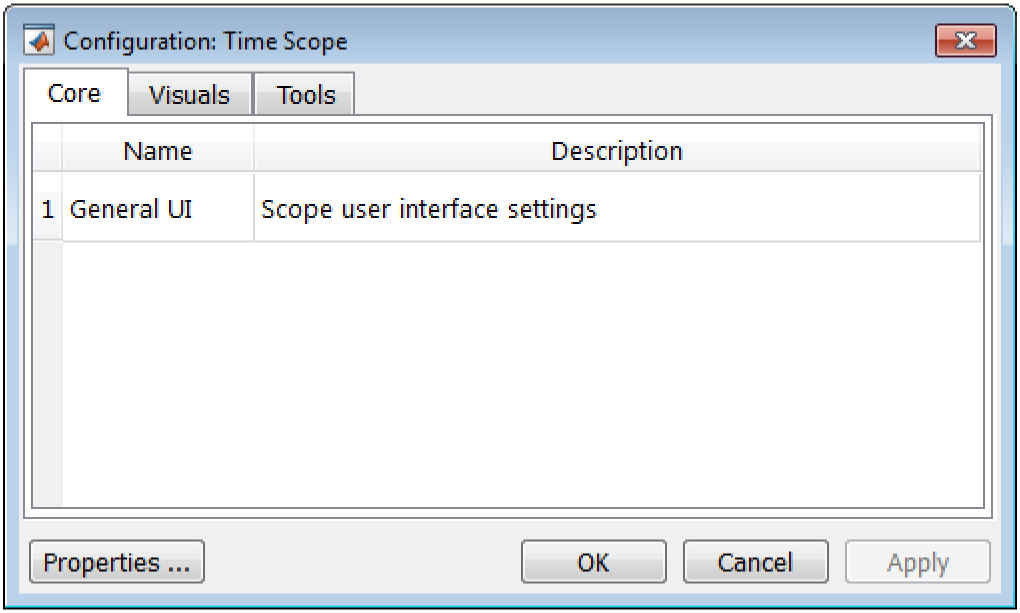
Select this check box to enable the Time Scope to pause the simulation when it reaches a specified time.

Pause simulation when time reaches

Specify the time at which you want Time Scope to pause when the check box is selected.

Core — General UI Options

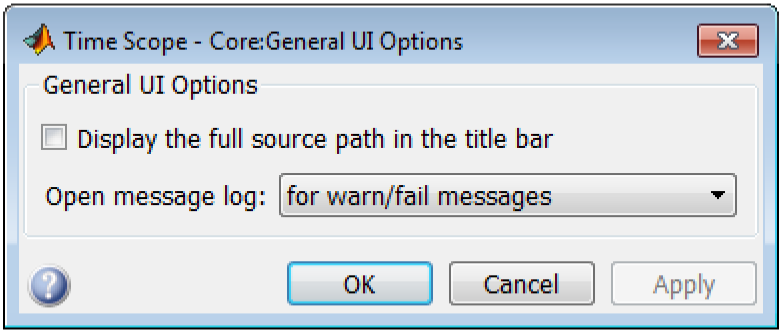
The Core—General UI Options dialog box controls the general settings for the scope figure. From the scope menu bar, select **File > Configuration** to open the Configuration dialog box. Click on the **Core** pane, and the following dialog box appears.


Configuration: Time Scope
Core
Visuals
Tools
Properties ...
OK
Cancel
Apply

	Name	Description
1	General UI	Scope user interface settings

To open the Core—General UI Options dialog box, select **General UI** and click **Properties**. The following dialog box appears.

General UI Options


Time Scope - Core:General UI Options
General UI Options
OK
Cancel
Apply

☐ Display the full source path in the title bar

Open message log: for warn/fail messages

Display the full source path in the title bar

If you select this check box, the scope figure displays the model name and full Simulink path to the data source in the title bar. Otherwise, a shortened name appears.

Open message log

Use this parameter to control when the Message Log window opens. The Message Log window helps you debug any issues with the Time Scope. Using the drop-down list, you can choose to open the Message Log window under any of the following conditions:

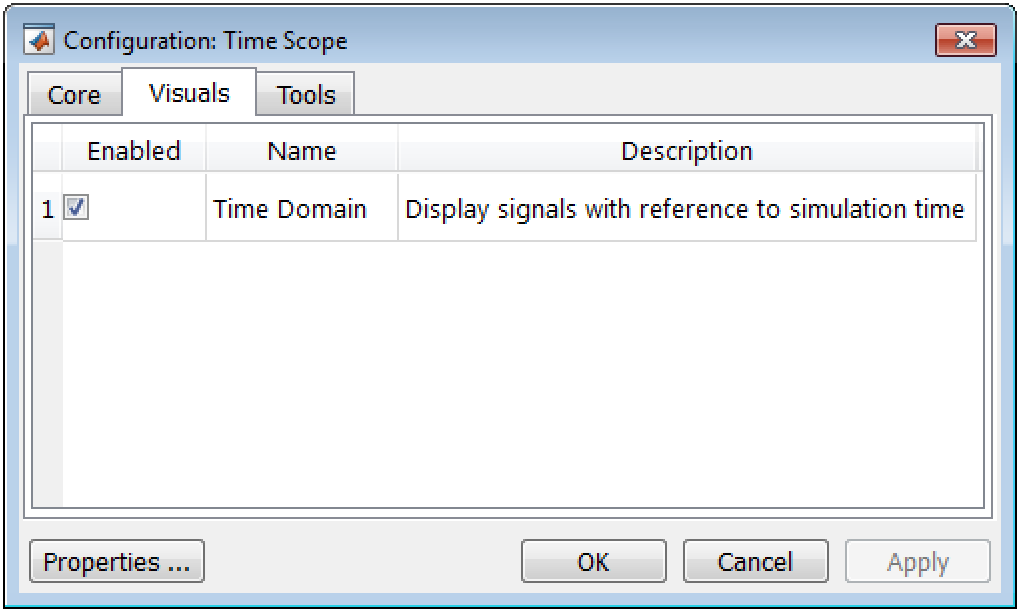
- `for any new messages`
- `for warn/fail messages`
- `only for fail messages`
- `manually`

You can open the Message Log at any time by selecting **Help > Message Log**. The Message Log dialog box provides a system-level record of loaded configuration settings and registered extensions. The Message Log displays summaries and details of each message, and you can filter the display of messages by **Type** and **Category**.

- The **Type** parameter allows you to select which types of messages to display in the Message Log. You can select `All`, `Info`, `Warn`, or `Fail`.
- The **Category** parameter allows you to select the category of messages to display in the Message Log. You can select `All`, `Configuration`, or `Extension`.
  - The scope uses `Configuration` messages to indicate when new configuration files are loaded.
  - The scope uses `Extension` messages to indicate when components are registered. For example, you may see a `Simulink registered` message, indicating that a component has been successfully registered with Simulink and is available for configuration.

Visuals — Time Domain Properties

The Visuals—Time Domain Properties dialog box controls the visual configuration settings of the Time Scope displays. You can open this dialog box from the Time Scope — Configuration dialog box. From the Time Scope menu bar, select **File > Configuration** to open the Configuration dialog box. Click the **Visuals** pane, and the following dialog box appears.

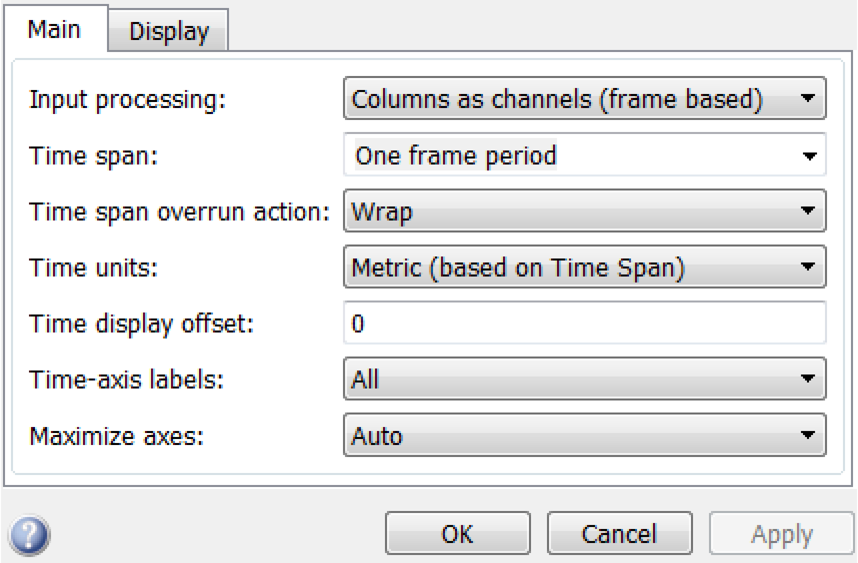


	Enabled	Name	Description
1	☑	Time Domain	Display signals with reference to simulation time

The Time Scope figure has only one available visual, so you must ensure that the **Enabled** check box is selected. To open the Visuals—Time Domain Properties dialog box, select **Time Domain** and click **Properties**.

Main Pane

The **Main** pane of the Time Scope Visuals—Time Domain Options dialog box appears as follows.



Input processing: Columns as channels (frame based)

Time span: One frame period

Time span overrun action: Wrap

Time units: Metric (based on Time Span)

Time display offset: 0

Time-axis labels: All

Maximize axes: Auto

Input processing

Specify whether the Time Scope should treat the input signal as `Columns as channels (frame based)` or `Elements as channels (sample based)`.

Frame-based processing is only available for discrete input signals. For more information about frame-based input channels, see the "What Is Frame-Based Processing?" section in the DSP System Toolbox documentation. For an example that uses the Time Scope block and frame-based input signals, see the Display Time-Domain Data in the Time Scope section in the DSP System Toolbox documentation.

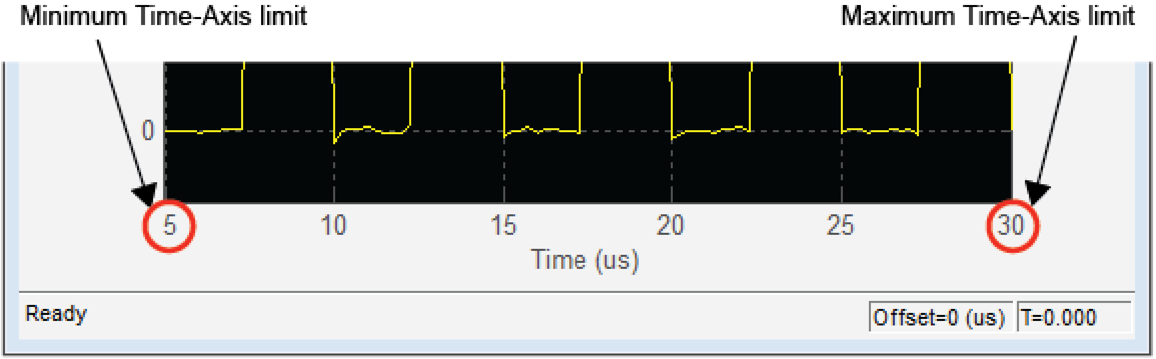
Time span

Specify the time span, either by selecting a predefined option or by entering a numeric value in seconds. You can select one of the following options:

- `Auto` — In this mode, the Time Scope automatically calculates the appropriate value for time span from the difference between the simulation "Start time" and "Stop time" parameters. This option is available only for the Time Scope block but not for the `dsp.TimeScope` System object™.

- `One frame period` — In this mode, the Time Scope uses the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to `Columns as channels (frame based)`. This option is not available when you set the **Input processing** parameter to `Elements as channels (sample based)`.
- `<user defined>` — In this mode, you specify the time span by replacing the text `<user defined>` with a numeric value in seconds.

The scope sets the *time*-axis limits using the value of this parameter and the value of the **Time display offset** parameter. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.



This parameter is Tunable.

Time span overrun action

Specify how the scope displays new data beyond the visible time span. You can select one of the following options:

- `Wrap` — In this mode, the scope displays new data until the data reaches the maximum *time*-axis limit. When the data reaches the maximum *time*-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum *time*-axis limit.

- `Scroll` — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

This parameter is Tunable.

The default setting is `Wrap`.

Time units

Specify the units used to describe the *time*-axis. The default setting is `Metric`. You can select one of the following options:

- `Metric` — In this mode, Time Scope converts the times on the *time*-axis to some metric units such as milliseconds, microseconds, days, etc. Time Scope chooses the appropriate metric units, based on the minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.
- `Seconds` — In this mode, Time Scope always displays the units on the *time*-axis as seconds.
- `None` — In this mode, Time Scope displays no units on the *time*-axis. Time Scope shows only the word `Time` on the *time*-axis.

This parameter is Tunable.

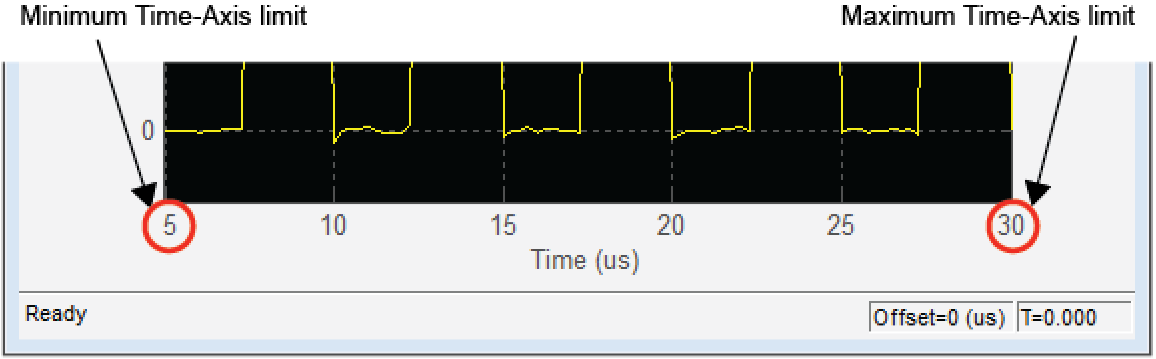
Time display offset

This parameter allows you to offset the values displayed on the *time*-axis by a specified number of seconds. When you specify a scalar value, the scope offsets all channels equally. When you specify a vector of offset values, the scope offsets each channel independently. Tunable.

When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When N is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.
- When N is less than the number of input channels, the scope applies the values you specify in the offset vector to the first N input channels. The scope does not offset the remaining channels.
- When N is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

The scope computes the *time*-axis range using the values of the **Time display offset** and **Time span** parameters. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.



Similarly, when you specify a vector of values, the scope sets the minimum *time*-axis limit using the smallest value in the vector. To set the maximum *time*-axis limit, the scope sums the largest value in the vector with the value of the **Time span** parameter. For more information, see “Signal Display” on page 1-1768.

Show time-axis labels

Specify how to display the time units used to describe the *time*-axis. The default setting is `All`. You can select one of the following options:

- `All` — In this mode, the *time*-axis labels appear in all displays.
- `None` — In this mode, the *time*-axis labels do not appear in the displays.
- `Bottom Displays Only` — In this mode, the *time*-axis labels appear in only the bottom row of the displays.

Tunable.

Maximize axes

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

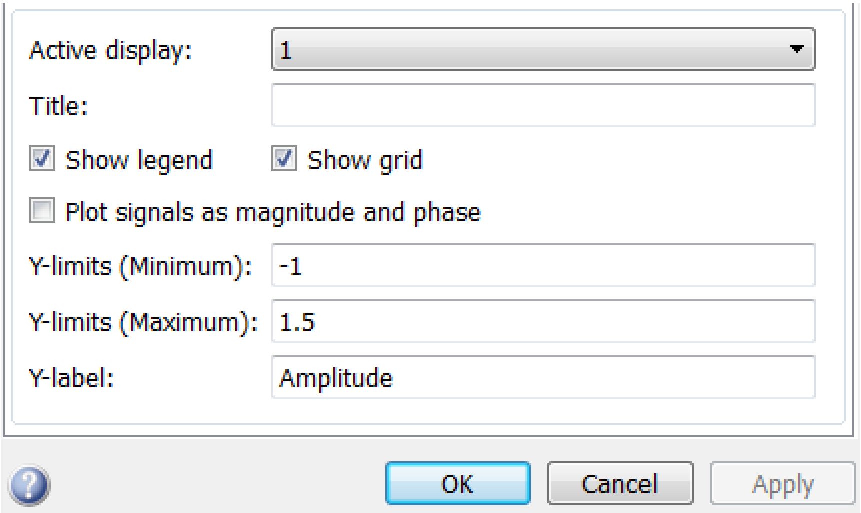
- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable.

The default setting is `Auto`.

Display Pane

The **Display** pane of the Visuals—Time Domain Options dialog box appears as follows.



Active display: 1

Title:

☑ Show legend ☑ Show grid

☐ Plot signals as magnitude and phase

Y-limits (Minimum): -1

Y-limits (Maximum): 1.5

Y-label: Amplitude

Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed. This property is Tunable.

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is 1.

Title

Specify the active display title as a string. By default, the active display has no title. Tunable.

Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn the legend off, clear the **Show legend** check box. Tunable.

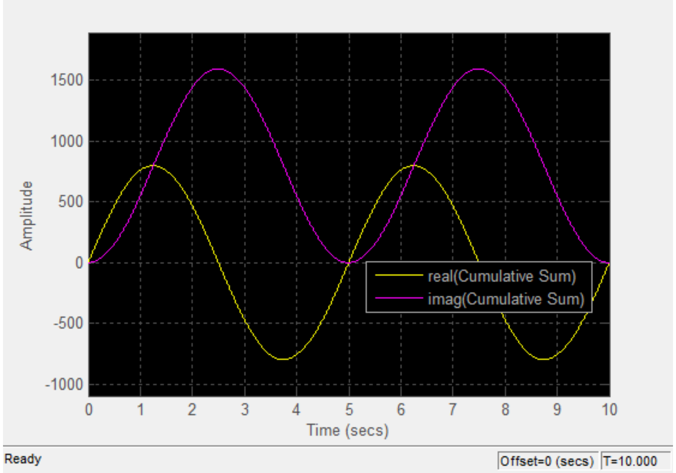
You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, if the signal has multiple channels, the scope uses an index number to identify each channel of that signal. To change the appearance of any channel of any input signal in the scope window, from the scope menu, select **View > Style**.

Show grid

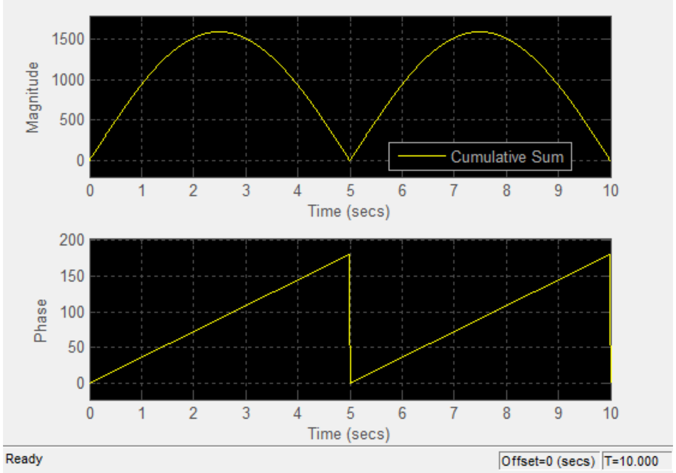
When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box. Tunable.

Plot signals as magnitude and phase

When you select this check box, the scope splits the display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal is complex valued, the scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes, as shown in the following figure.


1500
1000
500
0
-500
-1000
Amplitude
0
1
2
3
4
5
6
7
8
9
10
Time (secs)
real(Cumulative Sum)
imag(Cumulative Sum)
Ready
Offset=0 (secs)
T=10.000

Selecting this check box and clicking the **Apply** or **OK** button changes the display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.


Magnitude
1500
1000
500
0
Cumulative Sum
0
1
2
3
4
5
6
7
8
9
10
Time (secs)
Phase
200
150
100
50
0
0
1
2
3
4
5
6
7
8
9
10
Time (secs)
Ready
Offset=0 (secs)
T=10.000

This feature is particularly useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box always produces the same behavior. The phase is 0 for nonnegative input and 180 degrees for negative input. Tunable.

Y-limits (Minimum)

Specify the minimum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of –180 degrees.

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

Y-label

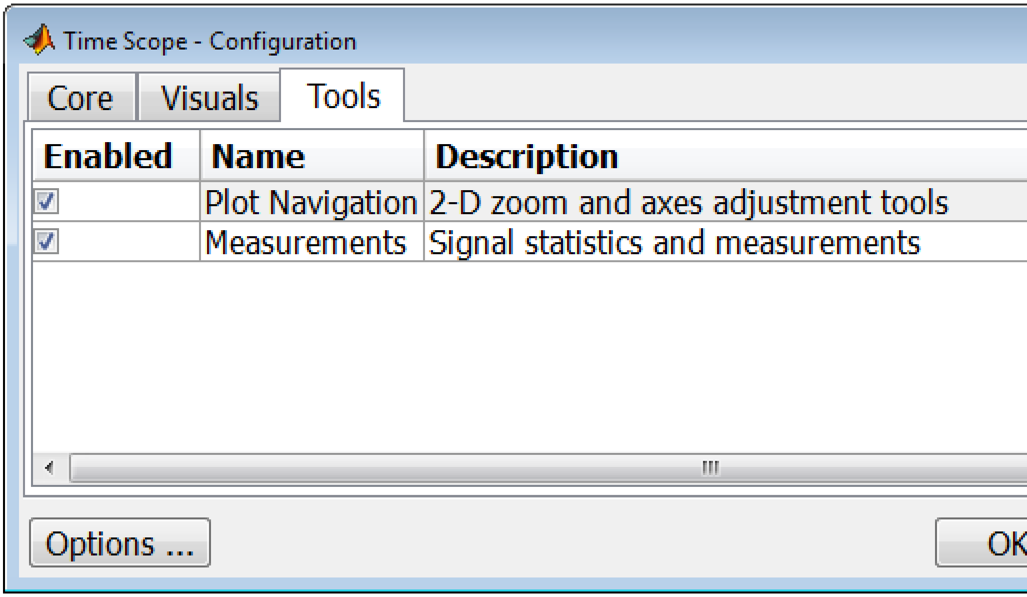
Specify as a string the text for the scope to display to the left of the *y*-axis. This property is Tunable.

This parameter becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that parameter, the *y*-axis label always appears as `Magnitude` on the top axes and `Phase` on the bottom axes.

Tools — Plot Navigation Options

The Tools:Plot Navigation Options dialog box provides you with the ability to automatically zoom in on and zoom out of your data, and to scale the axes of the Time Scope. In the Time Scope menu, select **Tools > Axes Scaling Options** to open this dialog box.

Alternatively, you can open this dialog box from the Time Scope — Configuration dialog box. In the Time Scope menu, select **File > Configuration** to open the Time Scope — Configuration dialog box. Click the **Tools** pane, and the following dialog box appears.



Enabled	Name	Description
☑	Plot Navigation	2-D zoom and axes adjustment tools
☑	Measurements	Signal statistics and measurements

The **Tools** pane in the Time Scope – Configuration dialog box contains the tools that appear on the Time Scope figure. By default, the **Plot Navigation** tool is enabled. If you disable the tool by clearing the

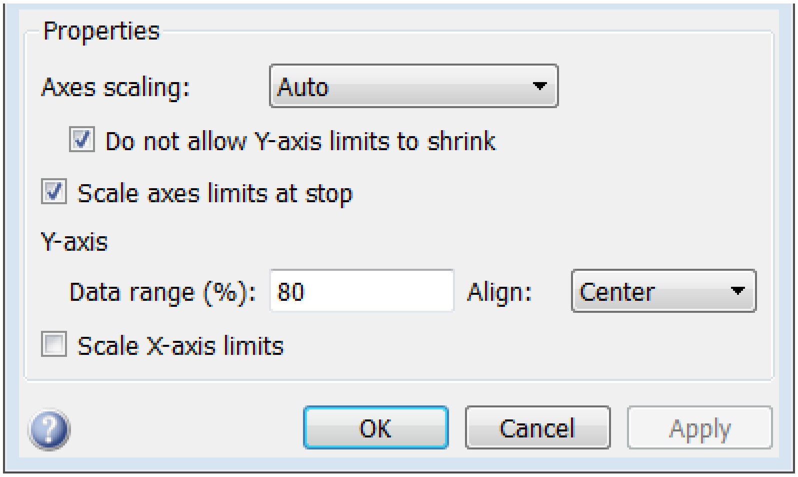
Enabled check box, the axes scaling and zoom tools no longer display in the Time Scope figure. You can reenable the tool at any time by selecting the **Enabled** check box.

By default, the **Measurements** tool is also enabled. If you disable the tool by clearing the **Enabled** check box, the Time Domain Measurements tools no longer display in the Time Scope figure. See the “Measurements Panels” on page 1-1779 section for more information. You can reenable the tool at any time by selecting the **Enabled** check box.

To open the Time Scope — Tools:Plot Navigation Options dialog box, select the **Plot Navigation** tool, and then click the **Options** button.

Properties

The Tools—Plot Navigation Properties dialog box appears as follows.



Properties

Axes scaling: Auto

☑ Do not allow Y-axis limits to shrink

☑ Scale axes limits at stop

Y-axis

Data range (%): 80

Align: Center

☐ Scale X-axis limits

Axes Scaling

Specify when the scope should automatically scale the axes. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools > Scale axes limits**.
  - Press the **Scale Axes Limits** toolbar button.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
- `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. Selecting this option shows the **Number of updates** edit box.

This parameter is Tunable.

By default, this parameter is set to `Manual`.

Do not allow Y-axis limits to shrink

When you select this parameter, the *y*-axis limits are only allowed to grow during axes scaling operations. If you clear this check box, the *y*-axis limits may shrink during axes scaling operations.

This parameter appears only when you select `Auto` for the **Axis Scaling** parameter. When you set the **Axes Scaling** parameter to `Manual` or `After N Updates`, the *y*-axis limits are allowed to shrink. Tunable.

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. This parameter appears only when you select `After N Updates` for the **Axes Scaling** parameter. Tunable.

Scale axes limits at stop

Select this check box to scale the axes limits when the simulation stops. The *y*-axis are always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this parameter to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable.

Y-axis Align

Specify where the scope should align your data with respect to the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable.

Scale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. Tunable.

X-axis Data range (%)

Set the percentage of the *x*-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to `100`, the Scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this parameter to `30`, the Scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** parameter to specify data placement with respect to the *x*-axis.

This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

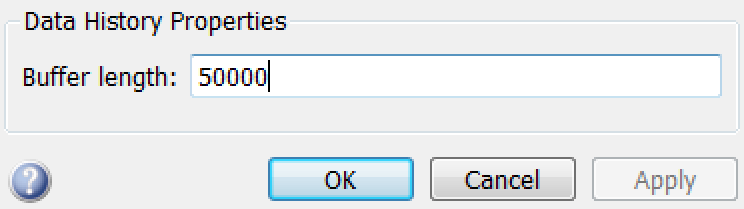
X-axis Align

Specify how the Scope should align your data with respect to the *x*-axis: `Left`, `Center`, or `Right`. This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

Sources — Simulink Properties

The Sources:Simulink Properties dialog box lets you control the number of samples for each input signal that Time Scope holds in its memory cache. In the Time Scope menu, select **View > Data History Properties** to open this dialog box.

Data History Properties



Buffer length: 50000

Buffer length

Specify the number of data points for each input signal that the scope holds in its memory cache. The number of data points equals the product of the number of channels in the signal and the number of time samples.

The default setting is `50000`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Simulink enumerations

Supported Simulation Modes

You can use the scope block in models running the following supported simulation modes.

Mode	Supported	Notes and Limitations
Normal	Yes	
Accelerator	Yes	
Rapid Accelerator	Yes	You can use Rapid Accelerator mode as a method to increase the execution speed of your Simulink model. Rapid Accelerator mode creates an executable that includes the solver and model methods. This executable resides outside MATLAB and Simulink. Rapid Accelerator mode uses External mode to communicate with Simulink. For more information about Rapid Accelerator mode, see "Acceleration" in the Simulink documentation.
PIL	No	
SIL	No	
External	Yes	You can use External mode to tune block parameters in real time and view block outputs in many types of blocks and subsystems. External mode establishes communication between a host system, where the Simulink environment resides, and a target system, where the executable runs after it is generated by the code generation and build process. For more information about External mode, see "Host/Target Communication" in the Simulink Coder documentation. Time Scope does not support data archiving. See "Set External Mode Data Archiving

Mode	Supported	Notes and Limitations
		Parameters” in the Real-Time Windows Target™ documentation.

For more information about these modes, see “How Acceleration Modes Work” in the Simulink documentation.

See Also

Scope

How To

- “View Simulation Results”

To File

Purpose

Write data to file

Library

Sinks

Description

The To File block inputs a signal and writes the signal data to a MAT-file. Use the To File block to log signal data.

untitled.mat

The To File block icon shows the name of the output file.

The block writes to the output file incrementally, with minimal memory overhead during simulation. If the output file exists when the simulation starts, the block overwrites the file. The file automatically closes when simulation is complete. If simulation terminates abnormally, the To File block saves the data it has logged up until the point of the abnormal termination.

Specifying the Format for Writing Data

Use the **Save format** parameter to specify which of the following two formats the To File block uses for writing data:

- `Timeseries` (default)
- `Array`

Use the `Array` format only for vector, double, noncomplex inputs. To save bus data, use the `Timeseries` format.

For the `Timeseries` format, the To File block:

- Writes data in a MATLAB `timeseries` object
- Supports writing multidimensional, real, or complex output values
- Supports writing output values that have any built-in data type, including `Boolean`, enumerated (`enum`), and fixed-point data with a word length of up to 32 bits
- For bus input signals, creates a MATLAB structure that matches the bus hierarchy. Each leaf of the structure is a MATLAB `timeseries` object.

For the `Array` format, the To File block:

- Writes data into a matrix containing two or more rows. The matrix has the following form:

  $$\begin{bmatrix} t_1 & t_2 & \dots & t_{final} \\ u1_1 & u1_2 & \dots & u1_{final} \\ \dots & & & \\ un_1 & un_2 & \dots & un_{final} \end{bmatrix}$$

  Simulink writes one column to the matrix for each data sample. The first element of the column contains the time stamp. The remainder of the column contains data for the corresponding output values.

- Supports writing data that is one-dimensional, double, and noncomplex.

Controlling When Data Is Written to the File

The To File block **Decimation** and **Sample Time** parameters control when data is written to the file.

Saving Data for Use by a From File Block

The From File block can use data written by a To File block in any format (`Timeseries` or `Array`) without any modifications to the data or other special provisions.

Saving Data for Use by a From Workspace Block

The From Workspace block can read data that is in the `Array` format and is the transposition of the data written by the To File block. To provide the required format, use MATLAB commands to load and transpose the data from a MAT-file.

Simulation Stepper Interaction with To File Block

If you pause using the Simulation Stepper, the To File block captures the simulation data up to the point of the pause. When you step back, the To File data file no longer contains any simulation data past the new reduced time of the last output.

Limitation on To File blocks in a Referenced Model

When a To File block is in a referenced model, that model must be a single-instance model. Only one instance of such a model can exist in a model hierarchy. See "General Reusability Limitations" for more information.

Compressing MAT-File Data

To avoid the overhead of compressing data in real time, the To File block writes an uncompressed Version 7.3 MAT-file. To compress the data within the MAT-file, load and save the file in MATLAB. The resaved file is smaller than the original MAT-file that the To File block created, because the **Save** command compresses the data in the MAT-file.

Saving Bus Data

The To File block supports virtual and nonvirtual bus input.

To save bus data, set the **Save format** parameter to `Timeseries`.

If the input signal is a bus, then the To File block creates a MATLAB structure that matches the bus hierarchy. Each leaf of the structure is a MATLAB `timeseries` object.

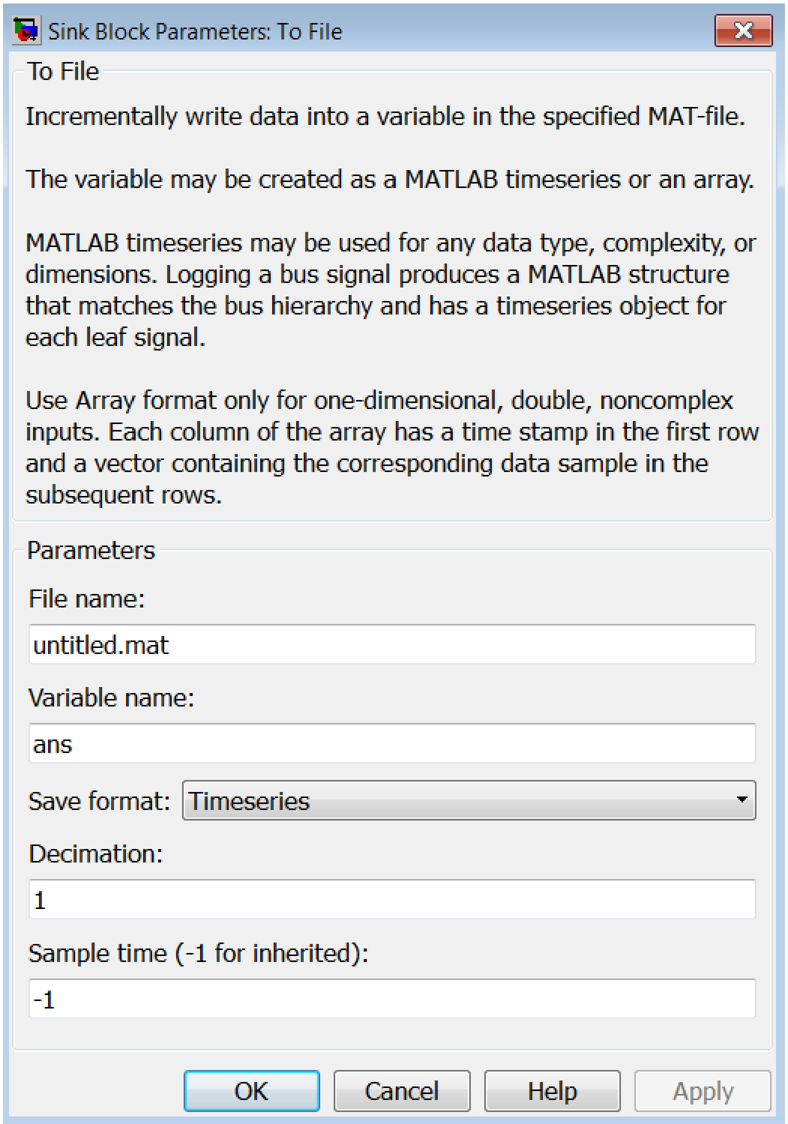
Pausing a Simulation

After pausing a simulation, do not alter any file that a To File block logs into. For example, do not save such a file with the MATLAB `save` command. Altering the file can cause an error when you resume the simulation. If you want to alter the file after pausing, copy the file and work with the copied version of the file.

Data Type Support

The To File block accepts real or complex signal data of any data type that Simulink supports, with the exception that the word length for fixed-point data must be 32 bits or less.

The To File block accepts bus data.



To File

Incrementally write data into a variable in the specified MAT-file.

The variable may be created as a MATLAB timeseries or an array.

MATLAB timeseries may be used for any data type, complexity, or dimensions. Logging a bus signal produces a MATLAB structure that matches the bus hierarchy and has a timeseries object for each leaf signal.

Use Array format only for one-dimensional, double, noncomplex inputs. Each column of the array has a time stamp in the first row and a vector containing the corresponding data sample in the subsequent rows.

Parameters

File name: untitled.mat

Variable name: ans

Save format: Timeseries

Decimation: 1

Sample time (-1 for inherited): -1

Parameters

and Dialog Box

File name

The path or file name of the MAT-file in which to store the output. On UNIX systems, the pathname can start with a tilde (~) character signifying your home folder. The default file name is `untitled.mat`. If you specify a file name without path information, Simulink stores the file in the MATLAB working folder. (To determine the working folder, type `pwd` at the MATLAB command line.) If the file already exists, Simulink overwrites it.

Variable name

The name of the matrix contained in the named file. The default name is `ans`.

Save format

The data format (timeseries or array) that the To File block uses for writing data. The default format is `Timeseries`.

Decimation

The decimation factor, `n`, where `n` specifies writing data at every `n`th time that the block executes. The default decimation is `1`, which writes data at every time step.

Sample time

Specifies the sample period and offset at which to collect points. This parameter is useful when you are using a variable-step solver where the interval between time steps might not be constant. The default is `-1`, which inherits the sample time from the driving block. See "Specify Sample Time" for more information.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes

See Also

"Export Runtime Information", From File, From Workspace, To Workspace

To Workspace

Purpose

Write data to MATLAB workspace

Library

Sinks

Description

The To Workspace block inputs a signal and writes the signal data to the MATLAB workspace. During the simulation, the block writes data to an internal buffer. When the simulation is completed or paused, that data is written to the workspace. The block icon shows the name of the array to which the data is written.

To specify the name of the workspace variable to which the To Workspace block writes the data, use the **Variable name** parameter. To specify the data format of the variable, use the **Save format** parameter. You can specify to save the data in one of the following formats:

- A MATLAB `timeseries` object (or structure of MATLAB `timeseries` objects for bus data)
- An array
- Structure
- Structure with time

Saving Data for Use by a From Workspace Block

To use a From Workspace block to play back the sample-based data that was saved by a To Workspace block in a previous simulation, in the To Workspace block, use the `Timeseries` or `Structure with time` format.

Controlling the Amount of Data Saved

For variable-step solvers, to control the amount of data available to the To Workspace block, use the **Model Configuration Parameters > Data Import/Export > Output options** parameter. For example, to have Simulink write data at identical time points over multiple simulations, select the `Produce specified output only` option.

Then use To Workspace block parameters to control when and how much of this data the block writes:

- Use the **Limit data points to last** parameter to specify how many sample points to save. If the simulation generates more data points than the specified maximum, the simulation saves only the most recently generated samples. To capture all the data, set this value to `inf`.
- Use the **Decimation** parameter to have the To Workspace block write data at every `n`th sample, where `n` is the decimation factor. The default decimation, 1, writes data at every time step.
- Use the **Sample time** parameter to specify a sampling interval at which to collect points. This parameter is useful when you are using a variable-step solver where the interval between time steps might not be the same. The default value of -1 causes the block to inherit the sample time from the driving block when determining the points to write. See “Specify Sample Time” in the online documentation for more information.

For example, suppose you have a simulation where the start time is 0, the **Limit data points to last** is 100, the **Decimation** is 1, and the **Sample time** is 0.5. The To Workspace block collects a maximum of 100 points, at time values of 0, 0.5, 1.0, 1.5, ..., seconds. Specifying a **Decimation** value of `1` directs the block to write data at each step.

In a similar example, the **Limit data points to last** is 100 and the **Sample time** is 0.5, but the **Decimation** is 5. In this example, the block collects up to 100 points, at time values of 0, 2.5, 5.0, 7.5, ..., seconds. Specifying a **Decimation** value of `5` directs the block to write data at every fifth sample. The sample time ensures that data is written at these points.

In another example, all parameters are as defined in the first example except that the **Limit data points to last** is 3. In this case, only the last three sample points collected are written to the workspace. If the simulation stop time is 100, data corresponds to times 99.0, 99.5, and 100.0 seconds (three points).

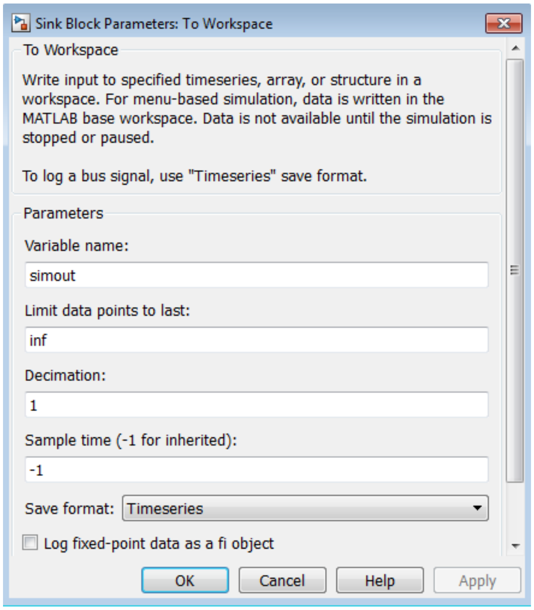
MAT-File Logging

If you enable the **Model Configuration Parameters > Code Generation > Interface > MAT-file logging** parameter, then To Workspace logs its data to a MAT-file.

Data Type Support

The To Workspace block can save to the MATLAB workspace real or complex inputs of any data type that Simulink supports, including fixed-point and enumerated data types, as well as bus objects.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Sink Block Parameters: To Workspace

To Workspace

Write input to specified timeseries, array, or structure in a workspace. For menu-based simulation, data is written in the MATLAB base workspace. Data is not available until the simulation is stopped or paused.

To log a bus signal, use "Timeseries" save format.

Parameters

Variable name: simout

Limit data points to last: inf

Decimation: 1

Sample time (-1 for inherited): -1

Save format: Timeseries

☐ Log fixed-point data as a fi object

Parameters and Dialog Box

Variable name

Specify the name of the variable to which to save the data.

Limit data points to last

Specify the maximum number of input samples to save. The default is `inf`.

Decimation

Specify the decimation factor. The default is `1`.

Sample time

Specify the sample period and offset at which to collect data. This parameter is useful when you are using a variable-step solver where the interval between time steps might not be constant. The default is `-1`, which inherits the sample time from the driving block. See “Specify Sample Time” for more information.

Save format

Specify one of these formats for saving simulation output to the workspace:

- `Timeseries` — (Default)

  Save non-bus signal input as a MATLAB `timeseries` object and bus signal input as a structure of MATLAB `timeseries` objects.

- `Array`

  Save the input as an N-dimensional array where N is one more than the number of dimensions of the input signal. For example, if the input signal is a 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 or vector or a matrix. If the input is a scalar or a vector, each input sample is output as a row of the array. For example, suppose that the name of the output array is `simout`. Then, `simout(1,:)` corresponds to the first sample, `simout(2,:)` corresponds to the second sample, and so on. If the input signal is a matrix, the third dimension of the workspace array corresponds to the values of the input signal at specified sampling point. For example, suppose again that `simout` is the name of the resulting workspace array. Then, `simout(:,:,1)` is the value of the input signal at the first sample point; `simout(:,:,2)` is the value of the input signal at the second sample point; and so on.

- `Structure`

  This format consists of a structure with three fields:

  - `time` — This field is empty for this format.
  - `signals` — A structure with three fields: `values`, `dimensions`, and `label`. The `values` field contains the array of signal values. The `dimensions` field specifies the dimensions of the values array. The `label` field contains the label of the input line.

  - `blockName` — Name of the To Workspace block

- `Structure With Time`

  This format is the same as `Structure`, except that the time field contains a vector of simulation time steps.

  To read To Workspace block output directly with a From Workspace block, use either the `Timeseries` or `Structure with Time` format. For details, see “Techniques for Importing Signal Data”.

  `Structure with Time` format does not support frame-based signals. Use `Array` or `Structure` format instead.

The support for simulation modes varies for the `Timeseries` format and non-`Timeseries` formats (`Array`, `Structure`, and `Structure With Time`).

Simulation Mode	Timeseries Format	Non-Timeseries Formats
Normal	Supported	Supported
Accelerator	Supported for all models, including model references	Supported for models without model references
Rapid Accelerator	Not supported	Supported for models without model references
External mode	Not supported	Supported for models without model references
SIL and PIL	Not supported	Supported for models without model references
Simulink Coder Targets	Not supported	Supported for models without model references

Log fixed-point data as a fi object

Select to log fixed-point data to the MATLAB workspace as a Fixed-Point Designer `fi` object. Otherwise, fixed-point data is logged to the workspace as `double`.

Examples

The following example shows how to use the To Workspace block:

- `aero_vibrati`

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	Yes
Multidimensionalized	Yes

See Also

“Data Export Basics”, From File, From Workspace, To File

Transfer Fcn

Purpose

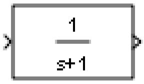
Model linear system by transfer function

Library

Continuous

Description

The Transfer Fcn block models a linear system by a transfer function of the Laplace-domain variable s. The block can model single-input single-output (SISO) and single-input multiple output (SIMO) systems.



Conditions for Using This Block

The Transfer Fcn block assumes the following conditions:

- The transfer function has the form

  $$H(s) = \frac{y(s)}{u(s)} = \frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + num(2)s^{nn-2} + \ldots + num(nn)}{den(1)s^{nd-1} + den(2)s^{nd-2} + \ldots + den(nd)},$$

  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

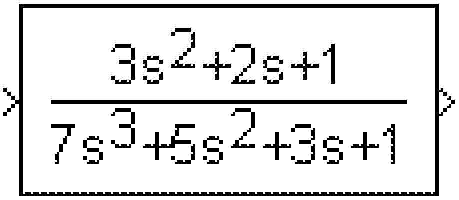
Initial conditions are preset to zero. To specify initial conditions, convert to state-space form using `tf2ss` and use the State-Space block. The `tf2ss` utility provides the A, B, C, and D matrices for the system. For more information, type `help tf2ss` or see the Control System Toolbox™ documentation.

Transfer Function Display on the Block

The Transfer Fcn block displays the transfer function depending on how you specify the numerator and denominator parameters.

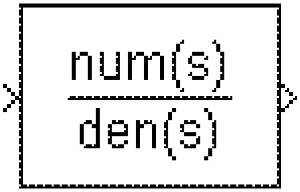
- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified coefficients and powers of *s*. If you specify a variable in parentheses, the block evaluates the variable.

  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)`.

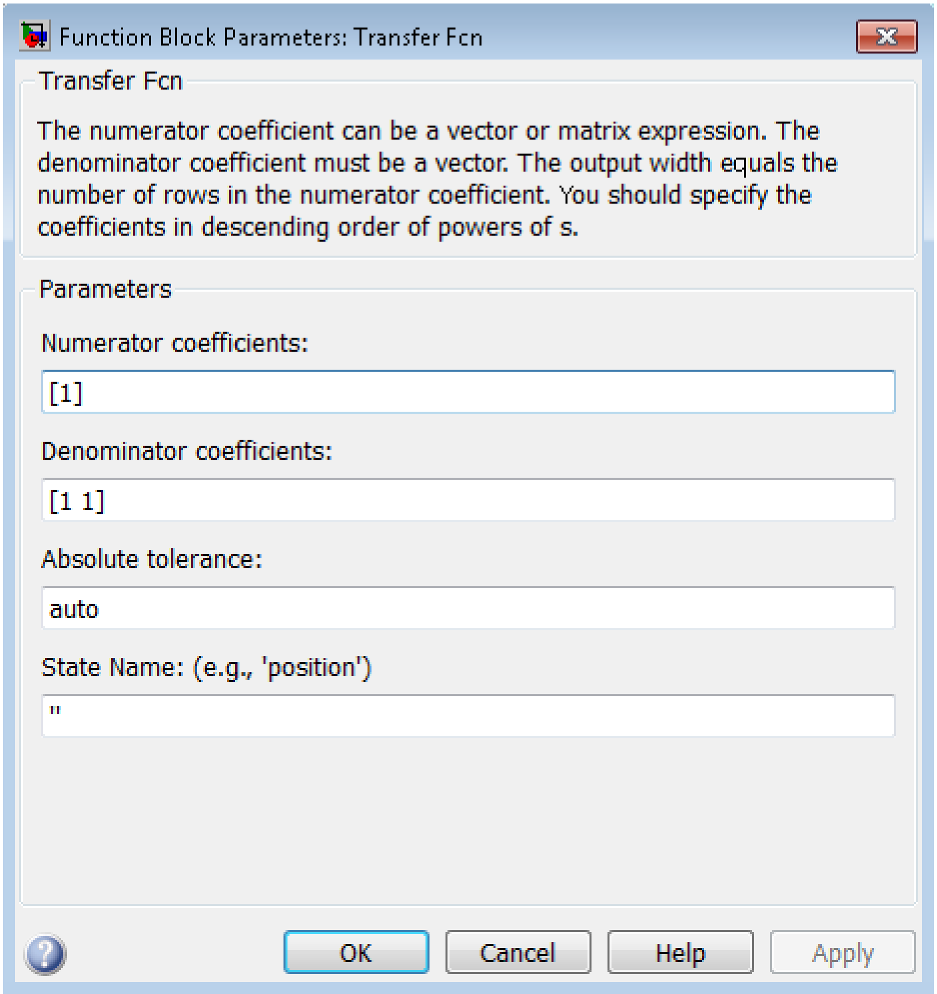
  For example, if you specify **Numerator coefficients** as `num` and **Denominator coefficients** as `den`, the block looks like this:



Data Type Support

The Transfer Fcn block accepts and outputs signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Transfer Fcn

Transfer Fcn

The numerator coefficient can be a vector or matrix expression. The denominator coefficient must be a vector. The output width equals the number of rows in the numerator coefficient. You should specify the coefficients in descending order of powers of s.

Parameters

Numerator coefficients: [1]

Denominator coefficients: [1 1]

Absolute tolerance: auto

State Name: (e.g., 'position') ''

OK
Cancel
Help
Apply

Parameters and Dialog Box

Numerator coefficients

Define the row vector of numerator coefficients.

Settings

Default: `[1]`

Tips

- 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Denominator coefficients

Define the row vector of denominator coefficients.

Settings

Default: `[1 1]`

Tips

- 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, `1`, a real scalar, or a real 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

State Name (e.g., ’position’)

Assign a unique name to each state.

Settings

Default: `'  '`

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 string, cell array, or structure.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Examples

The following Simulink examples show how to use the Transfer Fcn block:

- `slexAircraftExample`

- `sldemo_absbrake`
- `penddemo`

Characteristics

Direct Feedthrough	Only if the lengths of the **Numerator coefficients** and **Denominator coefficients** parameters are equal
Sample Time	Continuous
Scalar Expansion	No
States	Length of **Denominator coefficients** -1
Dimensionalized	Yes, the block expands scalar input into vector output when the transfer function numerator is a matrix.
Zero-Crossing Detection	No

See Also

Discrete Transfer Fcn

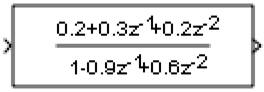
Purpose

Implement Direct Form II realization of transfer function

Library

Additional Math & Discrete / Additional Discrete

Description



The Transfer Fcn Direct Form II block implements a Direct Form II realization of the transfer function that the **Numerator coefficients** and **Denominator coefficients excluding lead** parameters specify. The block supports only single input-single output (SISO) transfer functions.

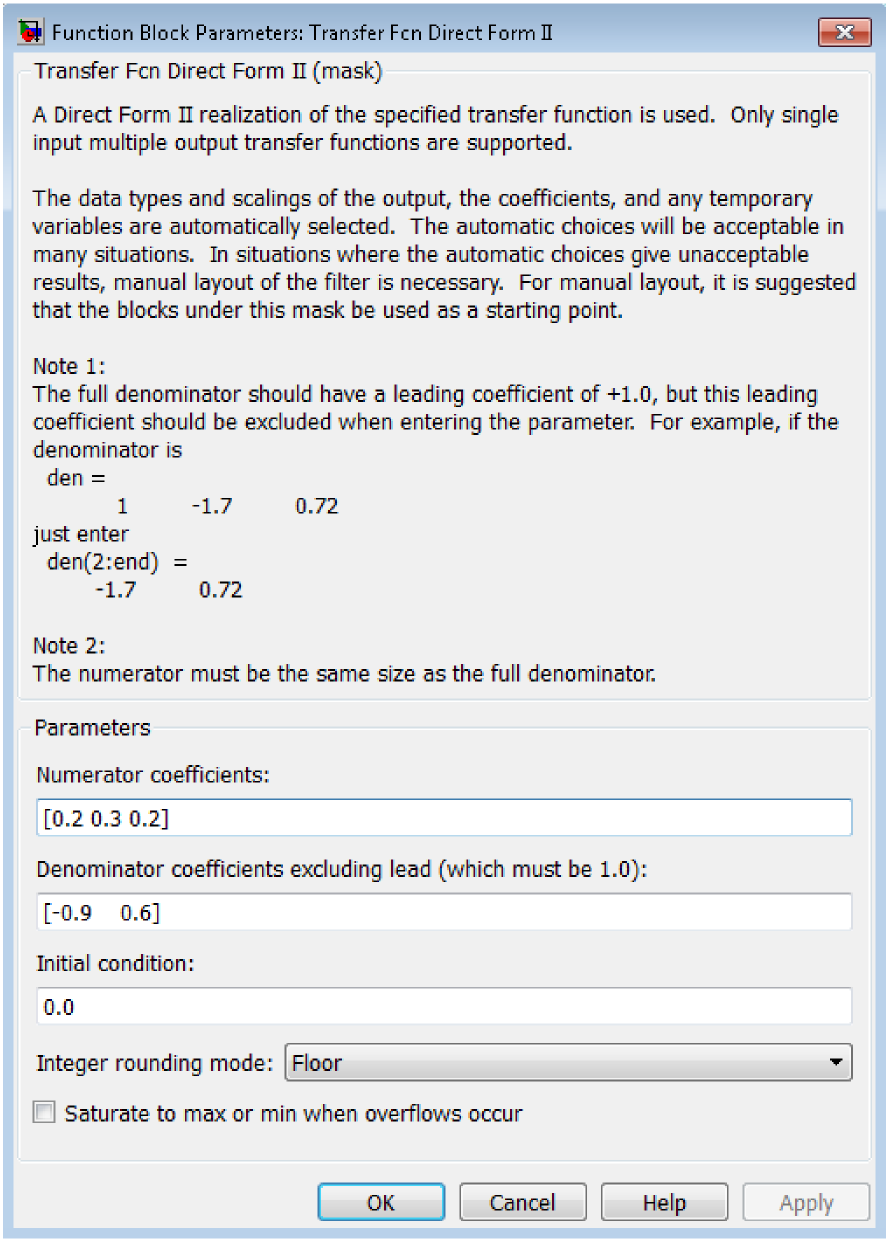
The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Data Type Support

The Transfer Fcn Direct Form II block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Transfer Fcn Direct Form II (mask)

A Direct Form II realization of the specified transfer function is used. Only single input multiple output transfer functions are supported.

The data types and scalings of the output, the coefficients, and any temporary variables are automatically selected. The automatic choices will be acceptable in many situations. In situations where the automatic choices give unacceptable results, manual layout of the filter is necessary. For manual layout, it is suggested that the blocks under this mask be used as a starting point.

Note 1:
The full denominator should have a leading coefficient of +1.0, but this leading coefficient should be excluded when entering the parameter. For example, if the denominator is

```
  den =
          1        -1.7        0.72
just enter
  den(2:end)  =
        -1.7        0.72
```

Note 2:
The numerator must be the same size as the full denominator.

Parameters

Numerator coefficients: [0.2 0.3 0.2]

Denominator coefficients excluding lead (which must be 1.0): [-0.9   0.6]

Initial condition: 0.0

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Parameters and Dialog Box

Numerator coefficients
Specify the numerator coefficients.

Denominator coefficients excluding lead
Specify the denominator coefficients, excluding the leading coefficient, which must be 1.0.

Initial condition

Set the initial condition.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of initial conditions
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Transfer Fcn Direct Form II Time Varying

Transfer Fcn Direct Form II Time Varying

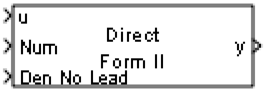
Purpose

Implement time varying Direct Form II realization of transfer function

Library

Additional Math & Discrete / Additional Discrete

Description



The Transfer Fcn Direct Form II Time Varying block implements a Direct Form II realization of the specified transfer function. The block supports only single input-single output (SISO) transfer functions.

The input signal labeled `Den No Lead` contains the denominator coefficients of the transfer function. The full denominator has a leading coefficient of one, but it is excluded from the input signal. For example, a denominator of [1 -1.7 0.72] is represented by a signal with the value [-1.7 0.72]. The input signal labeled `Num` contains the numerator coefficients. The data types of the numerator and denominator coefficients can be different, but the length of the numerator vector and the full denominator vector must be the same. Pad the numerator vector with zeros, if needed.

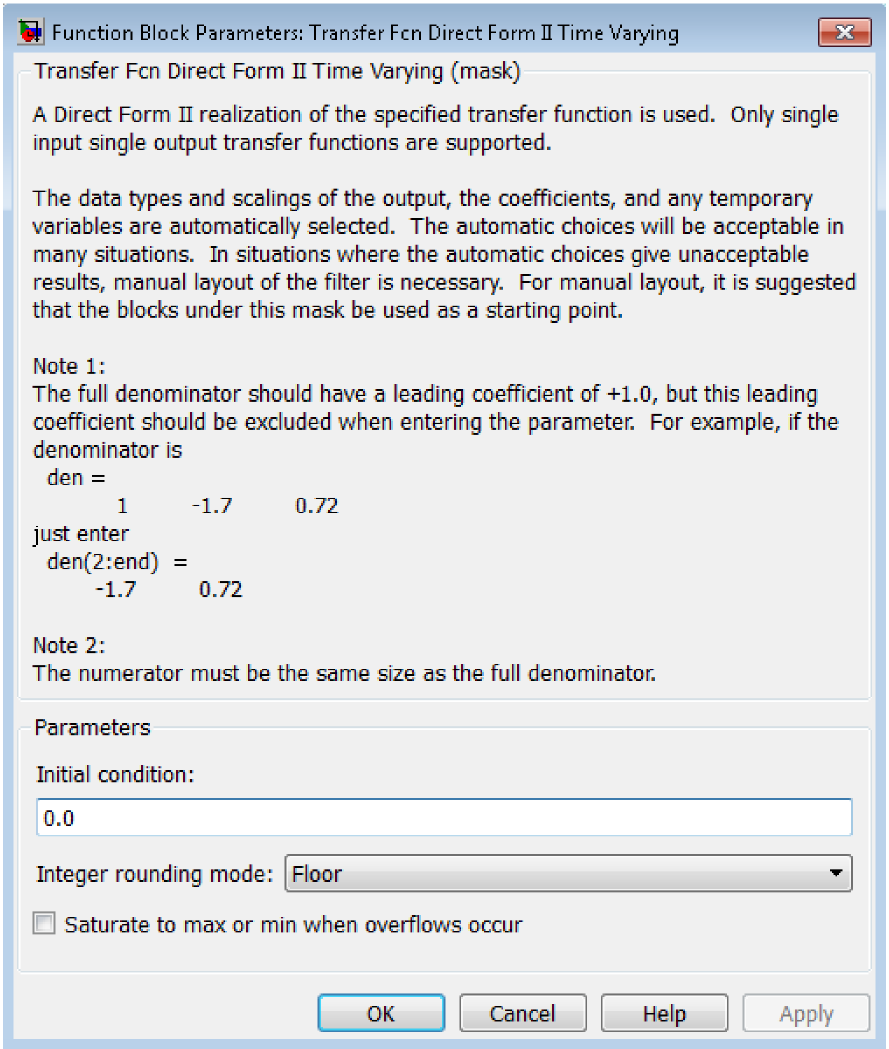
The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Data Type Support

The Transfer Fcn Direct Form II Time Varying block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Transfer Fcn Direct Form II Time Varying

Transfer Fcn Direct Form II Time Varying (mask)

A Direct Form II realization of the specified transfer function is used. Only single input single output transfer functions are supported.

The data types and scalings of the output, the coefficients, and any temporary variables are automatically selected. The automatic choices will be acceptable in many situations. In situations where the automatic choices give unacceptable results, manual layout of the filter is necessary. For manual layout, it is suggested that the blocks under this mask be used as a starting point.

Note 1:
The full denominator should have a leading coefficient of +1.0, but this leading coefficient should be excluded when entering the parameter. For example, if the denominator is

```
 den =
          1        -1.7        0.72
just enter
 den(2:end)  =
       -1.7        0.72
```

Note 2:
The numerator must be the same size as the full denominator.

Parameters

Initial condition: 0.0

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Parameters and Dialog Box

Initial condition

Set the initial condition.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of initial conditions
Multidimensionalized	No
Zero-Crossing Detection	No

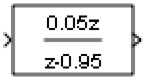
See Also

Transfer Fcn Direct Form II

Purpose Implement discrete-time first order transfer function

Library Discrete

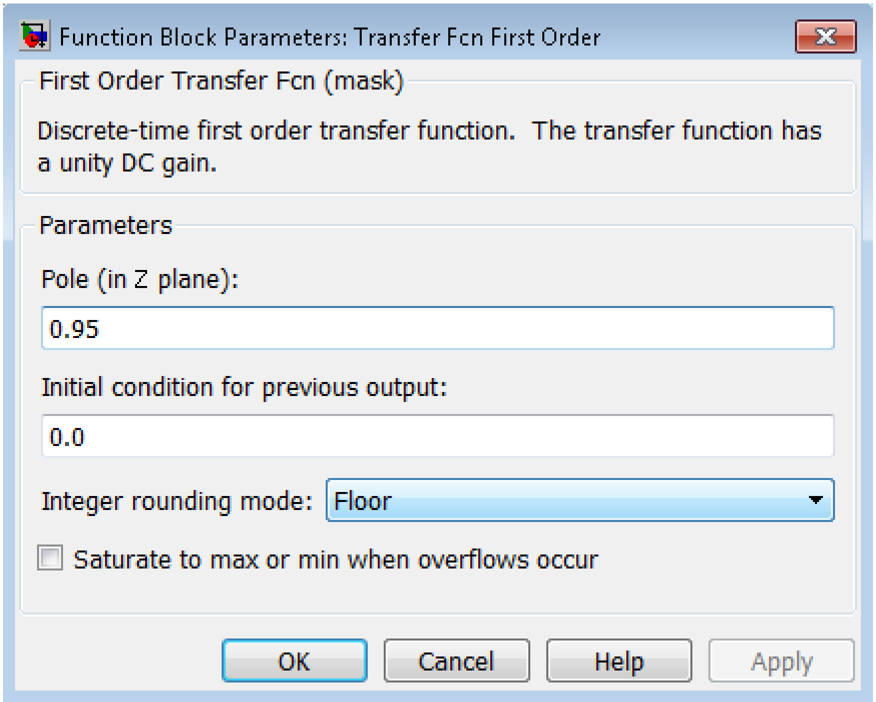
Description The Transfer Fcn First Order block implements a discrete-time first order transfer function of the input. The transfer function has a unity DC gain.



Data Type Support The Transfer Fcn First Order block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



First Order Transfer Fcn (mask)

Discrete-time first order transfer function. The transfer function has a unity DC gain.

Parameters

Pole (in Z plane): 0.95

Initial condition for previous output: 0.0

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Parameters and Dialog Box

Pole (in Z plane)
Set the pole.

Initial condition for previous output
Set the initial condition for the previous output.

Integer rounding mode
Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur
Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

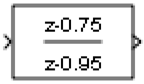
Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of initial conditions
Zero-Crossing Detection	No

Purpose Implement discrete-time lead or lag compensator

Library Discrete

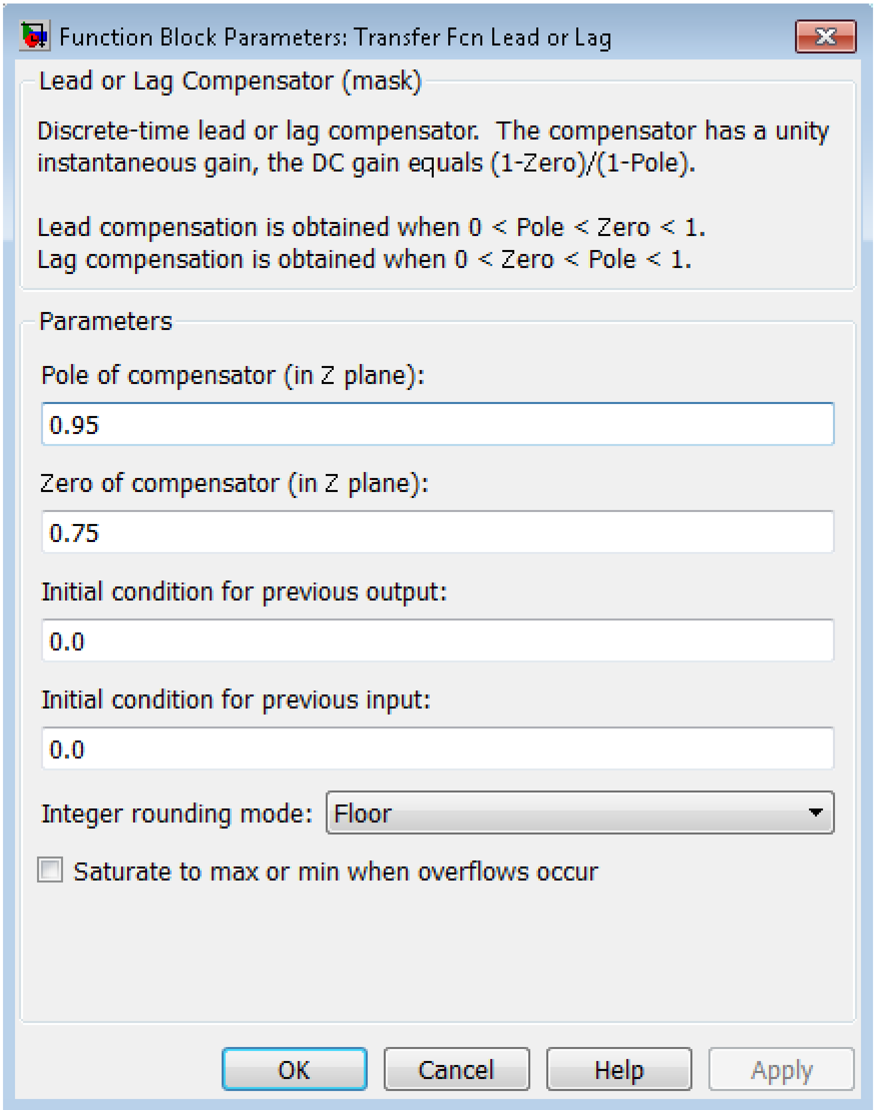
Description The Transfer Fcn Lead or Lag block implements a discrete-time lead or lag compensator of the input. The instantaneous gain of the compensator is one, and the DC gain is equal to (1-z)/(1-p), where z is the zero and p is the pole of the compensator.



The block implements a lead compensator when $0 < z < p < 1$, and implements a lag compensator when $0 < p < z < 1$.

Data Type Support The Transfer Fcn Lead or Lag block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Function Block Parameters: Transfer Fcn Lead or Lag

Lead or Lag Compensator (mask)

Discrete-time lead or lag compensator. The compensator has a unity instantaneous gain, the DC gain equals (1-Zero)/(1-Pole).

Lead compensation is obtained when 0 < Pole < Zero < 1.
Lag compensation is obtained when 0 < Zero < Pole < 1.

Parameters

Pole of compensator (in Z plane): 0.95

Zero of compensator (in Z plane): 0.75

Initial condition for previous output: 0.0

Initial condition for previous input: 0.0

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Parameters

and Dialog Box

Pole of compensator (in Z plane)
Set the pole.

Zero of compensator (in Z plane)
Set the zero.

Initial condition for previous output
Set the initial condition for the previous output.

Initial condition for previous input
Set the initial condition for the previous input.

Integer rounding mode
Specify the rounding mode for fixed-point operations. For more information, see “Rounding”. in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur
Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of initial conditions
Zero-Crossing Detection	No

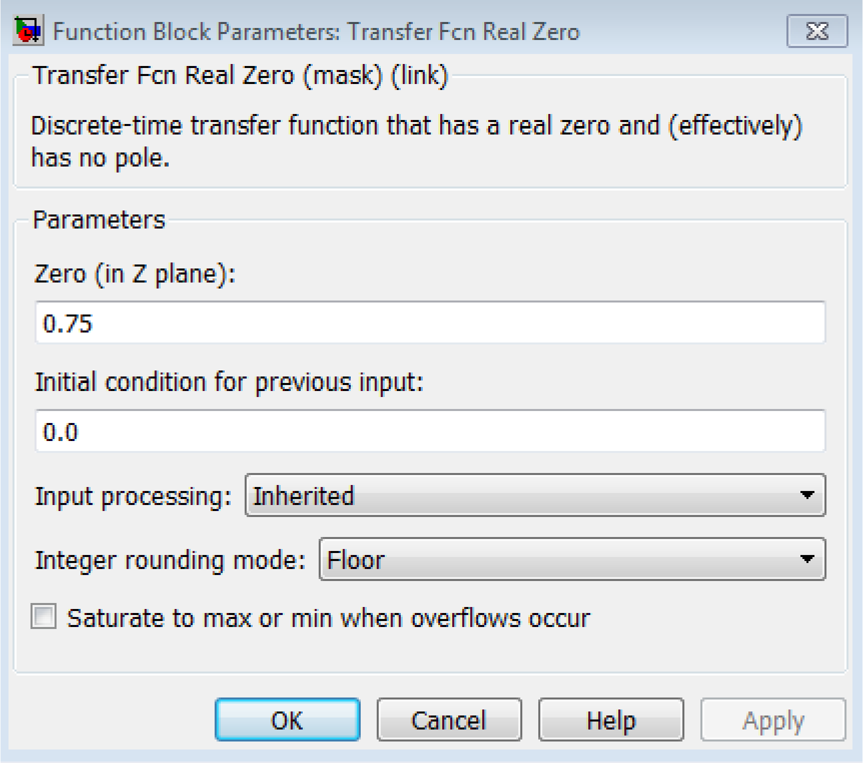
Purpose Implement discrete-time transfer function that has real zero and no pole

Library Discrete

Description The Transfer Fcn Real Zero block implements a discrete-time transfer function that has a real zero and effectively no pole.

Data Type Support The Transfer Fcn Real Zero block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Transfer Fcn Real Zero (mask) (link)

Discrete-time transfer function that has a real zero and (effectively) has no pole.

Parameters

Zero (in Z plane): 0.75

Initial condition for previous input: 0.0

Input processing: Inherited

Integer rounding mode: Floor

☐ Saturate to max or min when overflows occur

Parameters and Dialog Box

Zero (in Z plane)
Set the zero.

Initial condition for previous input
Set the initial condition for the previous input.

Input processing
Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).

- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal `u`	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes

Input Signal u	Input Processing Mode	Block Works?
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see "Sample- and Frame-Based Concepts" in the DSP System Toolbox documentation.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see "Rounding". in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes, of initial conditions
Zero-Crossing Detection	No

Transport Delay

Purpose

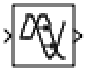
Delay input by given amount of time

Library

Continuous

Description

The Transport Delay block delays the input by a specified amount of time. You can use this block to simulate a time delay. The input to this block should be a continuous signal.



At the start of simulation, the block outputs the **Initial output** parameter until the simulation time exceeds the **Time delay** parameter. Then, the block begins generating the delayed input. During simulation, the block stores input points and simulation times in a buffer. You specify this size with the **Initial buffer size** parameter.

When you want output at a time that does not correspond to times of the stored input values, the block interpolates linearly between points. When the delay is smaller than the step size, the block extrapolates from the last output point, which can produce inaccurate results. Because the block does not have direct feedthrough, it cannot use the current input to calculate an output value. For example, consider a fixed-step simulation with a step size of 1 and the current time at $t = 5$. If the delay is 0.5, the block must generate a point at $t = 4.5$. Because the most recent stored time value is at $t = 4$, the block performs forward extrapolation.

The Transport Delay block does not interpolate discrete signals. Instead, the block returns the discrete value at the required time.

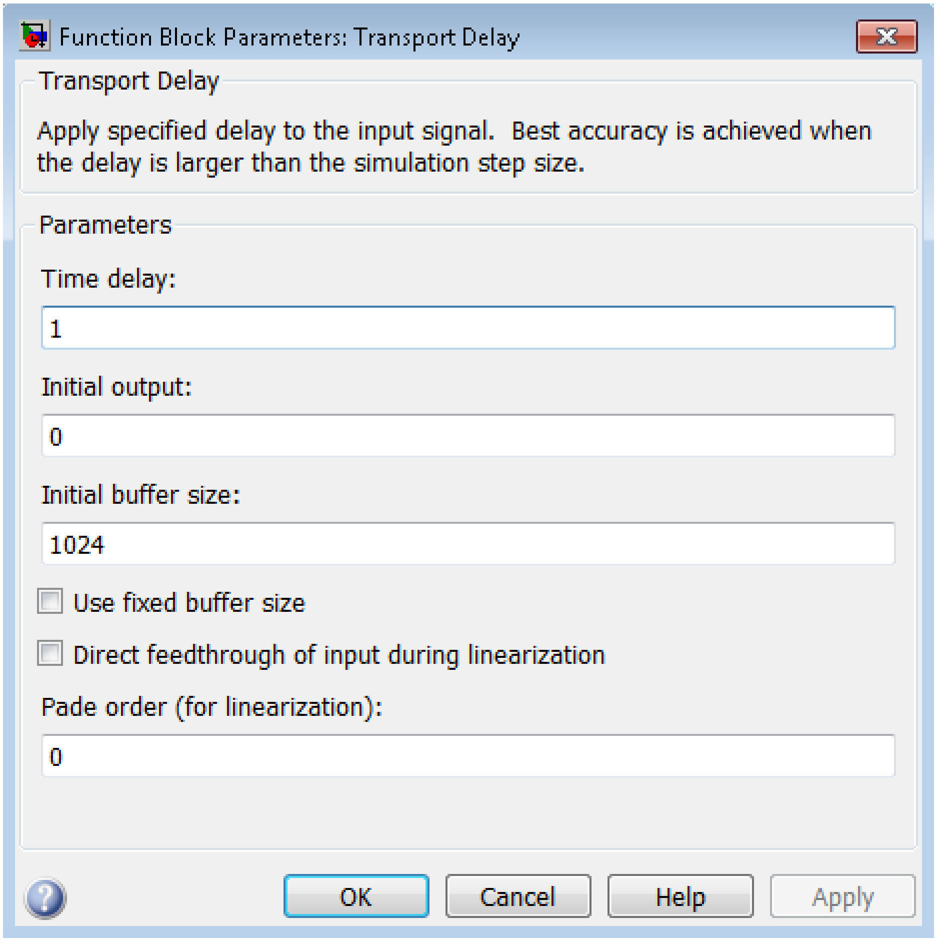
This block differs from the Unit Delay block, which delays and holds the output on sample hits only.

Tip Avoid using `linmod` to linearize a model that contains a Transport Delay block. For more information, see “Linearizing Models” in the Simulink documentation.

Data Type Support

The Transport Delay block accepts and outputs real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Transport Delay

Transport Delay

Apply specified delay to the input signal. Best accuracy is achieved when the delay is larger than the simulation step size.

Parameters

Time delay: 1

Initial output: 0

Initial buffer size: 1024

☐ Use fixed buffer size

☐ Direct feedthrough of input during linearization

Pade order (for linearization): 0

Parameters and Dialog Box

Time delay

Specify the amount of simulation time to delay the input signal before propagation to the output.

Settings

Default: `1`

This value must be nonnegative.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Initial output

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Settings

Default: `0`

The initial output of this block cannot be `inf` or `NaN`.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Initial buffer size

Define the initial memory allocation for the number of input points to store.

Settings

Default: `1024`

- If the number of input points exceeds the initial buffer size, the block allocates additional memory.
- After simulation ends, a message shows the total buffer size needed.

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Use fixed buffer size

Specify use of a fixed-size buffer to save input data from previous time steps.

Settings

Default: Off

☑ On

The block uses a fixed-size buffer.

☐ Off

The block does not use a fixed-size buffer.

The **Initial buffer size** parameter specifies the size of the buffer. If the buffer is full, new data replaces data already in the buffer. Simulink software uses linear extrapolation to estimate output values that are not in the buffer.

Note If you have a Simulink Coder license, ERT or GRT code generation uses a fixed-size buffer even if you do not select this check box.

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Direct feedthrough of input during linearization

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

Settings

Default: Off

☑ On

Enables direct feedthrough of input.

☐ Off

Disables direct feedthrough of input.

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Pade order (for linearization)

Set the order of the Pade approximation for linearization routines.

Settings

Default: `0`

- The default value is 0, which results in a unity gain with no dynamic states.
- Setting the order to a positive integer `n` adds `n` states to your model, but results in a more accurate linear model of the transport delay.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Direct Feedthrough	No
Sample Time	Continuous
Scalar Expansion	Yes, of input and all parameters except **Initial buffer size**
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also

Variable Time Delay

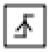
Purpose

Add trigger port to model or subsystem

Library

Ports & Subsystems

Description



Adding a Trigger block to a model allows an external signal to trigger its execution. You can add a trigger port to a root-level model or to a subsystem.

Configure the Trigger block to execute the model either:

- Once on each integration step, when the value of the external signal changes in a way that you specify.
- Multiple times during a time step, when the external signal is a function-call from a Function-Call Generator block or S-function.

Include only one Trigger block in a model or a subsystem.

Specify the properties that the trigger port enforces for any incoming signal, using the **Signal Attributes** tab.

The Trigger block supports signal label propagation.

For more information, see:

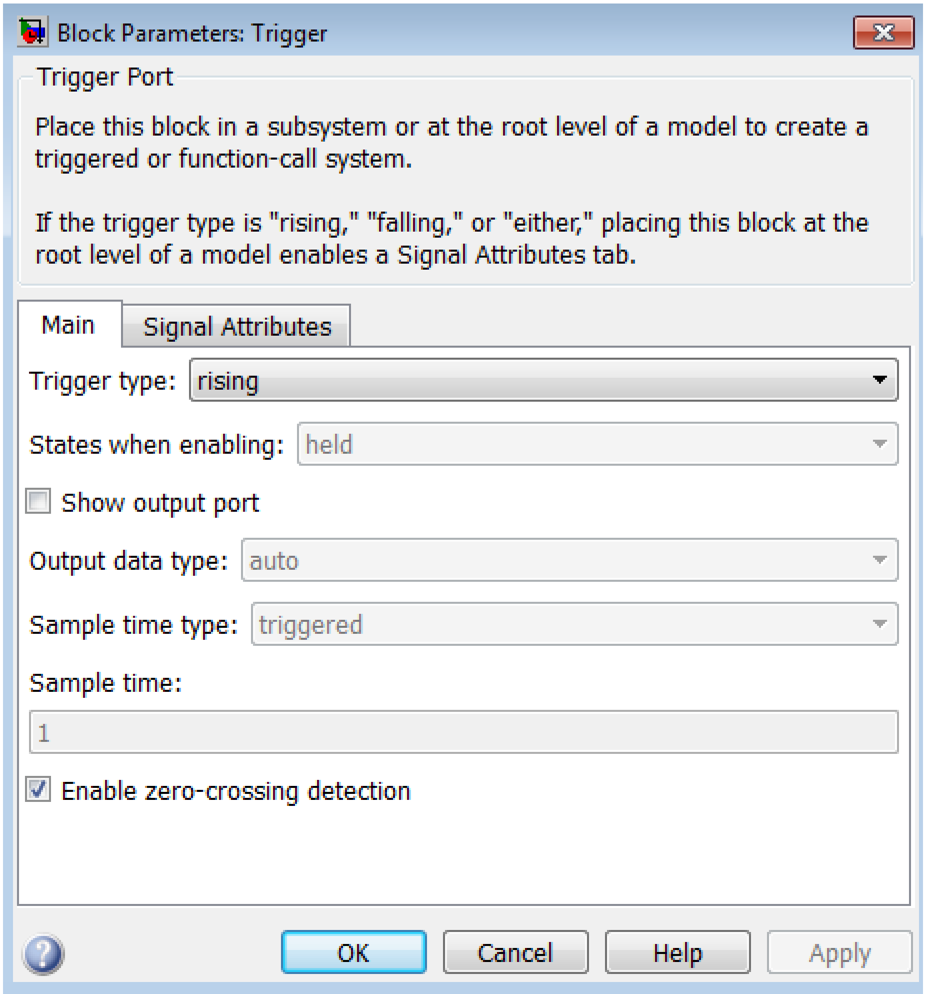
- “Triggered Subsystems”
- “Create Conditional Models”
- “Function-Call Models”
- “Function-Call Subsystems”

Data Type Support

The Trigger block accepts signals of supported Simulink numeric data types, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** tab of the Trigger block dialog box appears as follows:



Block Parameters: Trigger

Trigger Port

Place this block in a subsystem or at the root level of a model to create a triggered or function-call system.

If the trigger type is "rising," "falling," or "either," placing this block at the root level of a model enables a Signal Attributes tab.

Main
Signal Attributes

Trigger type: rising

States when enabling: held

☐ Show output port

Output data type: auto

Sample time type: triggered

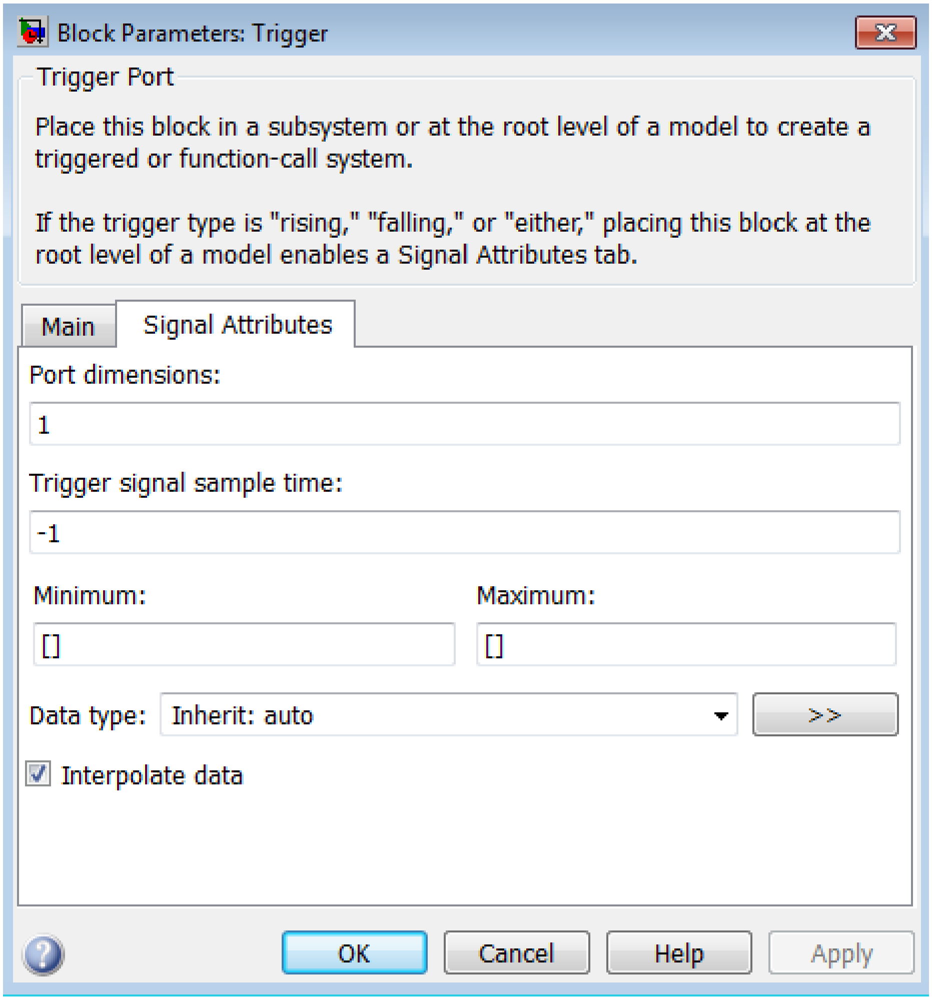
Sample time:

1

☑ Enable zero-crossing detection

OK
Cancel
Help
Apply

The **Signal Attributes** tab of the Trigger block dialog box appears as follows:



Block Parameters: Trigger

Trigger Port

Place this block in a subsystem or at the root level of a model to create a triggered or function-call system.

If the trigger type is "rising," "falling," or "either," placing this block at the root level of a model enables a Signal Attributes tab.

Main
Signal Attributes

Port dimensions: 1

Trigger signal sample time: -1

Minimum: []

Maximum: []

Data type: Inherit: auto

☑ Interpolate data

>>
OK
Cancel
Help
Apply

- “Trigger type” on page 1-1909
- “States when enabling” on page 1-1910
- “Propagate sizes of variable-size signals” on page 1-1912

- "Show output port" on page 1-1913
- "Output data type" on page 1-1914
- "Sample time type" on page 1-1915
- "Sample time" on page 1-1916
- "Enable zero-crossing detection" on page 1-1917
- "Port dimensions" on page 1-1918
- "Trigger signal sample time" on page 1-1919
- "Minimum" on page 1-1920
- "Maximum" on page 1-1921
- "Data type" on page 1-1922
- "Mode" on page 1-1924
- "Data type override" on page 1-1926
- "Signedness" on page 1-1927
- "Scaling" on page 1-1928
- "Word length" on page 1-1930
- "Fraction length" on page 1-1931
- "Slope" on page 1-1932
- "Bias" on page 1-1932
- "Interpolate data" on page 1-1933

Trigger type

Select the type of event that triggers execution of the subsystem.

Settings

Default: `rising`

`rising`
: Triggers execution of the model or subsystem when the control signal rises from a negative or zero value to a positive value. If the initial value is negative, rising to zero triggers execution.

`falling`
: Triggers execution of the model or subsystem when the control signal falls from a positive or a zero value to a negative value. If the initial value is positive, falling to zero triggers execution.

`either`
: Triggers execution of the model or subsystem when the signal is either rising or falling.

`function-call`
: Allows a Function-Call Generator or S-function to control execution of the subsystem or model.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

States when enabling

Specify the state values when triggered by a function-call.

Settings

Default: `held`

`held`
Leaves the states at their current values.

`reset`
Resets the states.

`inherit`
Uses the `held`/`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.

The parameter setting applies only if the model explicitly enables and disables the function-call subsystem. For example:

- The function-call subsystem resides in an enabled subsystem. In this case, the model enables and disables the function-call subsystem along with the parent subsystem.
- The function-call initiator that controls the function-call subsystem resides in an enabled subsystem. In this case, the model enables and disables the function-call subsystem along with the enabled subsystem containing the function-call initiator.
- The function-call initiator is a Stateflow event bound to a particular state. See "Control Function-Call Subsystems Using Bind Actions" in the Stateflow documentation.

- The function-call initiator is an S-function that explicitly enables and disables the function-call subsystem. See `ssEnableSystemWithTid` for an example.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Propagate sizes of variable-size signals

Specify when to propagate a variable-size signal.

Settings

Default: `During execution`

`Only when enabling`
: Propagates variable-size signals only when enabling the model or subsystem containing the Trigger block.

`During execution`
: Propagates variable-size signals at each time step.

Dependency

Select `Function-call` from the **Trigger type** list to enable this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Show output port

Select this check box to output the trigger signal.

Settings

Default: On

☑ On

Shows the Trigger block output port and outputs the trigger signal. This action allows the system to determine which signal caused the trigger. The width of the 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 otherwise

☐ Off

Removes the output port.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Output data type

Specify the trigger output data type.

Settings

Default: auto

`auto`
: Specifies that the data type is the same as the port connected to output.

`double`
: Sets the date type to double.

`int8`
: Sets the data type to integer.

Dependencies

To enable this parameter, select the **Show output port** check box.

The Trigger block ignores the **Data type override** setting of the Fixed-Point Tool.

Sample time type

Specify the calling frequency of a subsystem.

Settings

Default: `triggered`

`triggered`
: Applies to applications that do not have a periodic calling frequency.

`periodic`
: Applies 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

Select `Function-call` from the **Trigger type** list to enable this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Sample time

Specify the calling rate for blocks.

Settings

Default: `1`

Set this parameter to the sample time you expect for the calling rate of the function-call subsystem containing this Trigger block. If the actual calling rate for the subsystem differs from the rate that this parameter specifies, Simulink displays an error.

Dependency

Setting **Trigger type** to `function-call` and **Sample time type** to `periodic` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection.

Settings

Default: On

☑ On
    Detects zero crossings.

☐ Off
    Does not detect zero crossings.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Port dimensions

Specify the dimensions of the input signal to the block.

Settings

Default: `1`

Valid values are:

Value	Description
`n`	Accepts vector signal of width `n`
`[m n]`	Accepts matrix signal having `m` rows and `n` columns

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Trigger signal sample time

Specify the rate at which the block driving the triggered signal is expected to run.

Settings

Default: `-1`

To inherit the sample time, set this parameter to `-1`.

See “Specify Sample Time” for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Minimum

Specify the minimum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Maximum

Specify the maximum value that the block should output.

Settings

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Data type

Specify the expected data type of the signal feeding the trigger port.

Settings

Default: `Inherit:  auto`

`Inherit:  auto`
  Data type is `double`

`double`
  Data type is `double`.

`single`
  Data type is `single`.

`int8`
  Data type is `int8`.

`uint8`
  Data type is `uint8`.

`int16`
  Data type is `int16`.

`uint16`
  Data type is `uint16`.

`int32`
  Data type is `int32`.

`uint32`
  Data type is `uint32`.

`boolean`
  Data type is `boolean`.

`fixdt(1,16,0)`
  Data type is fixed point, `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
  Data type is fixed point, `fixdt(1,16,2^0,0)`.

`Enum: <class name>`
Data type is enumerated, for example, `Enum: Basic Colors`.

`<data type expression>`
The name of a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Mode

Select the category of data to specify.

Settings

Default: `Inherit`

`Inherit`

Inheritance rule for data types. Selecting `Inherit` enables a second list.

`Built in`

Built-in data types. Selecting `Built in` enables a second list. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

`Fixed point`

Fixed-point data types.

`Enumerated`

Enumerated data types. Selecting `Enumerated` enables a second text box, where you can enter the class name.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second text box, where you can enter the expression.

Dependency

To enable this parameter, click **Show data type assistant**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Data type override

Specify data type override mode for this signal.

Settings

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

Tip

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

Dependency

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Settings

Default: `Signed`

`Signed`
  Specify the fixed-point data as signed.

`Unsigned`
  Specify the fixed-point data as unsigned.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Settings

Default: `Best precision`, `Binary point`, `Integer`

`Binary point`
  Specify binary point location.

`Slope and bias`
  Enter slope and bias.

`Best precision`
  Specify best-precision values. This option appears for some blocks.

`Integer`
  Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0. This option appears for some blocks.

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Settings

Default: `16`

Minimum: `0`

Maximum: `32`

Dependencies

Selecting **Mode** > `Fixed point` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Settings

Default: `0`

Binary points can be positive or negative integers.

Dependencies

Selecting **Scaling** > `Binary point` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

See Also

For more information, see "Specifying a Fixed-Point Data Type".

Slope

Specify slope for the fixed-point data type.

Settings

Default: `2^0`

Specify any positive real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Settings

Default: `0`

Specify any real number.

Dependencies

Selecting **Scaling** > `Slope and bias` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

For more information, see “Specifying a Fixed-Point Data Type”.

Interpolate data

Cause the block to interpolate or extrapolate output at time steps for which no corresponding workspace data exists when loading data from the workspace.

Settings

Default: On

☑ On

Causes the block to interpolate or extrapolate output at time steps for which no corresponding workspace data exists when loading data from the workspace.

☐ Off

Does not cause the block to interpolate or extrapolate output at time steps for which no corresponding workspace data exists when loading data from the workspace.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Sample Time	Specified by the **Sample time** parameter if: • **Trigger type** is `function-call` • **Sample time type** is `periodic` Otherwise, specified by the signal at the trigger port.
Dimensionalized	Yes

Virtual	Yes, when the output port is *not* present For more information, see “Virtual Blocks” in the Simulink documentation.
Zero-Crossing Detection	Yes, if enabled

See Also

- “Create Conditional Models”
- Triggered Subsystem
- Function-Call Subsystem

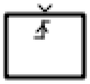
Purpose

Generate linear models in base workspace when triggered

Library

Model-Wide Utilities

Description



When triggered, this block calls `linmod` or `dlinmod` to create a linear model for the system at the current operating point. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.

The block sets the following model parameters to the indicated values:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`

The name of the structure used to save the snapshots is the name of the model appended by `_Trigger_Based_Linearization`, for example, `vdp_Trigger_Based_Linearization`. The structure has the following fields:

Field	Description
`a`	The A matrix of the linearization
`b`	The B matrix of the linearization
`c`	The C matrix of the linearization
`d`	The D matrix of the linearization
`StateName`	Names of the model's states
`OutputName`	Names of the model's output ports
`InputName`	Names of the model's input ports

Field	Description
`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

Use the Timed-Based Linearization block to generate linear models at predetermined times.

You can use state and simulation time logging to extract the model states at operating points. For example, suppose that you want to get the states of the vdp example model when the signal `x1` triggers the Trigger-Based Linearization block on a rising edge.

1. Open the model and drag an instance of this block from the Model-Wide Utilities library and drop the instance into the model.
2. Connect the block's trigger port to the signal labeled `x1`.
3. Open the model's **Model Configuration Parameters** dialog box.
4. Select the **Data Import/Export** pane.
5. Check **States** and **Time** on the **Save to Workspace** control panel
6. Select OK to confirm the selections and close the dialog box.
7. Simulate the model.

   At the end of the simulation, the following variables appear in the MATLAB workspace: `vdp_Trigger_Based_Linearization`, `tout`, and `xout`.

8. Get the index to the first operating point time by entering the following at the MATLAB command line:

```
ind1 = find(vdp_Trigger_Based_Linearization(1).OperPoint.t==tout);
```

9 Get the state vector at this operating point.

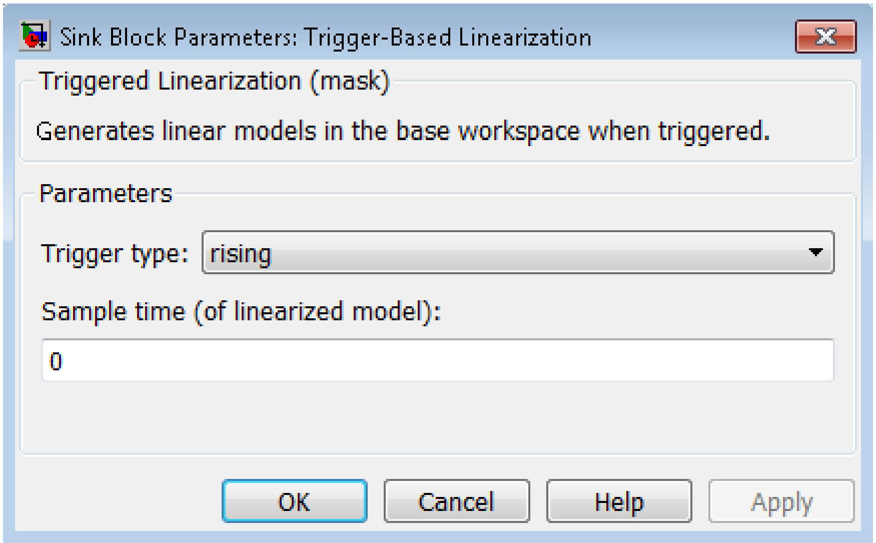
```
x1 = xout(ind1,:);
```

Data Type Support

The trigger port accepts signals of any numeric data type that Simulink supports.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



Sink Block Parameters: Trigger-Based Linearization

Triggered Linearization (mask)

Generates linear models in the base workspace when triggered.

Parameters

Trigger type: rising

Sample time (of linearized model): 0

Trigger type

Type of event on the trigger input signal that triggers generation of a linear model. See the **Trigger type** parameter of the Trigger block for an explanation of the various trigger types that you can select.

Sample time (of linearized model)

Specify a sample time to create a discrete-time linearization of the model (see “Discrete-Time System Linearization” on page 2-28).

Characteristics

Sample Time	Specified in the **Sample time** parameter
Dimensionalized	No

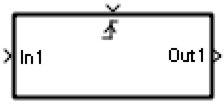
See Also

Timed-Based Linearization

Purpose Represent subsystem whose execution is triggered by external input

Library Ports & Subsystems

Description



This block is a Subsystem block that is preconfigured to serve as the starting point for creating a triggered subsystem (see "Triggered Subsystems").

Trigonometric Function

Purpose Specified trigonometric function on input

Library Math Operations

Description

Supported Functions

The Trigonometric Function block performs common trigonometric functions. You can select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
`sin`	Sine of the input	`sin(`*u*`)`	`sin`
`cos`	Cosine of the input	`cos(`*u*`)`	`cos`
`tan`	Tangent of the input	`tan(`*u*`)`	`tan`
`asin`	Inverse sine of the input	`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`
`sinh`	Hyperbolic sine of the input	`sinh(`*u*`)`	`sinh`
`cosh`	Hyperbolic cosine of the input	`cosh(`*u*`)`	`cosh`

Function	**Description**	**Mathematical Expression**	**MATLAB Equivalent**
`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	—	—

The block output is the result of applying the function to one or more inputs in radians. Each function supports:

- Scalar operations
- Element-wise vector and matrix operations

Note Not all compilers support the `asinh`, `acosh`, and `atanh` functions. If you use a compiler that does not support those functions, a warning appears and the generated code fails to link.

Block Appearance for the atan2 Function

If you select the `atan2` function, the block shows two inputs. The first input is the *y*-axis or imaginary part of the function argument. The second input is the *x*-axis or real part of the function argument. (See "How to Rotate a Block" in the Simulink documentation for a description of the port order for various block orientations.)

Block Appearance for the sincos Function

If you select the `sincos` function, the block shows two outputs. The first output is the sine of the function argument, and the second output is the cosine of the function argument.

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is among one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see [1], [2]). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

Data Type Support

The block accepts input signals of the following data types:

Functions	Input Data Types
• `sin` • `cos` • `sincos` • `cos + jsin` • `atan2`	• Floating point • Fixed point (only when **Approximation method** is `CORDIC`)
• `tan` • `asin` • `acos` • `atan` • `sinh` • `cosh` • `tanh` • `asinh` • `acosh` • `atanh`	• Floating point

Complex input signals are supported for all functions in this block, except `atan2`.

You can use floating-point input signals when you set **Approximation method** to `None` or `CORDIC`. However, the block output data type depends on which of these approximation method options you choose.

Input Data Type	**Approximation Method**	**Output Data Type**
Floating point	`None`	Depends on your selection for **Output signal type**. Options are `auto` (same data type as input), `real`, or `complex`.
Floating point	`CORDIC`	Same as input. **Output signal type** is not available when you use the CORDIC approximation method to compute the block output.

You can use fixed-point input signals only when **Approximation method** is set to `CORDIC`. The CORDIC approximation is available for the `sin`, `cos`, `sincos`, `cos + jsin`, and `atan2` functions. For the `atan2` function, the relationship between input and output data types depends also on whether the fixed-point input is signed or unsigned.

Input Data Type	**Function**	**Output Data Type**
Fixed point, signed or unsigned	`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)

For CORDIC approximations:

- Input must be real for the `sin`, `cos`, `sincos`, `cos + jsin`, and `atan2` functions.
- Output is real for the `sin`, `cos`, `sincos`, and `atan2` functions.
- Output is complex for the `cos + jsin` function.

Invalid Inputs for CORDIC Approximations

If you use the CORDIC approximation method (see "Definitions" on page 1-1942), the block input has some further requirements.

For the `sin`, `cos`, `sincos`, and `cos + jsin` functions:

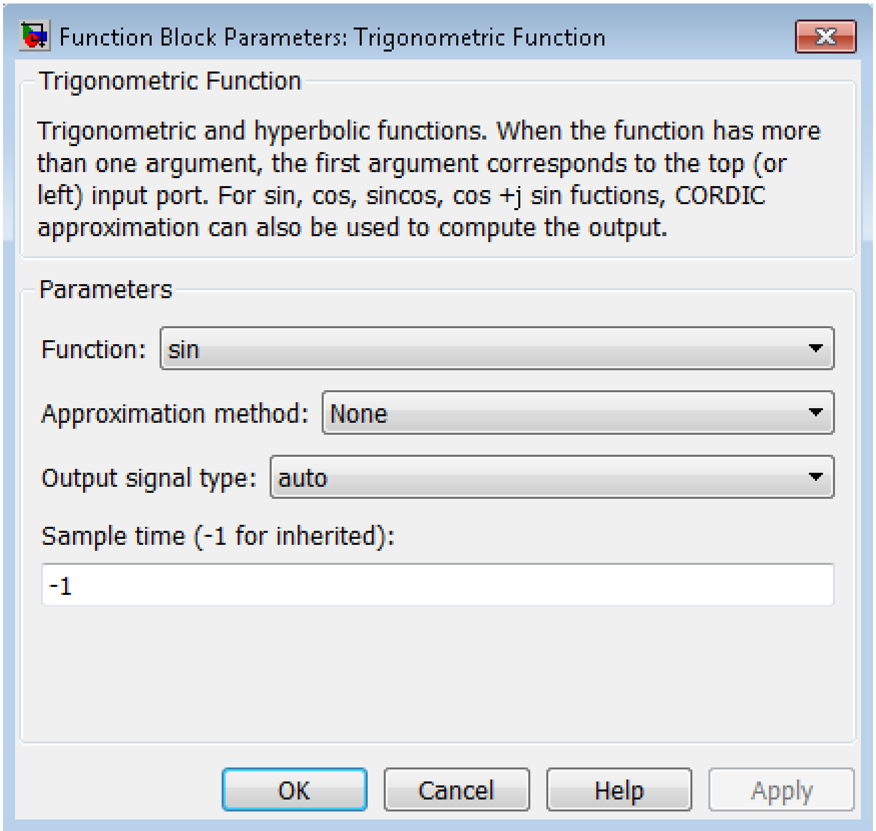
- When you use signed fixed-point types, the input angle must fall within the range [–2π, 2π) radians.
- When you use unsigned fixed-point types, the input angle must fall within the range [0, 2π) radians.

For the `atan2` function:

- Inputs must be the same size, or at least one value must be a scalar value.
- Both inputs must have the same data type.
- When you use signed fixed-point types, the word length must be `126` or less.
- When you use unsigned fixed-point types, the word length must be `125` or less.

This table summarizes what happens for an invalid input.

Block Usage	Effect of Invalid Input
Simulation	An error appears.
Generated code	Undefined behavior occurs. Avoid relying on undefined behavior for generated code or Accelerator modes.
Accelerator modes	



Function Block Parameters: Trigonometric Function

Trigonometric Function

Trigonometric and hyperbolic functions. When the function has more than one argument, the first argument corresponds to the top (or left) input port. For sin, cos, sincos, cos +j sin fuctions, CORDIC approximation can also be used to compute the output.

Parameters

Function: sin

Approximation method: None

Output signal type: auto

Sample time (-1 for inherited):

-1

Parameters and Dialog Box

Function

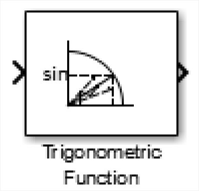
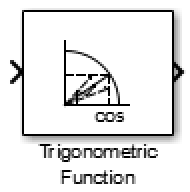
Specify the trigonometric function. The name of the function on the block icon changes to match your selection.

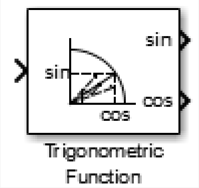
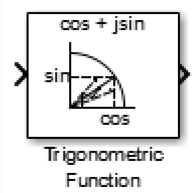
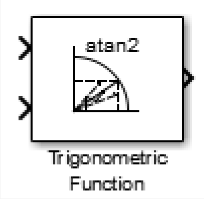
Approximation method

Specify the type of approximation for computing output. This parameter appears only when you set **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`.

Approximation Method	Data Types Supported	When to Use This Method
`None` (default)	Floating point	You want to use the default Taylor series algorithm.
`CORDIC`	Floating point and fixed point	You want a fast, approximate calculation.

If you select `CORDIC` and enlarge the block from the default size, the block icon changes:

Function	Block Icon
`sin`	 sin Trigonometric Function
`cos`	 cos Trigonometric Function

Function	Block Icon
`sincos`	 sin sin cos cos Trigonometric Function
`cos + jsin`	 cos + jsin sin cos Trigonometric Function
`atan2`	 atan2 Trigonometric Function

Number of iterations

Specify the number of iterations to perform the CORDIC algorithm. The default value is 11.

- When the block input uses a floating-point data type, the number of iterations can be a positive integer.
- When the block input is a fixed-point data type, the number of iterations cannot exceed the word length.

  For example, if the block input is `fixdt(1,16,15)`, the word length is 16. In this case, the number of iterations cannot exceed 16.

This parameter appears when both of the following conditions hold:

- You set **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`.
- You set **Approximation method** to `CORDIC`.

Output signal type

Specify the output signal type of the Trigonometric Function block as `auto`, `real`, or `complex`.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
Any selection for the **Function** parameter	real	real	real	complex
	complex	complex	error	complex

Note When **Function** is `atan2`, complex input signals are not supported for simulation or code generation.

Setting **Approximation method** to `CORDIC` disables this parameter.

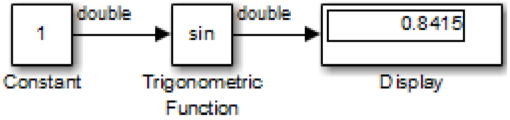
Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the Simulink documentation for more information.

Examples

sin Function with Floating-Point Input

Suppose that you have the following model:


1
double
sin
double
0.8415
Constant
Trigonometric Function
Display

The key block parameters for the Constant block are:

Parameter	Setting
Constant value	`1`
Output data type	`Inherit: Inherit from 'Constant value'`

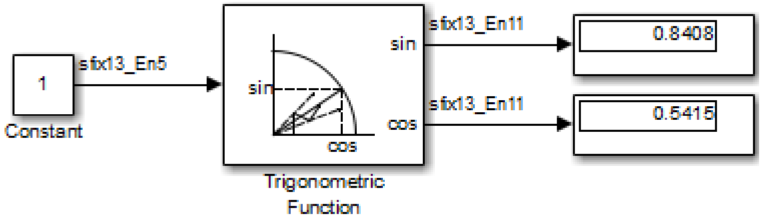
The block parameters for the Trigonometric Function block are:

Parameter	Setting
Function	`sin`
Approximation method	`None`
Output signal type	`auto`
Sample time	`-1`

The output type of the Trigonometric Function block is the same as the input because the input type is floating point and **Approximation method** is `None`.

sincos Function with Fixed-Point Input

Suppose that you have the following model:


1
Constant
sfix13_En5
sin
cos
sin
cos
Trigonometric Function
sfix13_En11
sfix13_En11
0.8408
0.5415

The key block parameters for the Constant block are:

Parameter	Setting
Constant value	1 This value must fall within the range [–2π, 2π) because the Trigonometric Function block uses the CORDIC algorithm and the block input uses a signed fixed-point type.
Output data type	`fixdt(1,13,5)`

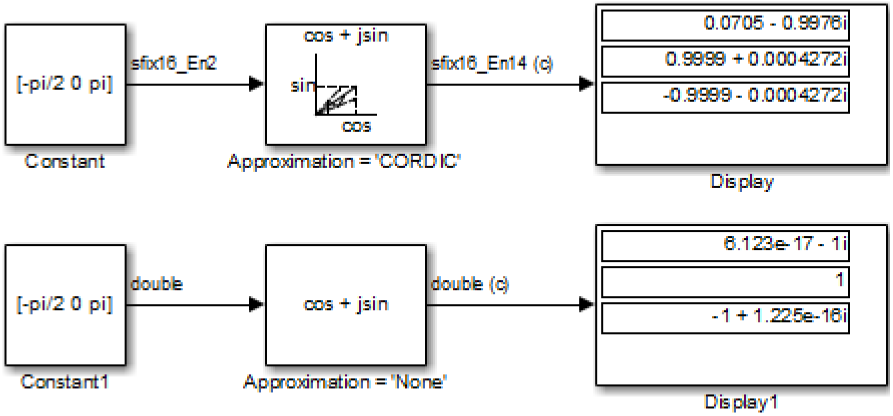
The block parameters for the Trigonometric Function block are:

Parameter	Setting
Function	`sincos`
Approximation method	`CORDIC`
Number of iterations	`11`
Sample time	`-1`

The output type of the Trigonometric Function block is `fixdt(1,13,11)` because the input type is fixed point and **Approximation method** is `CORDIC`. The output fraction length equals the input word length – 2.

Block Behavior for Complex Exponential Output

The following model compares the complex exponential output for the two different approximation methods:


[-pi/2 0 pi]
Constant
sfix16_En2
cos + jsin
sin
cos
Approximation = 'CORDIC'
sfix16_En14 (c)
0.0705 - 0.9976i
0.9999 + 0.0004272i
-0.9999 - 0.0004272i
Display
[-pi/2 0 pi]
Constant1
double
cos + jsin
Approximation = 'None'
double (c)
6.123e-17 - 1i
1
-1 + 1.225e-16i
Display1

The key block parameters for the Constant blocks are:

Block	Parameter	Setting
`Constant`	**Constant value**	`[-pi/2 0 pi]`
	Output data type	`fixdt(1,16,2)`
`Constant1`	**Constant value**	`[-pi/2 0 pi]`
	Output data type	`double`

The block parameters for the Trigonometric Function blocks are:

Block	Parameter	Setting
`Approximation = 'CORDIC'`	**Function**	`cos + jsin`
	Approximation method	`CORDIC`
	Number of iterations	`11`
	Sample time	`-1`
`Approximation = 'None'`	**Function**	`cos + jsin`
	Approximation method	`None`
	Sample time	`-1`

When the **Approximation method** is `CORDIC`, the input data type can be fixed point, in this case: `fixdt(1,16,2)`. The output data type is `fixdt(1,16,14)` because the output fraction length equals the input word length – 2.

When the **Approximation method** is `None`, the input data type must be floating point. The output data type is the same as the input.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	Yes, of the input when the function requires two inputs
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

References

[1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

See Also

Math Function, Sqrt

Unary Minus

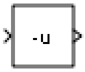
Purpose

Negate input

Library

Math Operations

Description



The Unary Minus block negates the input.

For signed-integer data types, the unary minus of the most negative value is not representable by the data type. In this case, the **Saturate on integer overflow** check box controls the behavior of the block:

If you...	The block...	And...
Select this check box	Saturates to the most positive value of the integer data type	• 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.

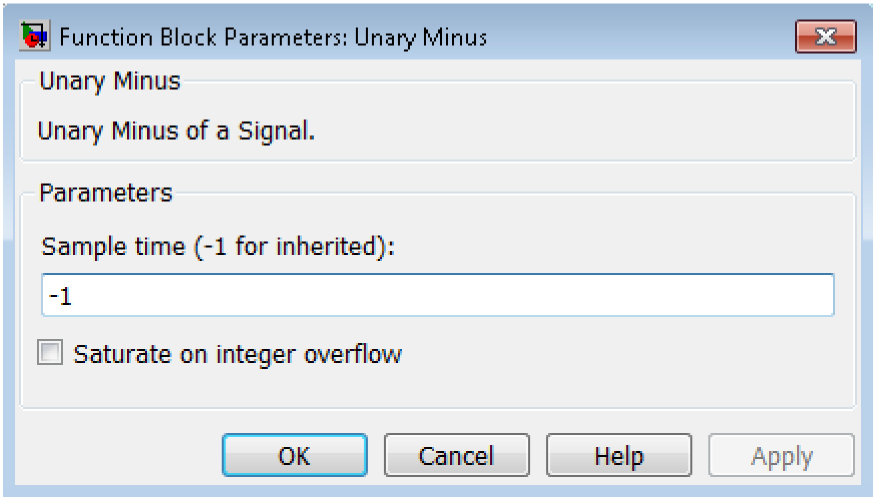
Data Type Support

The Unary Minus block accepts and outputs signals of the following data types:

- Floating point
- Signed integer
- Fixed point

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



Unary Minus of a Signal.

Sample time (-1 for inherited):

-1

☐ Saturate on integer overflow

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see "Specify Sample Time" in the Simulink documentation.

Saturate on integer overflow

Select to have integer overflows saturate. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	No
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of input or initial conditions
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

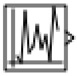
Generate uniformly distributed random numbers

Library

Sources

Description

The Uniform Random Number block generates uniformly distributed random numbers over an interval that you specify. To generate normally distributed random numbers, use the Random Number block.



You can generate a repeatable sequence using any Uniform Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts.

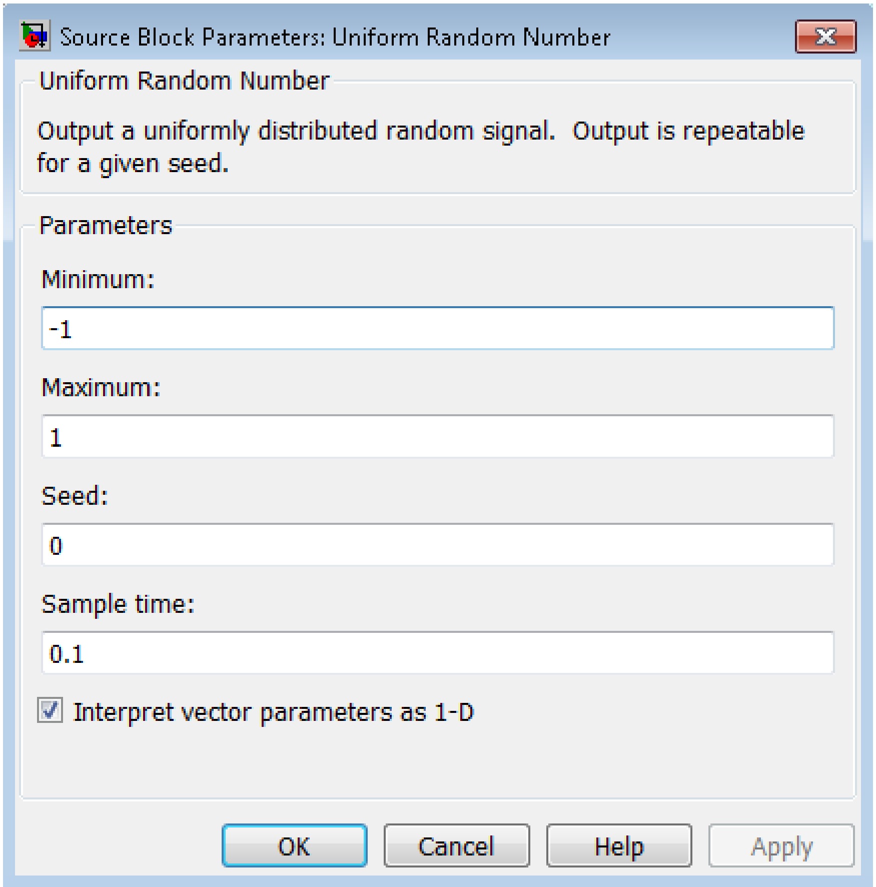
Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the **Interpret vector parameters as 1-D** check box and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensionality as the parameters.

Data Type Support

The Uniform Random Number block accepts and outputs a real signal of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Source Block Parameters: Uniform Random Number

Uniform Random Number

Output a uniformly distributed random signal. Output is repeatable for a given seed.

Parameters

Minimum:

-1

Maximum:

1

Seed:

0

Sample time:

0.1

☑ Interpret vector parameters as 1-D

OK
Cancel
Help
Apply

Parameters and Dialog Box

Minimum

Specify the minimum of the interval. The default is `-1`.

Maximum

Specify the maximum of the interval. The default is `1`.

Seed

Specify the starting seed for the random number generator. The default is `0`.

The seed must be 0 or a positive integer. Output is repeatable for a given seed.

Sample time

Specify the time interval between samples. The default is `0.1`. See “Specify Sample Time” in the Simulink documentation for more information.

Interpret vector parameters as 1-D

If you select this check box and the other parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. Otherwise, the block outputs a signal of the same dimensionality as the other parameters. For more information, see “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Characteristics

Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of parameters
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Random Number

Unit Delay

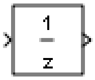
Purpose

Delay signal one sample period

Library

Discrete

Description



The Unit Delay block holds and delays its input by the sample period you specify. 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**.

When the Unit Delay block inherits a continuous sample time, the block is analogous to the Memory block.

Tip Do not use the Unit Delay block to create a slow-to-fast transition between blocks operating at different sample rates. Instead, use the Rate Transition block.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
Unit Delay	Implement a delay using a discrete sample time that you specify. Ideally, the block accepts and outputs signals with a discrete sample time.	• `sldemo_enginewc` (Compression subsystem)
Memory	Implement a delay by one integration time step. Ideally, the block accepts and outputs signals where the sample time is continuous or fixed in minor time step. For more information, see "Types of Sample Time" in the Simulink documentation.	• `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`

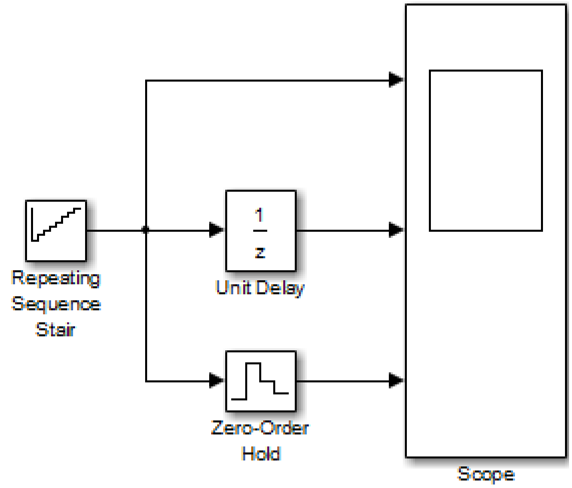
Overview of Block Capabilities

Capability	Block		
	Unit Delay	**Memory**	**Zero-Order Hold**
Specification of initial condition	Yes	Yes	No, because the block output at time t = 0 must match the input value.
Specification of sample time	Yes	No, because the block can only inherit sample time (from the driving block or the solver used for the entire model).	Yes
Support for frame-based signals	Yes	No	Yes
Support for state logging	Yes	No	No

Effect of Solver Specification on Block Output

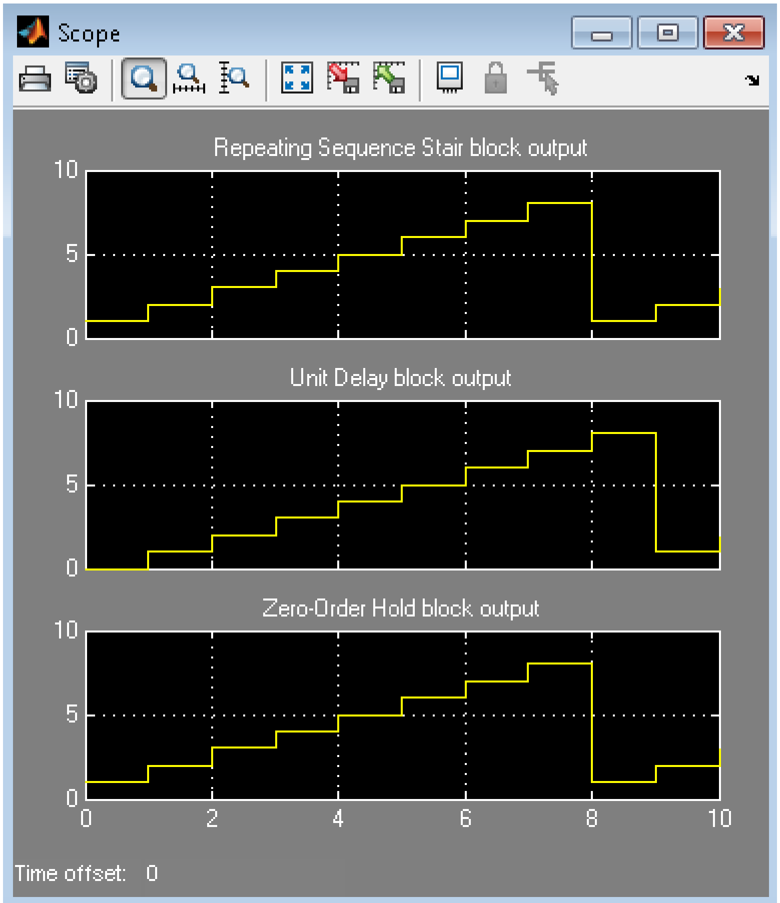
When you specify a discrete sample time in the dialog box for a Unit Delay or Zero-Order Hold block, the block output can differ depending on the solver specification for the model.

Suppose that you have a model with Unit Delay and Zero-Order Hold blocks, which both use a discrete sample time of 1:

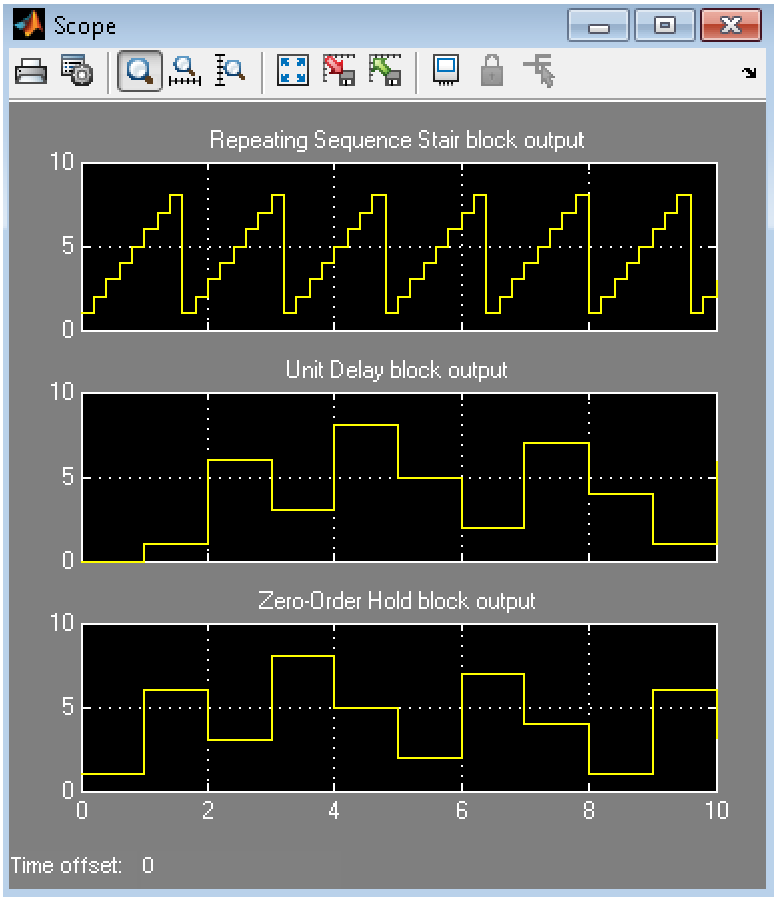

1
z
Repeating Sequence Stair
Unit Delay
Zero-Order Hold
Scope

The Repeating Sequence Stair block uses a continuous sample time of 0 to provide input signals to the Unit Delay and Zero-Order Hold blocks.

If the model uses a fixed-step solver with a step size of 1, the scope shows the following simulation results:


Scope
Repeating Sequence Stair block output
Unit Delay block output
Zero-Order Hold block output
10
5
0
0
2
4
6
8
10
Time offset: 0

If the model uses a variable-step solver, the scope shows the following simulation results:


Scope
Repeating Sequence Stair block output
Unit Delay block output
Zero-Order Hold block output
10
5
0
0
2
4
6
8
10
Time offset: 0

The Zero-Order Hold block takes the input value of the Repeating Sequence Stair block at t = 0, 1, 2, ... , 9 and holds each input value for a sample period (1 second). The Unit Delay block applies the same 1-second hold to each input value of the Repeating Sequence Stair block, but also delays each value by a sample period. The **Initial conditions**

parameter specifies the output for the Unit Delay block during the first sample period. For more information about sample time, see "What Is Sample Time?" and "Specify Sample Time".

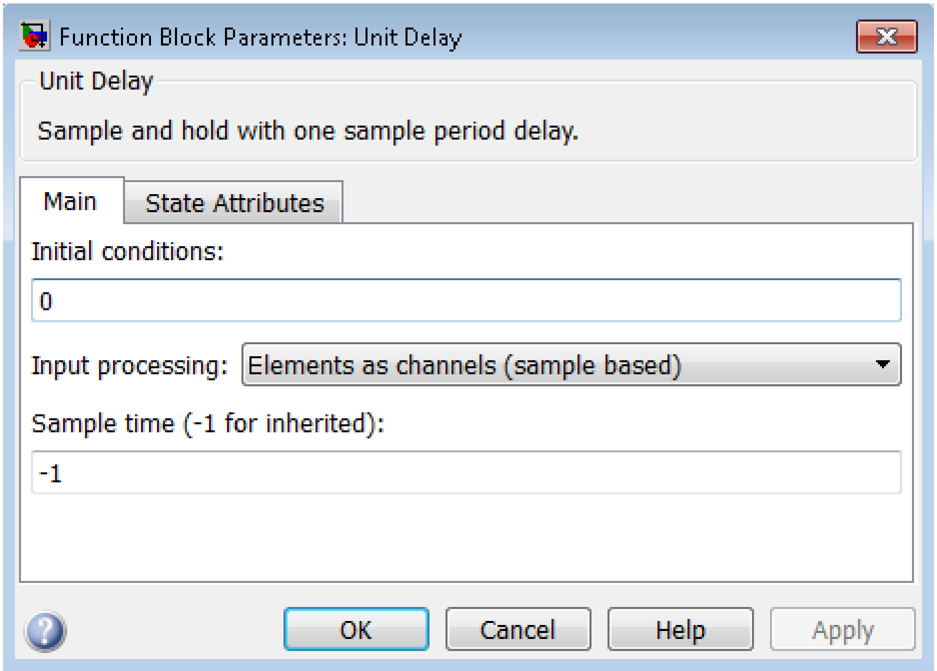
Solver specification for a model also affects the behavior of the Memory block. For details, see "Examples of Memory Block Usage" on page 1-952.

Data Type Support

The Unit Delay block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. If the data type of the input signal is user-defined, the initial condition must be zero.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box


Function Block Parameters: Unit Delay

Unit Delay

Sample and hold with one sample period delay.

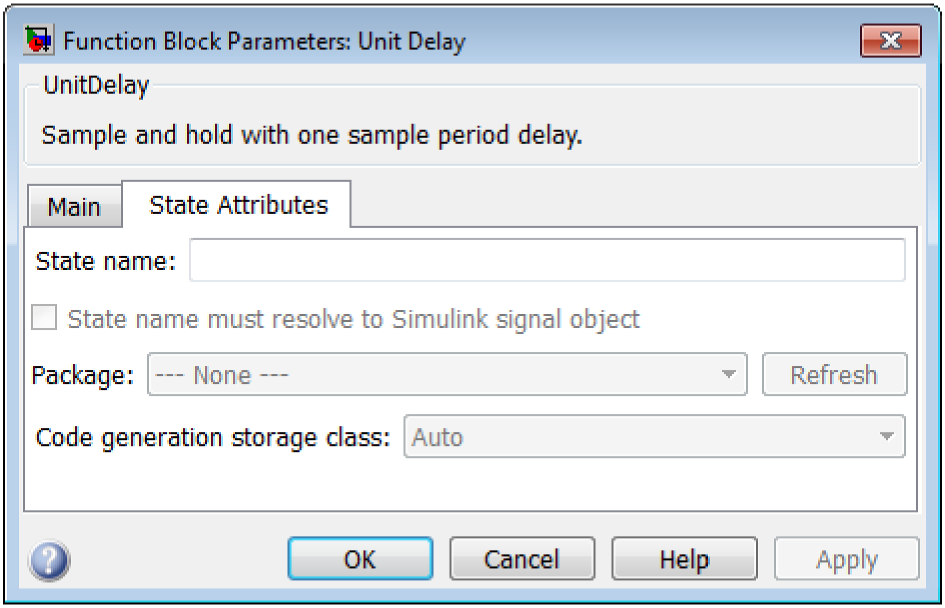
Main
State Attributes

Initial conditions: 0

Input processing: Elements as channels (sample based)

Sample time (-1 for inherited): -1

OK
Cancel
Help
Apply


Function Block Parameters: Unit Delay

UnitDelay

Sample and hold with one sample period delay.

Main
State Attributes

State name:

☐ State name must resolve to Simulink signal object

Package: --- None ---

Refresh

Code generation storage class: Auto

OK
Cancel
Help
Apply

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 “States” in the Simulink Coder documentation.

Initial condition

Specify the output of the simulation for the first sampling period, during which the output of the Unit Delay block is otherwise undefined.

Settings

Default: `0`

The **Initial conditions** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Input processing

Specify whether the Unit Delay block performs sample- or frame-based processing.

Settings

Default: `Elements as channels (sample based)`

`Elements as channels (sample based)`
Treat each element of the input as a separate channel (sample-based processing).

`Columns as channels (frame based)`
Treat each column of the input as a separate channel (frame-based processing).

`Inherited`
Sets the block to inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink® will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal 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" in the DSP System Toolbox documentation.

Dependency

Frame-based processing requires a DSP System Toolbox license.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Settings

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

State name

Use this parameter to assign a unique name to each state.

Settings

Default: `' '`

- If left blank, no name is assigned.

Tips

- 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.

Dependency

This parameter enables **State name must resolve to Simulink signal object** when you click the **Apply** button.

For more information, see “States” in the Simulink Coder documentation.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Settings

Default: Off

☑ On
Require that state name resolve to Simulink signal object.

☐ Off
Do not require that state name resolve to Simulink signal object.

Dependencies

State name enables this parameter.

Selecting this check box disables **Code generation storage class**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Package

Select a package that defines the custom storage class you want to apply.

Settings

Default: `---None---`

`---None---`
: Sets internal storage class attributes.

`mpt`
: Applies the built-in `mpt` package.

`Simulink`
: Applies the built-in `Simulink` package.

Dependencies

If you have defined any packages of your own, click **Refresh**. This action adds all user-defined packages on your search path to the package list.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Code generation storage class

Select state storage class.

Settings

Default: `Auto`

`Auto`
: Auto is the appropriate storage class for states that you do not need to interface to external code.

`ExportedGlobal`
: State is stored in a global variable

`ImportedExtern`
: *`model`*`_private.h` declares the state as an extern variable.

`ImportedExternPointer`
: *`model`*`_private.h` declares the state as an extern pointer.

Dependencies

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“State Storage Classes” in the Simulink Coder documentation.

Code generation storage class (when Package is selected)

Select custom storage class for state.

Settings

Default: `Auto`

`Auto`
: Auto is the appropriate storage class for states that you do not need to interface to external code.

`SimulinkGlobal`
: *`model_P`* initializes the state to its corresponding value in the workspace.

`ExportedGlobal`
: State is stored in a global variable

`ImportedExtern`
: *`model`*`_private.h` declares the state as an extern variable.

`ImportedExternPointer`
: *`model`*`_private.h` declares the state as an extern pointer.

`Default`
: A non-editable placeholder storage class is created.

`BitField`
: A `struct` declaration is created that embeds Boolean data.

`Volatile`
: Volatile type qualifier is used in state declaration.

`ExportToFile`
: Header `(.h)` file containing global variable declarations is generated with user-specified name.

`ImportFromFile`
: Predefined header `(.h)` files containing global variable declarations are included.

`FileScope`
A static qualifier is generated in front of the state declaration to make the state visible only to the current file.

`Struct`
A `struct` declaration is created to encapsulate parameter or signal object data.

`StructVolatile`
Volatile type qualifier is used in `struct` declaration.

`GetSet`
Supports specialized function calls to read and write memory.

Dependencies

State name enables this parameter.

The list of valid storage classes differs based on the **Package** selection.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Code generation storage type qualifier**.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

See Also

“State Storage Classes” in the Simulink Coder documentation.

Code generation storage type qualifier

Specify the Simulink Coder storage type qualifier.

Settings

Default: `'  '`

If left blank, no qualifier is assigned.

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Bus Support

The Unit Delay block is a bus-capable block. The input can be a virtual or nonvirtual bus signal subject to the following restrictions:

- **Initial conditions** must be zero, a nonzero scalar, or a finite numeric structure.
- If **Initial conditions** is zero or a structure, and you specify a **State name**, the input cannot be a virtual bus.
- If **Initial conditions** is a nonzero scalar, no **State name** can be specified.

For information about specifying an initial condition structure, see "Specify Initial Conditions for Bus Signals".

All signals in a nonvirtual bus input to a Unit Delay block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See "About Composite Signals" and Bus-Capable Blocks for more information.

Examples

For an example of how to use the Unit Delay block, see the `sldemo_enginewc` model. The Unit Delay block appears in the `Compression` subsystem.

Characteristics

Bus-capable	Yes, with restrictions as noted in “Bus Support” on page 1-1981
Direct Feedthrough	No
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes, of input or initial conditions
States	Yes, inherited from the driving block for nonfixed-point data types
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Memory, Zero-Order Hold

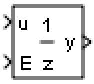
Purpose

Delay signal one sample period, if external enable signal is on

Library

Additional Math & Discrete / Additional Discrete

Description



The Unit Delay Enabled block delays a signal by one sample period when the external enable signal `E` is on. While the enable is off, the block is disabled. It holds the current state at the same value and outputs that value. The enable signal is on when `E` is not 0, and off when `E` is 0.

You specify the block output for the first sampling period with the value of the **Initial condition** parameter.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

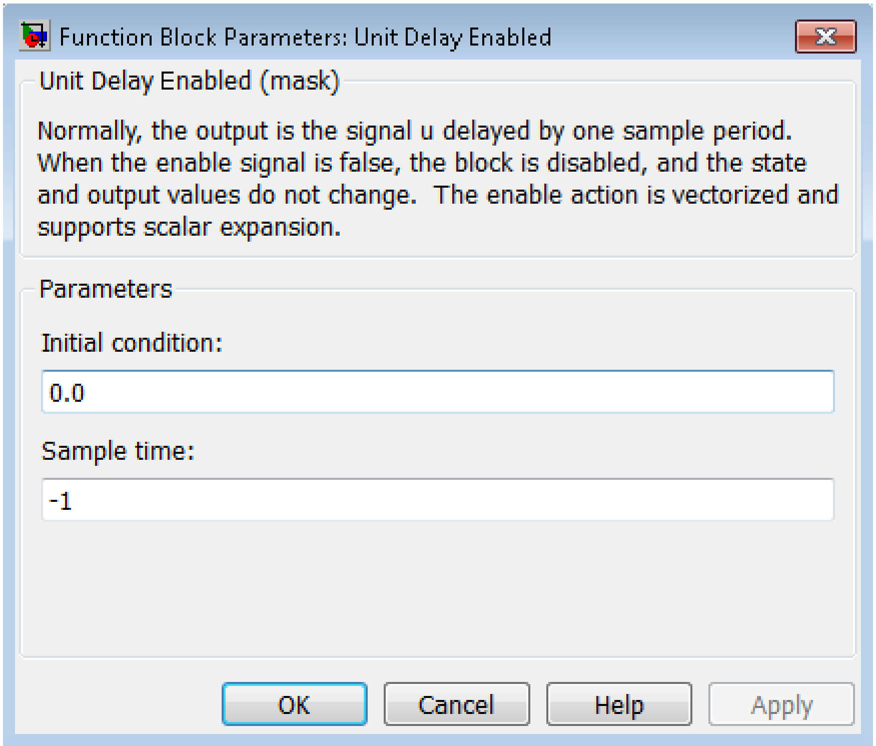
Data Type Support

The Unit Delay Enabled block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The output has the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.


Function Block Parameters: Unit Delay Enabled
Unit Delay Enabled (mask)

Normally, the output is the signal u delayed by one sample period. When the enable signal is false, the block is disabled, and the state and output values do not change. The enable action is vectorized and supports scalar expansion.

Parameters

Initial condition: 0.0

Sample time: -1

OK
Cancel
Help
Apply

Parameters and Dialog Box

Initial condition
Specify the initial output of the simulation.

Sample time
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	No
Sample Time	Specified in the **Sample time** parameter

Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Unit Delay Enabled External IC

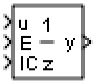
Purpose

Delay signal one sample period, if external enable signal is on, with external initial condition

Library

Additional Math & Discrete / Additional Discrete

Description



The Unit Delay Enabled External IC block delays a signal by one sample period when the enable signal `E` is on. While the enable is off, the block holds the current state at the same value and outputs that value. The enable `E` is on when `E` is not 0, and off when `E` is 0.

The initial condition of this block is given by the signal `IC`.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means the block inherits the **Sample time**.

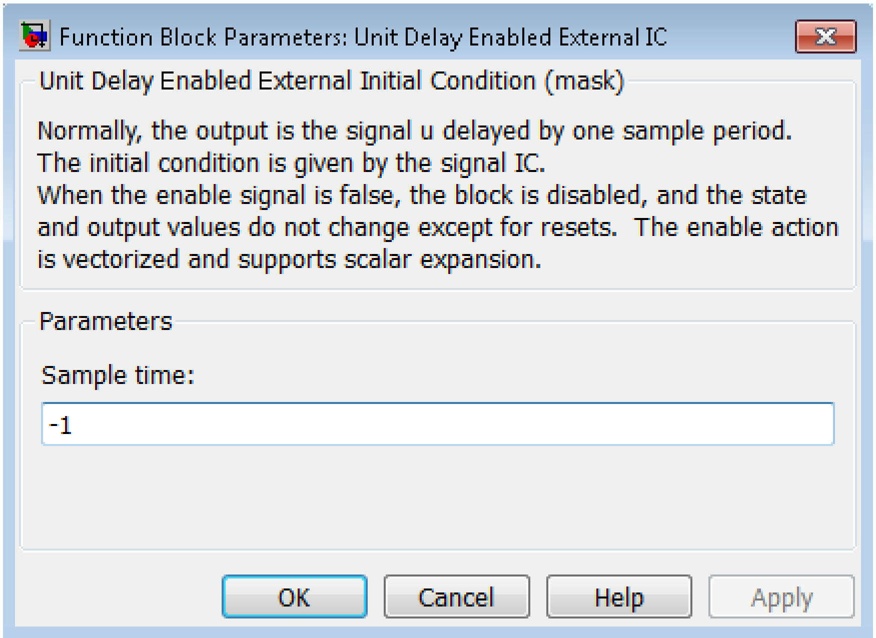
Data Type Support

The Unit Delay Enabled External IC block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The data types of the inputs `u` and `IC` must be the same. The output has the same data type as `u` and `IC`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.


Function Block Parameters: Unit Delay Enabled External IC
Unit Delay Enabled External Initial Condition (mask)

Normally, the output is the signal u delayed by one sample period.
The initial condition is given by the signal IC.
When the enable signal is false, the block is disabled, and the state and output values do not change except for resets. The enable action is vectorized and supports scalar expansion.

Parameters

Sample time: -1

OK
Cancel
Help
Apply

Parameters and Dialog Box

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes, of the reset input port No, of the enable input port Yes, of the external `IC` port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

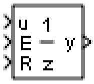
See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Purpose Delay signal one sample period, if external enable signal is on, with external Boolean reset

Library Additional Math & Discrete / Additional Discrete

Description The Unit Delay Enabled Resettable block combines the features of the Unit Delay Enabled and Unit Delay Resettable blocks.



The block can reset its state based on an external reset signal `R`. When the enable signal `E` is on and the reset signal `R` is false, the block outputs the input signal delayed by one sample period.

When the enable signal `E` is on and the reset signal `R` is true, the block resets the current state to the initial condition, specified by the **Initial condition** parameter, and outputs that state delayed by one sample period.

When the enable signal is off, the block is disabled, and the state and output do not change except for resets. The enable signal is on when `E` is not 0, and off when `E` is 0.

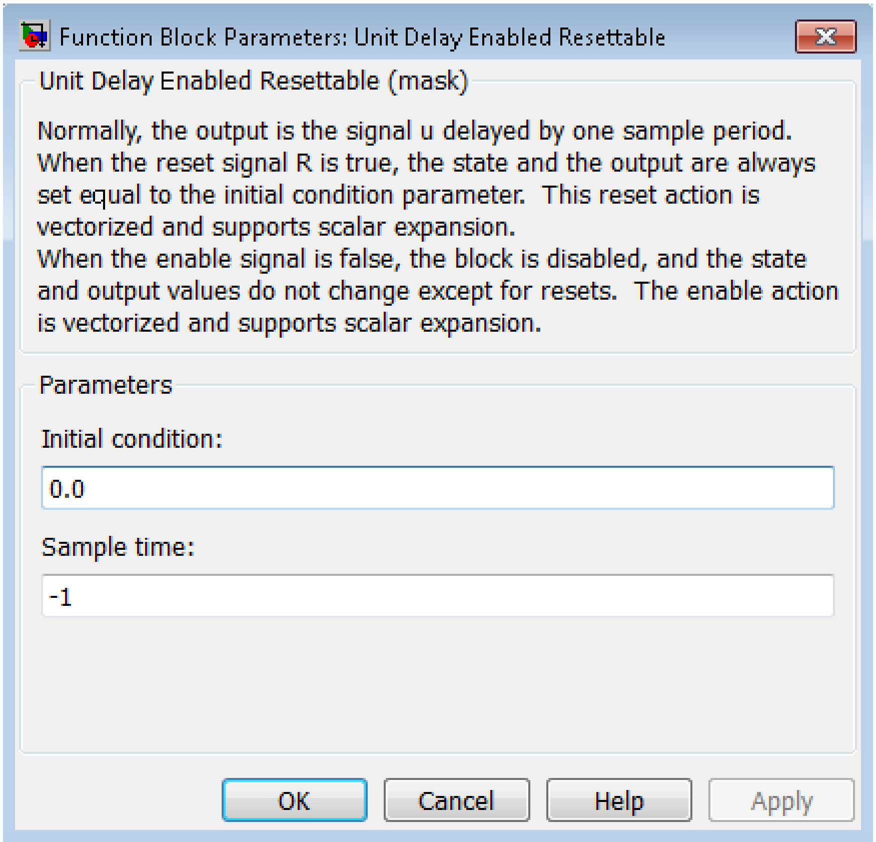
You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

Data Type Support The Unit Delay Enabled Resettable block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The output has the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Unit Delay Enabled Resettable (mask)

Normally, the output is the signal u delayed by one sample period. When the reset signal R is true, the state and the output are always set equal to the initial condition parameter. This reset action is vectorized and supports scalar expansion.
When the enable signal is false, the block is disabled, and the state and output values do not change except for resets. The enable action is vectorized and supports scalar expansion.

Parameters

Initial condition: 0.0

Sample time: -1

Parameters and Dialog Box

Initial condition

Specify the initial output of the simulation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	No, of the input port No, of the enable port Yes, of the reset port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Unit Delay Enabled Resettable External IC

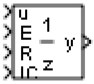
Purpose

Delay signal one sample period, if external enable signal is on, with external Boolean reset and initial condition

Library

Additional Math & Discrete / Additional Discrete

Description



The Unit Delay Enabled Resettable External IC block combines the features of the Unit Delay Enabled, Unit Delay External IC, and Unit Delay Resettable blocks.

The block can reset its state based on an external reset signal `R`. When the enable signal `E` is on and the reset signal `R` is false, the block outputs the input signal delayed by one sample period.

When the enable signal `E` is on and the reset signal `R` is true, the block resets the current state to the initial condition given by the signal `IC`, and outputs that state delayed by one sample period.

When the enable signal is off, the block is disabled, and the state and output do not change except for resets. The enable signal is on when `E` is not 0, and off when `E` is 0.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

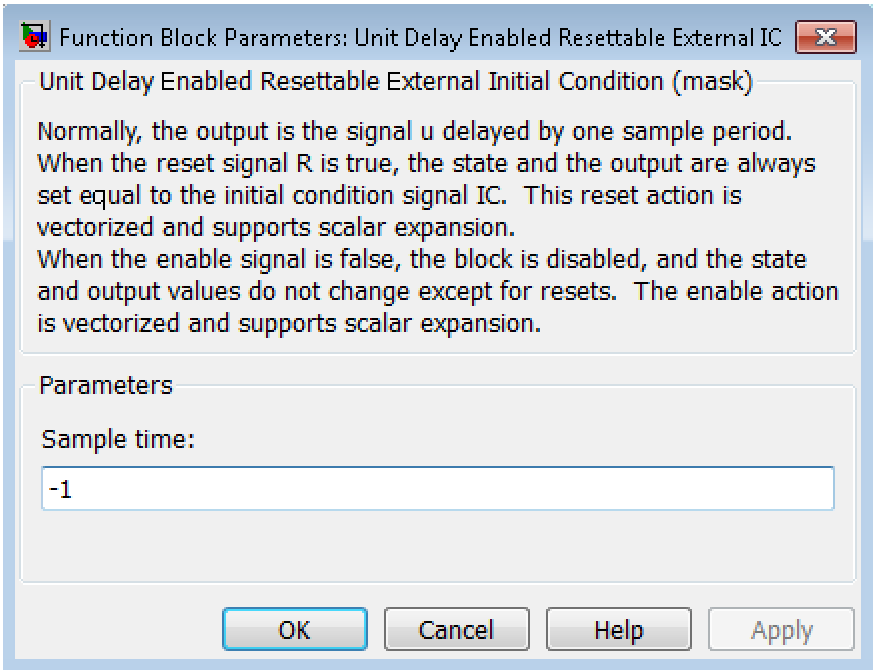
Data Type Support

The Unit Delay Enabled Resettable External IC block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The data types of the inputs `u` and `IC` must be the same. The output has the same data type as `u` and `IC`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Function Block Parameters: Unit Delay Enabled Resettable External IC

Unit Delay Enabled Resettable External Initial Condition (mask)

Normally, the output is the signal u delayed by one sample period. When the reset signal R is true, the state and the output are always set equal to the initial condition signal IC. This reset action is vectorized and supports scalar expansion.
When the enable signal is false, the block is disabled, and the state and output values do not change except for resets. The enable action is vectorized and supports scalar expansion.

Parameters

Sample time: -1

Parameters and Dialog Box

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	No, of the input port No, of the enable port Yes, of the enable port Yes, of the external `IC` port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Unit Delay External IC

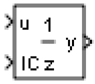
Purpose

Delay signal one sample period, with external initial condition

Library

Additional Math & Discrete / Additional Discrete

Description



The Unit Delay External IC block delays its input by one sample period. This block is equivalent to the z^{-1} discrete-time operator. The block accepts one input and generates one output, both of which can be scalar or vector. If the input is a vector, all elements of the vector are delayed by the same sample period.

The block's output for the first sample period is equal to the signal `IC`.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

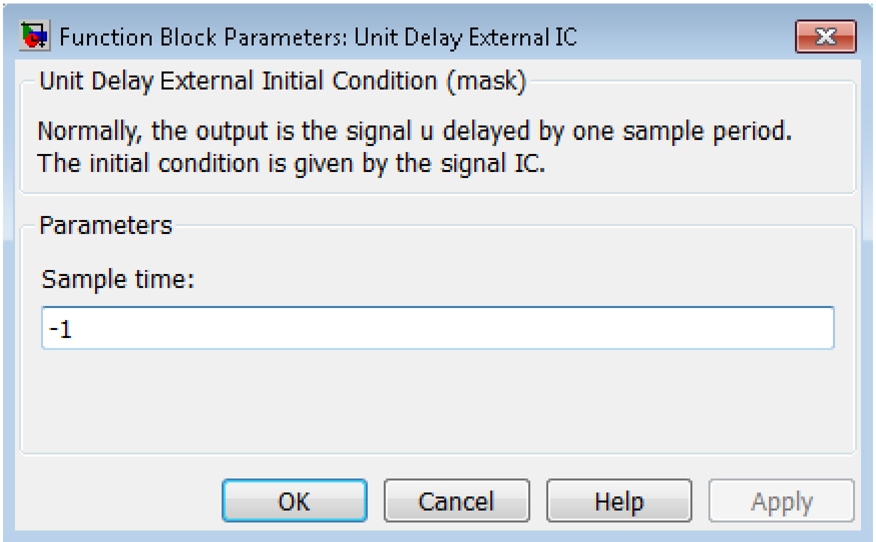
Data Type Support

The Unit Delay External IC block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The data types of the inputs `u` and `IC` must be the same. The output has the same data type as `u` and `IC`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Function Block Parameters: Unit Delay External IC

Unit Delay External Initial Condition (mask)

Normally, the output is the signal u delayed by one sample period.
The initial condition is given by the signal IC.

Parameters

Sample time:

-1

Parameters and Dialog Box

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	No, of the input port Yes, of the external `IC` port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Purpose

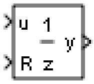
Delay signal one sample period, with external Boolean reset

Library

Additional Math & Discrete / Additional Discrete

Description

The Unit Delay Resettable block delays a signal one sample period.



The block can reset both its state and output based on an external reset signal `R`. The block has two input ports, one for the input signal `u` and the other for the external reset signal `R`.

At the start of simulation, the block's **Initial condition** parameter determines its initial output. During simulation, when the reset signal is false, the block outputs the input signal delayed by one time step. When the reset signal is true, the block resets the current state and its output to the **Initial condition**.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

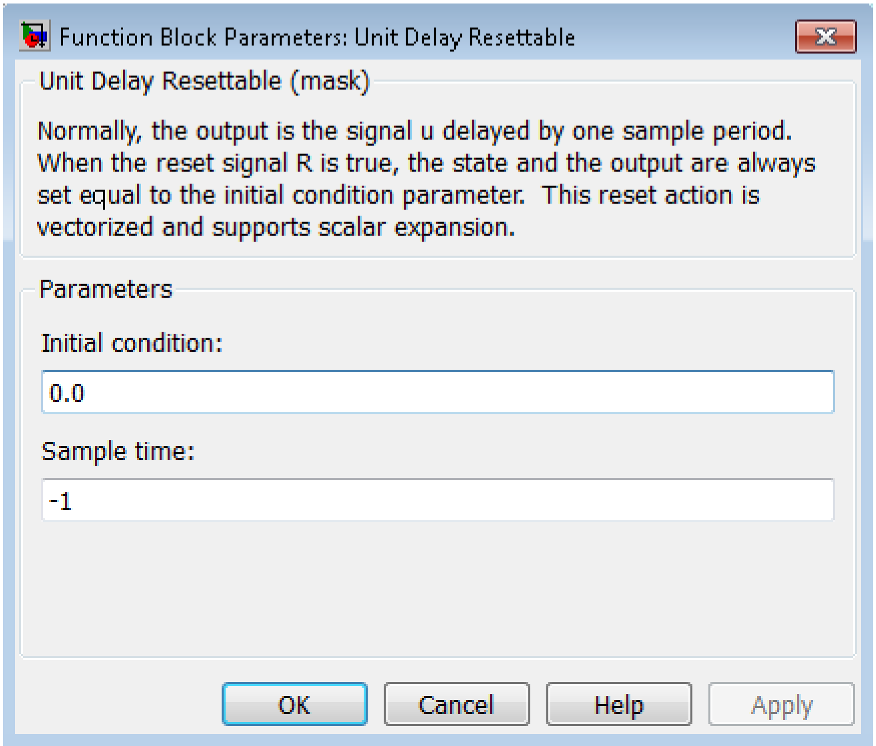
Data Type Support

The Unit Delay Resettable block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The output has the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.


Function Block Parameters: Unit Delay Resettable
OK
Cancel
Help
Apply

Unit Delay Resettable (mask)

Normally, the output is the signal u delayed by one sample period. When the reset signal R is true, the state and the output are always set equal to the initial condition parameter. This reset action is vectorized and supports scalar expansion.

Parameters

Initial condition: 0.0

Sample time: -1

Parameters and Dialog Box

Initial condition
Specify the initial output of the simulation.

Sample time
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Direct Feedthrough	No, of the input port Yes, of the reset port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

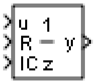
Purpose

Delay signal one sample period, with external Boolean reset and initial condition

Library

Additional Math & Discrete / Additional Discrete

Description



The Unit Delay Resettable External IC block delays a signal one sample period.

The block can reset its state based on an external reset signal `R`. The block has two input ports, one for the input signal `u` and the other for the reset signal `R`. When the reset signal is false, the block outputs the input signal delayed by one time step. When the reset signal is true, the block resets the current state to the initial condition given by the signal `IC` and outputs that state delayed by one time step.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

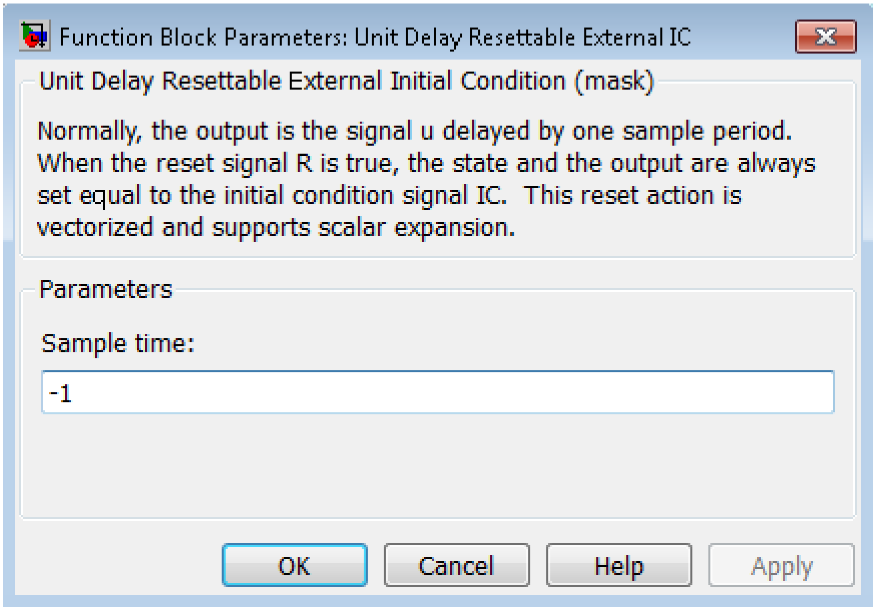
Data Type Support

The Unit Delay Resettable External IC block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The data types of the inputs `u` and `IC` must be the same. The output has the same data type as `u` and `IC`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.


Function Block Parameters: Unit Delay Resettable External IC
OK
Cancel
Help
Apply

Unit Delay Resettable External Initial Condition (mask)

Normally, the output is the signal u delayed by one sample period. When the reset signal R is true, the state and the output are always set equal to the initial condition signal IC. This reset action is vectorized and supports scalar expansion.

Parameters

Sample time:

-1

Parameters and Dialog Box

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	No, of the input port Yes, of the reset port Yes, of the external `IC` port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Unit Delay With Preview Enabled

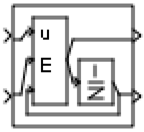
Purpose

Output signal and signal delayed by one sample period, if external enable signal is on

Library

Additional Math & Discrete / Additional Discrete

Description



The Unit Delay With Preview Enabled block supports calculations that have feedback and depend on the current input.

The block has two input ports: one for the input signal `u` and one for the external enable signal `E`.

When the enable signal `E` is on, the first port outputs the signal and the second port outputs the signal delayed by one sample period. When the enable signal `E` is off, the block is disabled, and the state and output values do not change, except during resets.

The enable signal is on when `E` is not 0, and off when `E` is 0. This enable action is vectorized and supports scalar expansion.

Having two outputs is useful for implementing recursive calculations where the result includes the most recent inputs. The second output can feed back into calculations of the block's inputs without causing an algebraic loop. Meanwhile, the first output shows the most up-to-date calculations.

You specify the block output for the first sampling period with the value of the **Initial condition** parameter.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

Data Type Support

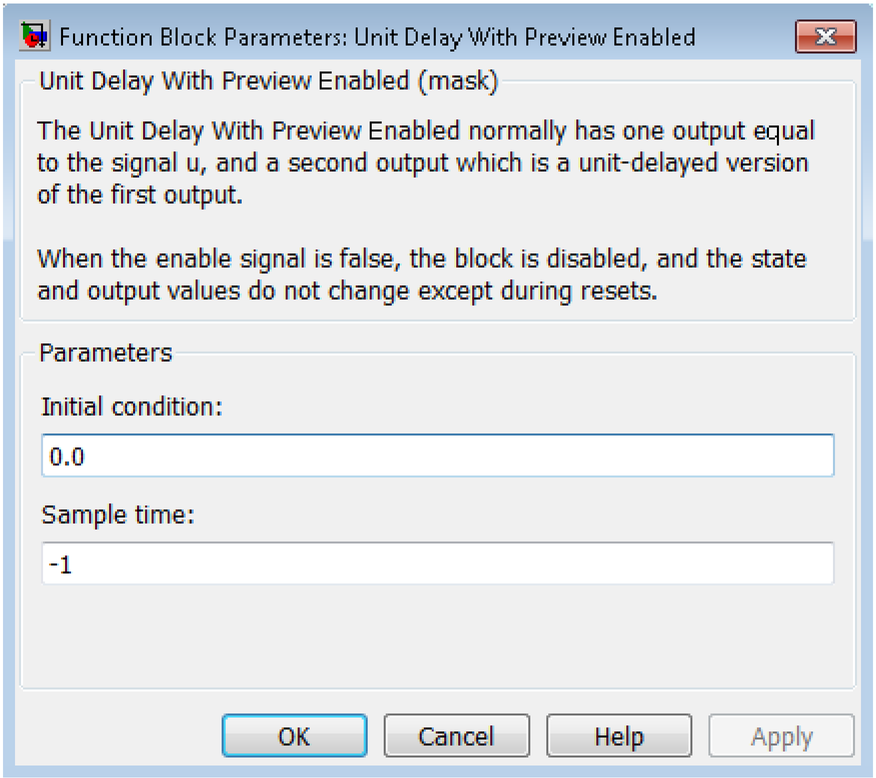
The Unit Delay With Preview Enabled block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

- Boolean
- Enumerated

The outputs have the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Unit Delay With Preview Enabled
OK
Cancel
Help
Apply

Unit Delay With Preview Enabled (mask)

The Unit Delay With Preview Enabled normally has one output equal to the signal u, and a second output which is a unit-delayed version of the first output.

When the enable signal is false, the block is disabled, and the state and output values do not change except during resets.

Parameters

Initial condition: 0.0

Sample time: -1

Parameters and Dialog Box

Initial condition

Specify the initial output of the simulation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes, to first output port No, to second output port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Unit Delay With Preview Enabled Resettable

Purpose

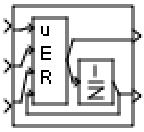
Output signal and signal delayed by one sample period, if external enable signal is on, with external reset

Library

Additional Math & Discrete / Additional Discrete

Description

The Unit Delay With Preview Enabled Resettable block supports calculations that have feedback and depend on the current input. The block can reset its state based on a reset signal `R`.



The block has three input ports: one for the input signal `u`, one for the external enable signal `E`, and one for the external reset signal `R`.

When the enable signal `E` is on and the reset signal `R` is false, the first port outputs the signal and the second port outputs the signal delayed by one sample period.

When the enable signal `E` is on and the reset signal `R` is true, the block resets the current state to the initial condition given by the **Initial condition** parameter. The first output signal is forced to equal the initial condition. The second output signal is not affected until one time step later.

When the enable signal is off, the block is disabled, and the state and output values do not change, except during resets.

The enable signal is on when `E` is not 0, and off when `E` is 0. The enable and reset actions are vectorized and support scalar expansion.

Having two outputs is useful for implementing recursive calculations where the result includes the most recent inputs. The second output can feed back into calculations of the block's inputs without causing an algebraic loop. Meanwhile, the first output shows the most up-to-date calculations.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

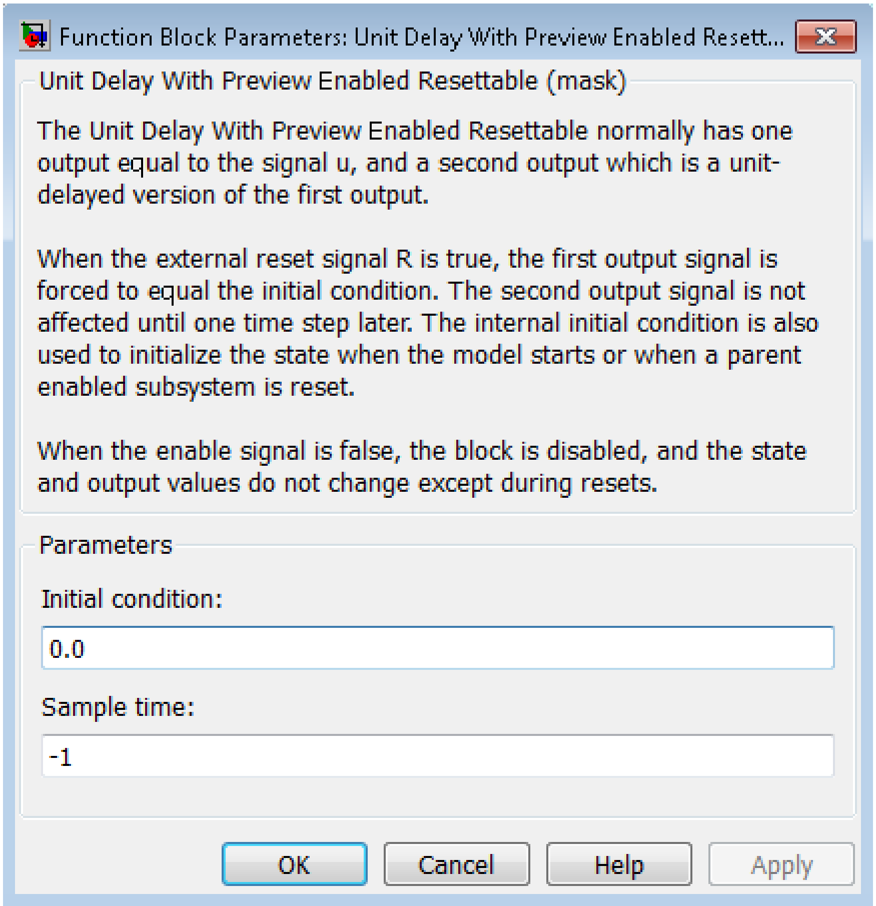
Data Type Support

The Unit Delay With Preview Enabled Resettable block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The outputs have the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Unit Delay With Preview Enabled Resett...

Unit Delay With Preview Enabled Resettable (mask)

The Unit Delay With Preview Enabled Resettable normally has one output equal to the signal u, and a second output which is a unit-delayed version of the first output.

When the external reset signal R is true, the first output signal is forced to equal the initial condition. The second output signal is not affected until one time step later. The internal initial condition is also used to initialize the state when the model starts or when a parent enabled subsystem is reset.

When the enable signal is false, the block is disabled, and the state and output values do not change except during resets.

Parameters

Initial condition: 0.0

Sample time: -1

Parameters and Dialog Box

Initial condition

Specify the initial output of the simulation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes, to first output port No, to second output port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Unit Delay With Preview Enabled Resettable External RV

Purpose

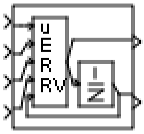
Output signal and signal delayed by one sample period, if external enable signal is on, with external RV reset

Library

Additional Math & Discrete / Additional Discrete

Description

The Unit Delay With Preview Enabled Resettable External RV block supports calculations that have feedback and depend on the current input. The block can reset its state based on a reset signal `R`.



The block has four input ports: one for the input signal `u`, one for the external enable signal `E`, one for the external reset signal `R`, and one for the external reset value `RV`.

When the enable signal `E` is on and the reset signal `R` is false, the first port outputs the signal and the second port outputs the signal delayed by one sample period.

When the enable signal `E` is on and the reset signal `R` is true, the first output signal is forced to equal the reset value `RV`. The second output signal is not affected until one time step later, at which time it is equal to the reset value `RV` at the previous time step. The internal **Initial condition** has a direct effect on the second output only when the model starts or when a parent enabled subsystem is reset.

When the enable signal is off, the block is disabled, and the state and output values do not change, except during resets.

The enable signal is on when `E` is not 0, and off when `E` is 0. The enable and reset actions are vectorized and support scalar expansion.

Having two outputs is useful for implementing recursive calculations where the result includes the most recent inputs. The second output can feed back into calculations of the block's inputs without causing an algebraic loop. Meanwhile, the first output shows the most up-to-date calculations.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

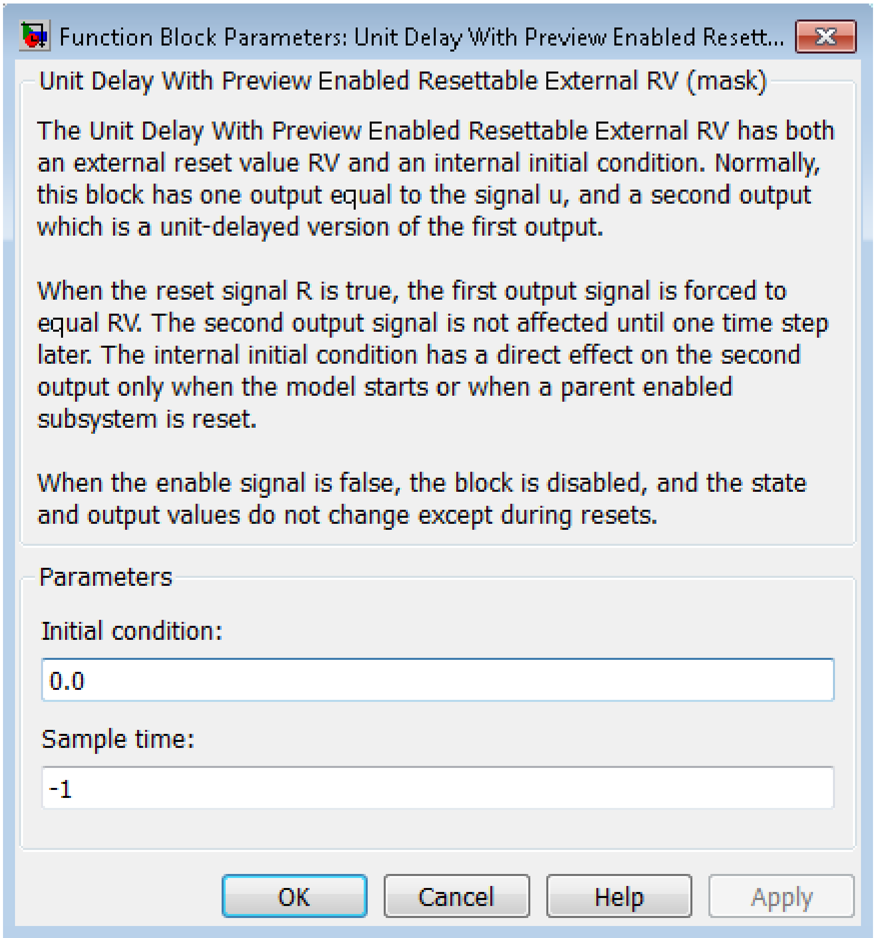
Data Type Support

The Unit Delay With Preview Enabled Resettable External RV block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The outputs have the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Unit Delay With Preview Enabled Resett...

Unit Delay With Preview Enabled Resettable External RV (mask)

The Unit Delay With Preview Enabled Resettable External RV has both an external reset value RV and an internal initial condition. Normally, this block has one output equal to the signal u, and a second output which is a unit-delayed version of the first output.

When the reset signal R is true, the first output signal is forced to equal RV. The second output signal is not affected until one time step later. The internal initial condition has a direct effect on the second output only when the model starts or when a parent enabled subsystem is reset.

When the enable signal is false, the block is disabled, and the state and output values do not change except during resets.

Parameters

Initial condition: 0.0

Sample time: -1

Parameters and Dialog Box

Initial condition

Specify the initial output of the simulation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes, to first output port No, to second output port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Resettable, Unit Delay With Preview Resettable External RV

Purpose

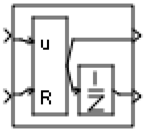
Output signal and signal delayed by one sample period, with external reset

Library

Additional Math & Discrete / Additional Discrete

Description

The Unit Delay With Preview Resettable block supports calculations that have feedback and depend on the current input. The block can reset its state based on a reset signal `R`.



The block has two input ports: one for the input signal `u` and one for the external reset signal `R`.

When the reset signal `R` is false, the first port outputs the signal and the second port outputs the signal delayed by one sample period.

When the reset signal `R` is true, the block resets the current state to the initial condition given by the **Initial condition** parameter. The first output signal is forced to equal the initial condition. The second output signal is not affected until one time step later.

This reset action is vectorized and supports scalar expansion.

Having two outputs is useful for implementing recursive calculations where the result includes the most recent inputs. The second output can feed back into calculations of the block's inputs without causing an algebraic loop. Meanwhile, the first output shows the most up-to-date calculations.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

Data Type Support

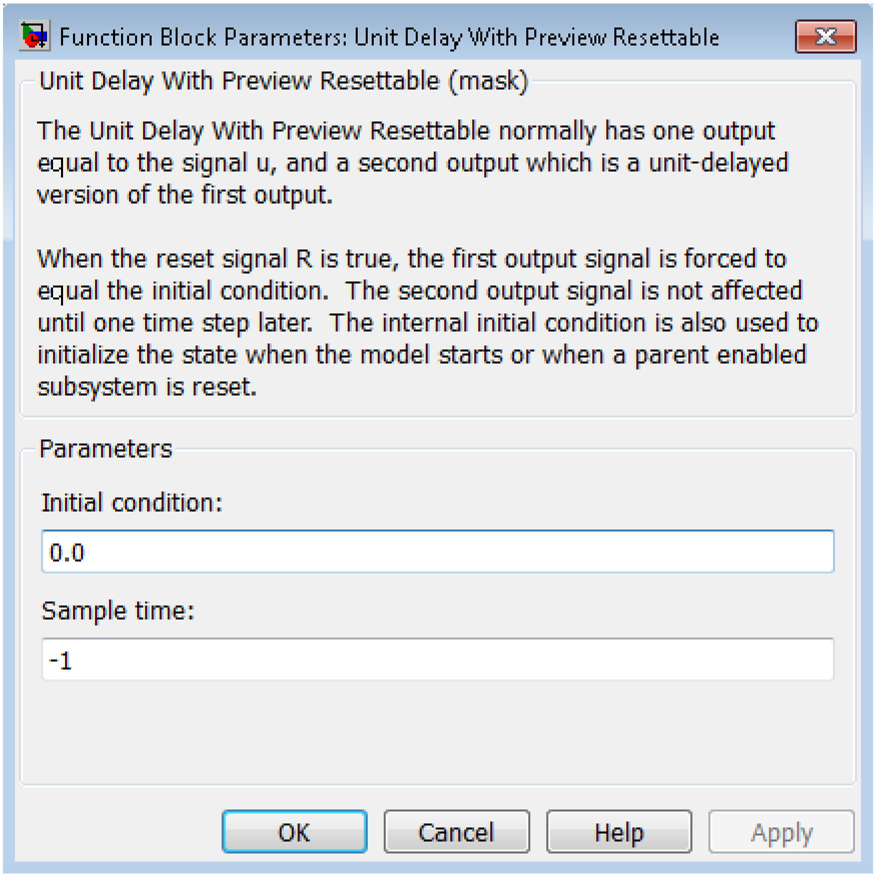
The Unit Delay With Preview Resettable block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

- Boolean
- Enumerated

The outputs have the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Unit Delay With Preview Resettable

Unit Delay With Preview Resettable (mask)

The Unit Delay With Preview Resettable normally has one output equal to the signal u, and a second output which is a unit-delayed version of the first output.

When the reset signal R is true, the first output signal is forced to equal the initial condition. The second output signal is not affected until one time step later. The internal initial condition is also used to initialize the state when the model starts or when a parent enabled subsystem is reset.

Parameters

Initial condition: 0.0

Sample time: -1

Parameters and Dialog Box

Initial condition

Specify the initial output of the simulation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes, to first output port No, to second output port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable External RV

Purpose

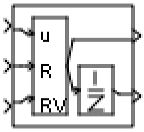
Output signal and signal delayed by one sample period, with external RV reset

Library

Additional Math & Discrete / Additional Discrete

Description

The Unit Delay With Preview Resettable External RV block supports calculations that have feedback and depend on the current input. The block can reset its state based on a reset signal `R`.



The block has three input ports: one for the input signal `u`, one for the external reset signal `R`, and one for the external reset value `RV`.

When the reset signal `R` is false, the first port outputs the signal and the second port outputs the signal delayed by one sample period.

When the reset signal `R` is true, the first output signal is forced to equal the reset value `RV`. The second output signal is not affected until one time step later, at which time it is equal to the reset value `RV` at the previous time step. The internal **Initial condition** has a direct effect on the second output only when the model starts or when a parent enabled subsystem is reset.

This reset action is vectorized and supports scalar expansion.

Having two outputs is useful for implementing recursive calculations where the result includes the most recent inputs. The second output can feed back into calculations of the block's inputs without causing an algebraic loop. Meanwhile, the first output shows the most up-to-date calculations.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means that the block inherits the **Sample time**.

Data Type Support

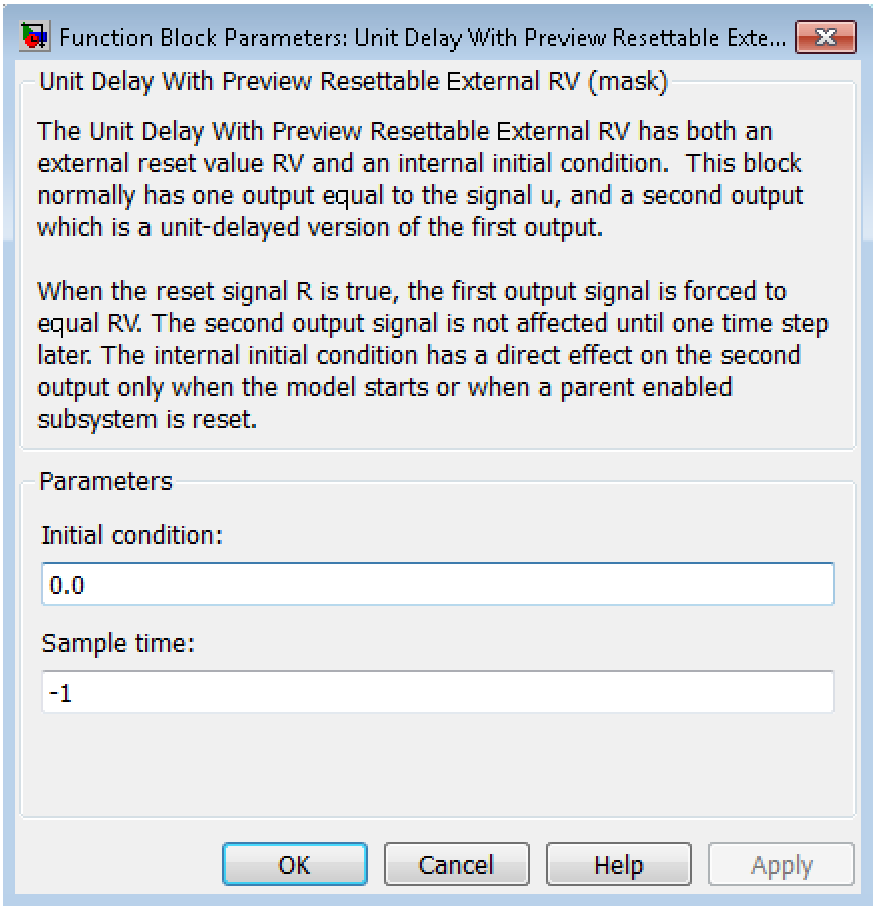
The Unit Delay With Preview Resettable External RV block accepts signals of the following data types:

- Floating point
- Built-in integer

- Fixed point
- Boolean
- Enumerated

The outputs have the same data type as the input `u`. For enumerated signals, the **Initial condition** must be of the same enumerated type as the input `u`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.



Function Block Parameters: Unit Delay With Preview Resettable Exte...
OK
Cancel
Help
Apply

Unit Delay With Preview Resettable External RV (mask)

The Unit Delay With Preview Resettable External RV has both an external reset value RV and an internal initial condition. This block normally has one output equal to the signal u, and a second output which is a unit-delayed version of the first output.

When the reset signal R is true, the first output signal is forced to equal RV. The second output signal is not affected until one time step later. The internal initial condition has a direct effect on the second output only when the model starts or when a parent enabled subsystem is reset.

Parameters

Initial condition: 0.0

Sample time: -1

Parameters and Dialog Box

Initial condition

Specify the initial output of the simulation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Direct Feedthrough	Yes, to first output port No, to second output port
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Yes
Zero-Crossing Detection	No

See Also

Unit Delay, Unit Delay Enabled, Unit Delay Enabled External IC, Unit Delay Enabled Resettable, Unit Delay Enabled Resettable External IC, Unit Delay External IC, Unit Delay Resettable, Unit Delay Resettable External IC, Unit Delay With Preview Enabled, Unit Delay With Preview Enabled Resettable, Unit Delay With Preview Enabled Resettable External RV, Unit Delay With Preview Resettable

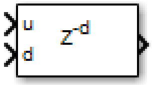
Purpose

Delay input signal by variable sample period

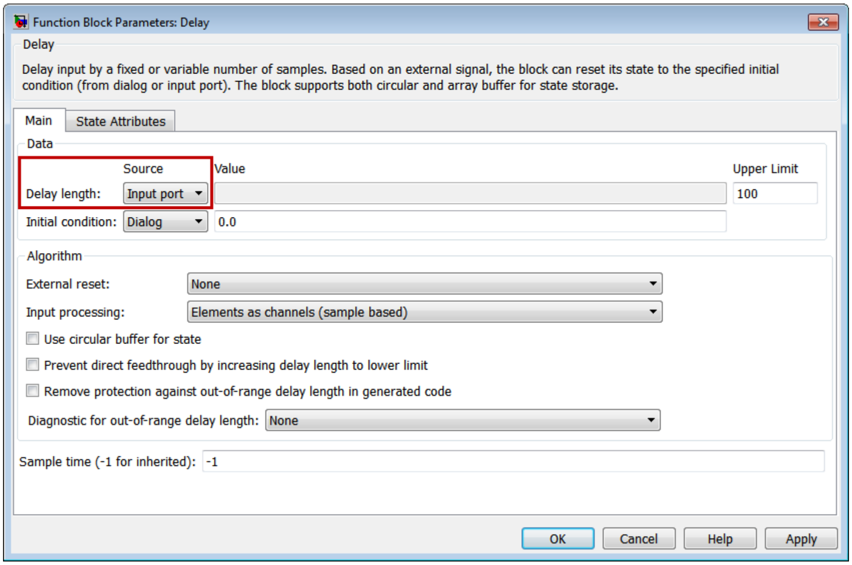
Library

Discrete

Description



The Variable Integer Delay block is a variant of the Delay block that has the source of the delay length set to `Input port`, by default.



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

See Also

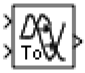
- Delay
- Resettable Delay

Variable Time Delay, Variable Transport Delay

Purpose Delay input by variable amount of time

Library Continuous

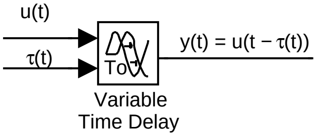
Description The Variable Transport Delay and Variable Time Delay appear as two blocks in the Simulink block library. However, they are the same Simulink block with different settings of a **Select delay type** parameter. Use this parameter to specify the mode in which the block operates.



Variable Time Delay

In this mode, the block has a data input, a time delay input, and a data output. (See "How to Rotate a Block" in the Simulink documentation for a description of the port order for various block orientations.) The output at the current time step equals the value of its data input at a previous time equal to 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.

When you want the output at a time that does not correspond to times of the stored input values and the solver is a continuous solver, the block interpolates linearly between points. If the time delay is smaller than the step size, the block extrapolates an output point from a previous point. For example, consider a fixed-step simulation with a step size of 1 and the current time at t = 5. If the delay is 0.5, the block needs to generate a point at t = 4.5. Because the most recent stored time value is at t = 4, 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

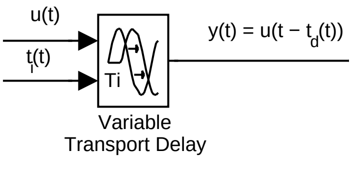
In this mode, the block output at the current time step is equal to the value of its data (top, or left) input at an earlier time equal to the current time minus a transportation delay.

$$y(t) = u(t - t_d(t))$$

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.



For example, suppose you want to use this block to model the flow of a fluid through a pipe where the speed of the flow varies with time. In this case, the time delay input to the block would be

$$t_i(t) = \frac{L}{v_i(t)}$$

where L is the length of the pipe and $v_i(t)$ is the speed of the fluid.

Data Type Support

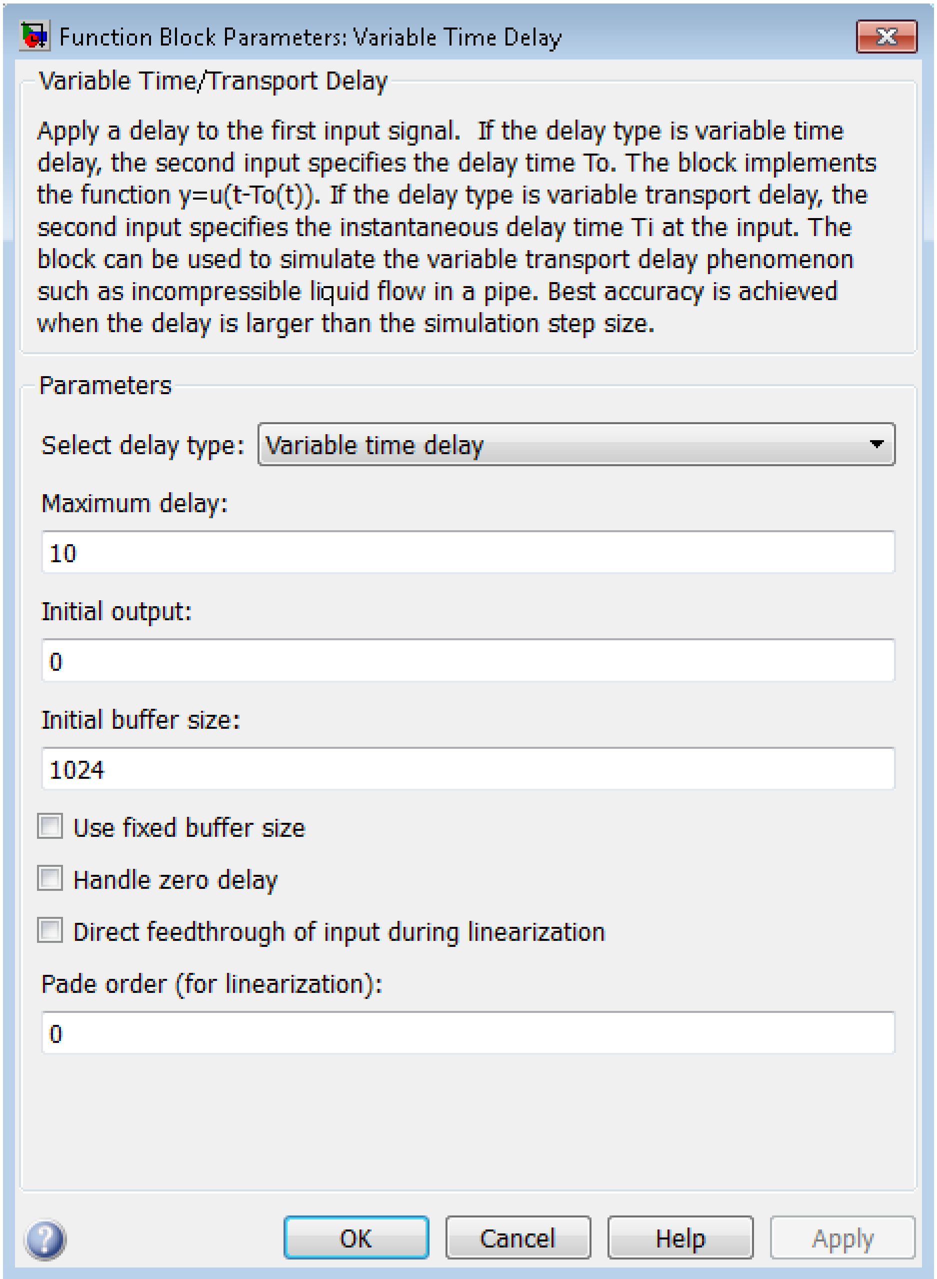
The Variable Time Delay and Variable Transport Delay blocks accept and output real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The parameters and dialog box differ, based on the mode in which the block is operating: variable time or variable transport. Most parameters exist in both modes.

The dialog box for the Variable Time Delay block appears as follows.



Variable Time/Transport Delay

Apply a delay to the first input signal. If the delay type is variable time delay, the second input specifies the delay time To. The block implements the function y=u(t-To(t)). If the delay type is variable transport delay, the second input specifies the instantaneous delay time Ti at the input. The block can be used to simulate the variable transport delay phenomenon such as incompressible liquid flow in a pipe. Best accuracy is achieved when the delay is larger than the simulation step size.

Parameters

Select delay type: Variable time delay

Maximum delay:

10

Initial output:

0

Initial buffer size:

1024

☐ Use fixed buffer size

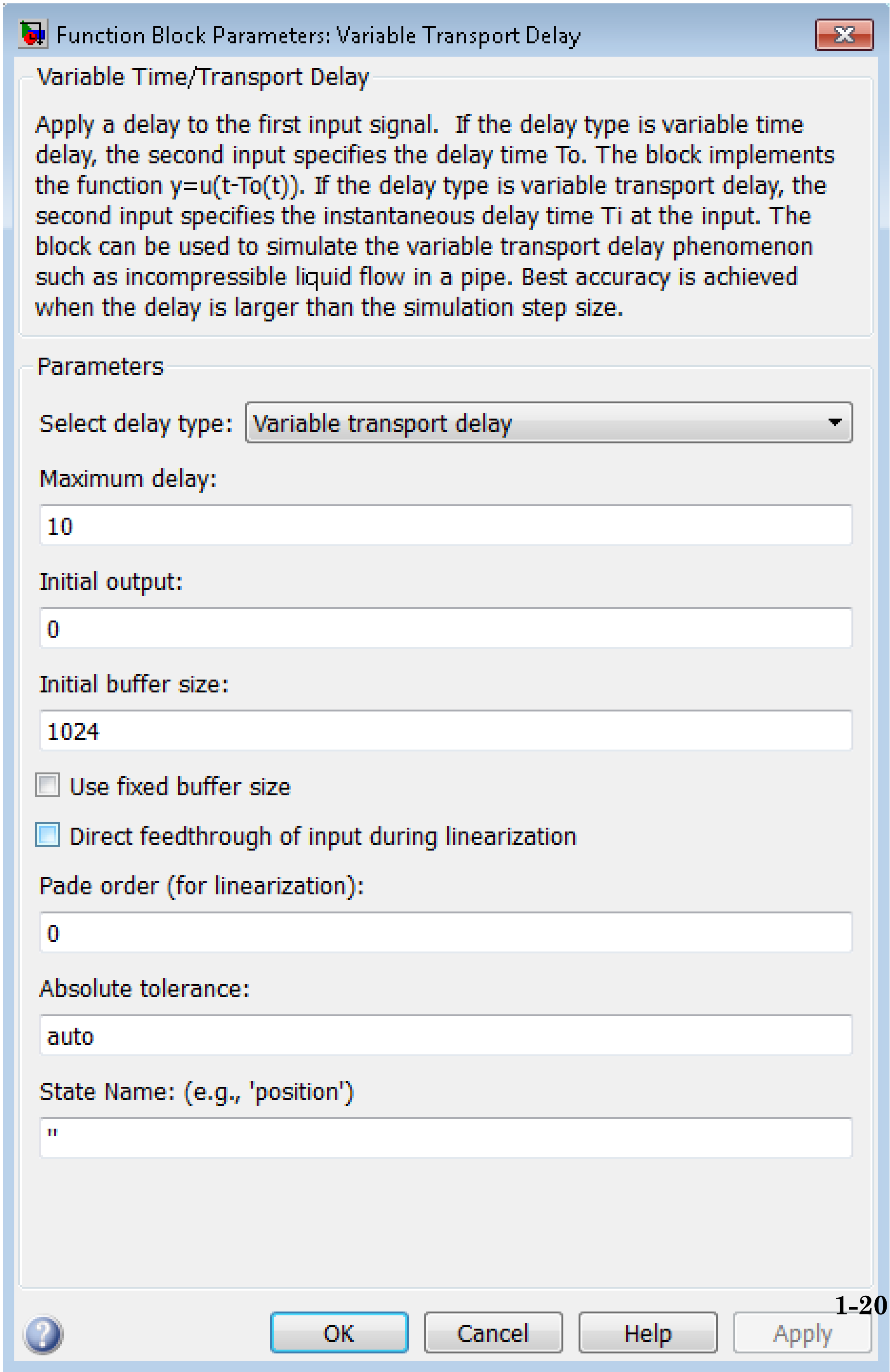
☐ Handle zero delay

☐ Direct feedthrough of input during linearization

Pade order (for linearization):

0

The dialog box for the Variable Transport Delay block appears as follows.



Variable Time/Transport Delay

Apply a delay to the first input signal. If the delay type is variable time delay, the second input specifies the delay time To. The block implements the function y=u(t-To(t)). If the delay type is variable transport delay, the second input specifies the instantaneous delay time Ti at the input. The block can be used to simulate the variable transport delay phenomenon such as incompressible liquid flow in a pipe. Best accuracy is achieved when the delay is larger than the simulation step size.

Parameters

Select delay type: Variable transport delay

Maximum delay: 10

Initial output: 0

Initial buffer size: 1024

☐ Use fixed buffer size

☐ Direct feedthrough of input during linearization

Pade order (for linearization): 0

Absolute tolerance: auto

State Name: (e.g., 'position') ''

Select delay type

Specify the mode in which the block operates.

Settings

Default: The Variable Time Delay block has a default value of `Variable time delay`. The Variable Transport Delay block has a default value of `Variable transport delay`.

`Variable time delay`
Specifies a Variable Time Delay block.

`Variable transport delay`
Specifies a Variable Transport Delay block.

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Maximum delay

Set the maximum value of the time delay input.

Settings

Default: `10`

- This value defines the largest time delay input that this block allows. The block clips any delay that exceeds this value.
- This value cannot be negative. If the time delay becomes negative, the block clips it to zero and issues a warning message.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Initial output

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Settings

Default: `0`

The initial output of this block cannot be `inf` or `NaN`.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Initial buffer size

Define the initial memory allocation for the number of input points to store.

Settings

Default: `1024`

- If the number of input points exceeds the initial buffer size, the block allocates additional memory.
- After simulation ends, a message shows the total buffer size needed.

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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Use fixed buffer size

Specify use of a fixed-size buffer to save input data from previous time steps.

Settings

Default: Off

☑ On

The block uses a fixed-size buffer.

☐ Off

The block does not use a fixed-size buffer.

The **Initial buffer size** parameter specifies the buffer's 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Handle zero delay

Convert this block to a direct feedthrough block.

Settings

Default: Off

☑ On

The block uses direct feedthrough.

☐ Off

The block does not use direct feedthrough.

Dependency

Setting **Select delay type** to `Variable time delay` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Direct feedthrough of input during linearization

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

Settings

Default: Off

☑ On

Enables direct feedthrough of input.

☐ Off

Disables direct feedthrough of input.

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Pade order (for linearization)

Set the order of the Pade approximation for linearization routines.

Settings

Default: `0`

- The default value is 0, which results in a unity gain with no dynamic states.
- Setting the order to a positive integer `n` adds `n` states to your model, but results in a more accurate linear model of the transport delay.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Absolute tolerance

Specify the absolute tolerance for computing the block state.

Default: `auto`

- You can enter `auto`, `1`, a real scalar, or a real vector.
- If you enter `auto` or `1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute the block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Dependency

Setting **Select delay type** to `Variable transport delay` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

State Name (e.g., 'position')

Assign a unique name to each state.

Settings

Default: `' '`

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 string, cell array, or structure.

Dependency

Setting **Select delay type** to `Variable transport delay` enables this parameter.

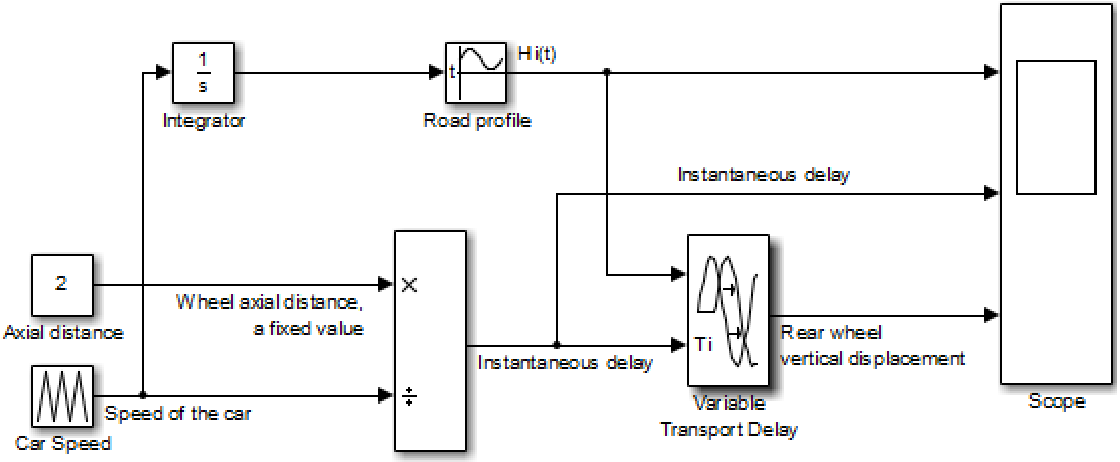
Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

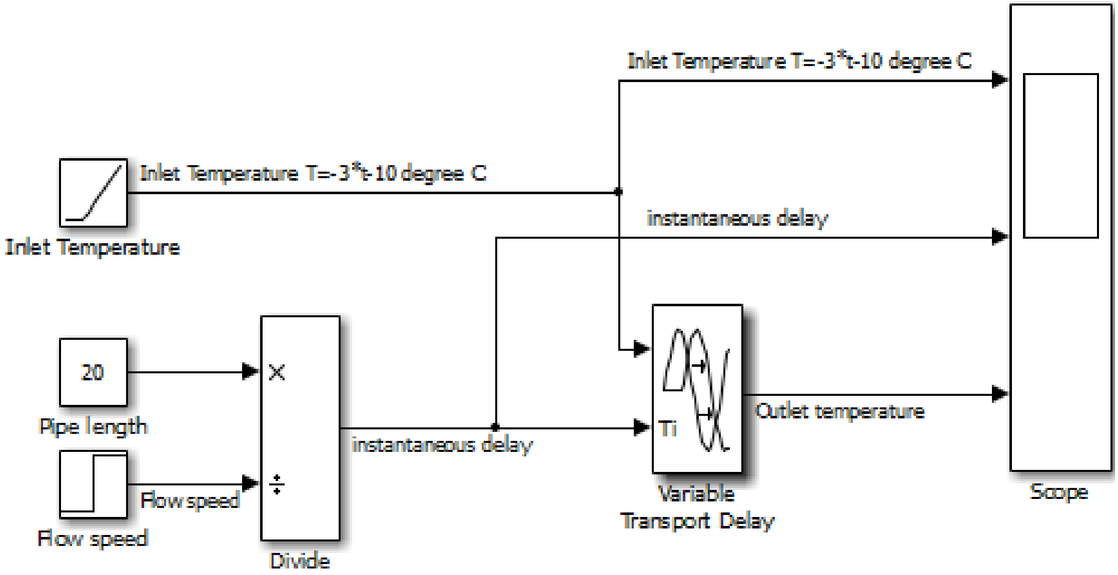
Examples

The `sldemo_VariableTransportDelay` and `sldemo_VariableTransportDelay_pipe` models show how you can use the Variable Transport Delay block.

The `sldemo_VariableTransportDelay` model shows how to model vertical wheel displacement on a one-dimensional car. The Variable Transport Delay block models the delay in vertical displacement of the rear wheel when the road profile changes:


1
s
Integrator
t
Road profile
Hi(t)
Instantaneous delay
2
Axial distance
Wheel axial distance,
a fixed value
×
÷
Speed of the car
Car Speed
Instantaneous delay
Ti
Variable
Transport Delay
Rear wheel
vertical displacement
Scope

The `sldemo_VariableTransportDelay_pipe` model shows how to model incompressible flow through a fixed-length pipe. The Variable Transport Delay block models the delay in temperature change at the outlet when fluid flow occurs:


Inlet Temperature T=-3*t-10 degree C
Inlet Temperature T=-3*t-10 degree C
Inlet Temperature
instantaneous delay
20
Pipe length
×
÷
Flow speed
Flow speed
Divide
instantaneous delay
Ti
Outlet temperature
Variable Transport Delay
Scope

Characteristics

Direct Feedthrough	Yes, of the time delay (second) input
Sample Time	Continuous
Scalar Expansion	Yes, of input and all parameters except **Initial buffer size**
Dimensionalized	Yes
Zero-Crossing Detection	No

See Also

Transport Delay

Variant Subsystem

Purpose Represent a subsystem with multiple subsystems

Library Ports & Subsystems

Description

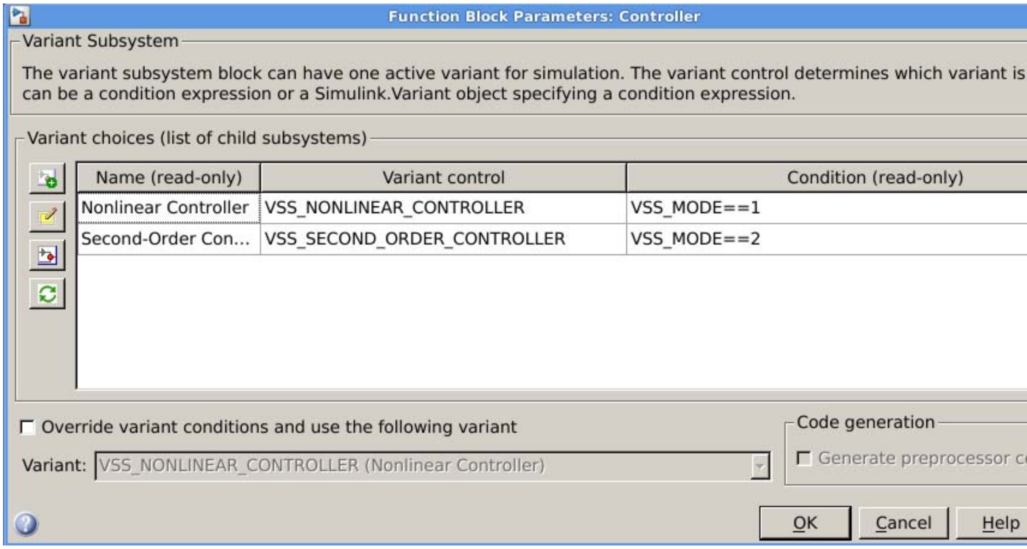
Variant subsystems provide multiple implementations for a subsystem where only one implementation is active during simulation. You can programmatically swap out the active implementation with another implementation without modifying the model.

The Variant Subsystem block includes multiple child subsystems, where only one subsystem is active during simulation. The Variant Subsystem block can include Inport, Outport, and Connection Port blocks. There are no drawn connections inside the Variant Subsystem block. Each child subsystem is associated with a variant control, which is created in the base workspace. The variant control that evaluates to `true`, determines the active variant.

Data Type Support

For information on the data types accepted by a subsystem input ports, see Inport block. For information on data types output by a subsystem output ports, see Outport block.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.


Function Block Parameters: Controller
Variant Subsystem
Variant choices (list of child subsystems)
Code generation
OK
Cancel
Help

The variant subsystem block can have one active variant for simulation. The variant control determines which variant is
can be a condition expression or a Simulink.Variant object specifying a condition expression.

Name (read-only)	Variant control	Condition (read-only)
Nonlinear Controller	VSS_NONLINEAR_CONTROLLER	VSS_MODE==1
Second-Order Con...	VSS_SECOND_ORDER_CONTROLLER	VSS_MODE==2

☐ Override variant conditions and use the following variant

Variant: VSS_NONLINEAR_CONTROLLER (Nonlinear Controller)

☐ Generate preprocessor c

Parameters and Dialog Box

- “Variant choices (list of child subsystems)” on page 1-2046
- “Name (read-only)” on page 1-2048
- “Variant Control” on page 1-2049
- “Condition (read-only)” on page 1-2050
- “Override variant conditions and use following variant” on page 1-2051
- “Variant” on page 1-2052
- “Generate preprocessor conditionals” on page 1-2053

Variant choices (list of child subsystems)

Displays a table of variant choices, variant control, and conditions. The Variant control that evaluates to true determines the active variant.

Settings

Default: The table has a row for each subsystem in the Variant Subsystem block. If the Variant Subsystem block does not contain any subsystems, then the table is empty. See each column parameter for its default value:

- "Name (read-only)" on page 1-2048
- "Variant Control" on page 1-2049
- "Condition (read-only)" on page 1-2050

Tip

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: Places a new subsystem choice in the table and creates a new subsystem block in the Variant Subsystem block diagram.	⊕
Create/Edit selected variant object: Creates a `Simulink.Variant` object in the base workspace and opens the `Simulink.Variant` object parameter dialog box to specify the variant **Condition**.	✎

To...	Click...
Open selected subsystem variant choice: Opens the subsystem block diagram for the selected row in the **Variant choices** table.	
Refresh dialog information from variant subsystem contents: Updates the **Variant choices** table according to the Subsystem block configuration and values of the variant control in the base workspace.	🔄

See Also

- "Configure the Variant Subsystem Block"
- "Set Up Variant Subsystems"
- "Variant Systems"

Name (read-only)

Name of the subsystem (a subsystem contained in the Variant Subsystem block)

Settings

A read only field, based on the subsystems contained in the Variant Subsystem block. To add a new subsystem to the Variant Subsystem block, click the **Create and add a new subsystem choice** button ⊕.

Variant Control

Displays the variant controls available in the base workspace. The variant control can be a boolean condition expression, or a `Simulink.Variant` object representing a boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.

Settings

Default: `Variant`

To enter a variant name, double-click a **Variant control** cell in a new row and type in the variant control expression.

Command-Line Information

Structure field: Represented by the `variant.Name` field in the `Variant` parameter structure
Type: string
Value: Variant control that is associated with the variant choice.
Default: `''`

Condition (read-only)

Displays the **Condition** for the variant controls that are of type `Simulink.Variant` object.

Settings

A read-only field, based on the condition for the associated variant control in the base workspace. Create or change a variant condition in the `Simulink.Variant` parameter dialog box or in the base workspace.

See Also

- "Variant Controls and Active Variants"
- `Simulink.Variant`

Override variant conditions and use following variant

Specify whether to designate the active variant from the evaluation of the variant conditions or from the value of the **Variant** parameter.

Settings

Default: Off

☑ On

Override the variant conditions and set the active variant to the variant choice represented by the **Variant** field.

☐ Off

Determine the active variant by the value of the variant conditions.

Dependencies

This parameter enables **Variant**.

Command-Line Information

Parameter: `OverrideUsingVariant`
Type: string
Value: `''` if no overriding variant is specified.
Default: `''`

See Also

- “Override Variants”
- “Select the Active Variant for Simulation”

Variant

Specify the name of the variant to use if you select **Override variant conditions and use the following variant**.

Settings

Default: `''`

Must be a valid MATLAB identifier.

Tips

You can use the **Variant** drop-down list to see a list of all variants currently in the base workspace.

Dependencies

Enable variants and **Override variant conditions and use the following variant** enables this parameter.

Command-Line Information

Parameter: `OverrideUsingVariant`
Type: string
Value: Specified by the variant control expression.

See Also

- "Override Variants"
- `Simulink.Variant`

Generate preprocessor conditionals

Control whether generated code contains preprocessor conditionals for this Variant Subsystem block.

Settings

Default: Disabled

Dependencies

- The check box is available for generating only ERT targets.
- **Override variant conditions and use following variant** is cleared (`'off'`)
- **Model Configuration Parameters > Optimization > Signals and Parameters > Inline parameters** is selected (`'on'`)
- **Model Configuration Parameters > Code Generation > Interface > Generate preprocessor conditionals** is set to **Use local settings**.

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`
Type: string
Value: `'off'` | `'on'`
Default: `'off'`

See Also

- "Set Up Variant Subsystems"
- "Variant Systems"

Vector Concatenate, Matrix Concatenate

Purpose

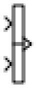
Concatenate input signals of same data type to create contiguous output signal

Library

Math Operations, Signal Routing

Description

The Concatenate block concatenates the signals at its inputs to create an output signal whose elements reside in contiguous locations in memory.



Tip The Concatenate block is useful for creating an output signal that is nonvirtual. However, to create a vector of function calls, use a Mux block instead.

You use a Concatenate block to define an array of buses. For details about defining an array of buses, see "Combine Buses into an Array of Buses".

The Concatenate block operates in either vector or multidimensional array concatenation mode, depending on the setting of its **Mode** parameter. In either case, the block concatenates the inputs from the top to bottom, or left to right, input ports.

Vector Mode

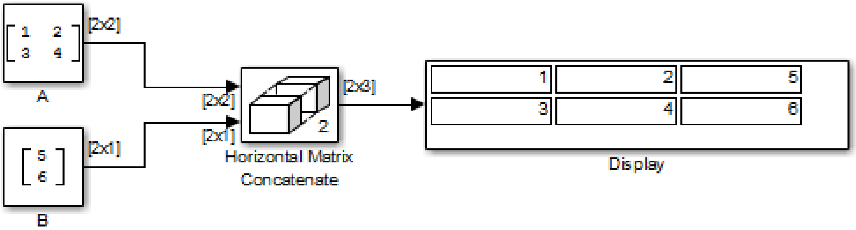
In vector mode, all input signals must be either vectors or row vectors [1xM matrices] or column vectors [Mx1 matrices] or a combination of vectors and either row or column vectors. The output is a vector if all inputs are vectors.

The output is a row or column vector if any of the inputs are row or column vectors, respectively.

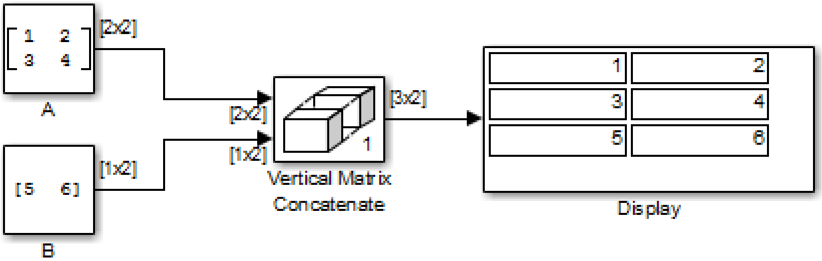
Multidimensional Array Mode

Multidimensional array mode accepts vectors and arrays of any size. It assumes that the trailing dimensions are all ones for input signals with lower dimensionality. For example, if the output is 4-D and the input is `[2x3]` (2-D) this block treats the input as `[2x3x1x1]`. The output is always an array. The block's **Concatenate dimension** parameter allows you to specify the output dimension along which the block concatenates its input arrays.

If you set the **Concatenate dimension** parameter to 2 and inputs are 2-D matrices, the block performs horizontal matrix concatenation and places the input matrices side-by-side to create the output matrix, for example:

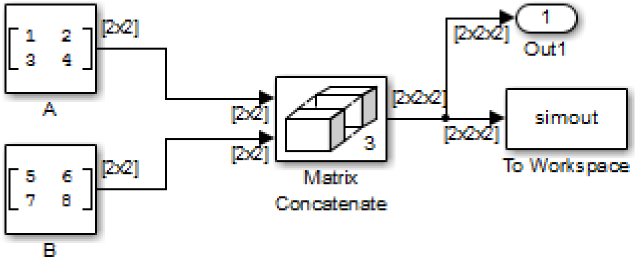

1 2
3 4
A
[2x2]
5
6
B
[2x1]
[2x2]
[2x1]
2
Horizontal Matrix Concatenate
[2x3]
1
2
5
3
4
6
Display

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:


1 2
3 4
A
[2x2]
[5 6]
B
[1x2]
[2x2]
[1x2]
1
Vertical Matrix Concatenate
[3x2]
1
2
3
4
5
6
Display

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:


1 2
3 4
A
5 6
7 8
B
[2x2]
[2x2]
[2x2]
[2x2]
3
Matrix Concatenate
[2x2x2]
[2x2x2]
[2x2x2]
1
Out1
simout
To Workspace

Data Type Support

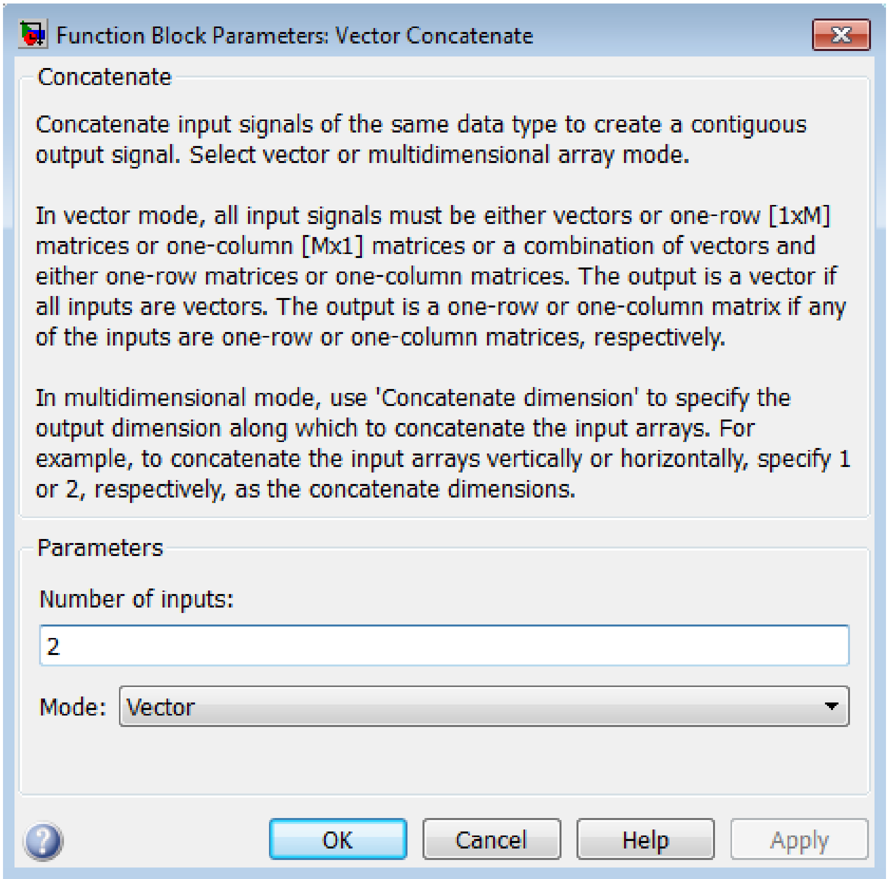
Accepts signals of any data type that Simulink supports. All inputs must be of the same data type. Outputs have the same data type as the input.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

The parameters and dialog box differ, based on the mode in which the block is operating: vector or matrix. Most parameters exist in both modes.

The dialog box for the Vector Concatenate block appears as follows.



Function Block Parameters: Vector Concatenate

Concatenate

Concatenate input signals of the same data type to create a contiguous output signal. Select vector or multidimensional array mode.

In vector mode, all input signals must be either vectors or one-row [1xM] matrices or one-column [Mx1] matrices or a combination of vectors and either one-row matrices or one-column matrices. The output is a vector if all inputs are vectors. The output is a one-row or one-column matrix if any of the inputs are one-row or one-column matrices, respectively.

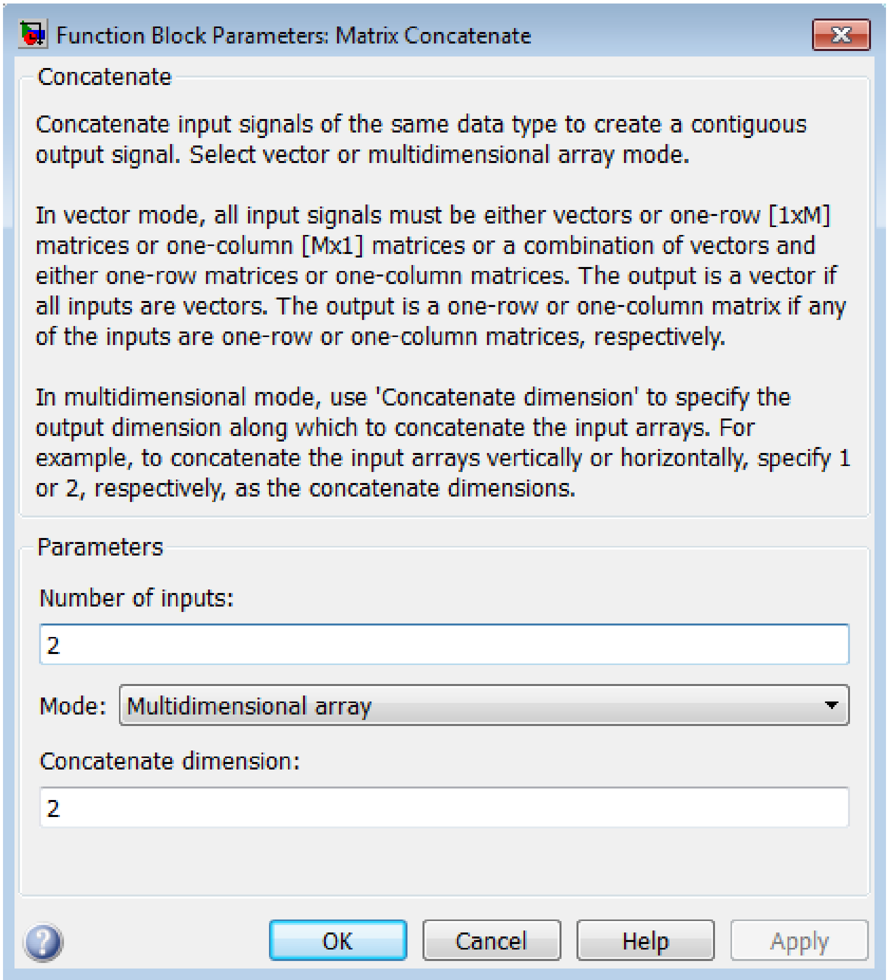
In multidimensional mode, use 'Concatenate dimension' to specify the output dimension along which to concatenate the input arrays. For example, to concatenate the input arrays vertically or horizontally, specify 1 or 2, respectively, as the concatenate dimensions.

Parameters

Number of inputs: 2

Mode: Vector

The dialog box for the Matrix Concatenate block appears as follows.



Concatenate

Concatenate input signals of the same data type to create a contiguous output signal. Select vector or multidimensional array mode.

In vector mode, all input signals must be either vectors or one-row [1xM] matrices or one-column [Mx1] matrices or a combination of vectors and either one-row matrices or one-column matrices. The output is a vector if all inputs are vectors. The output is a one-row or one-column matrix if any of the inputs are one-row or one-column matrices, respectively.

In multidimensional mode, use 'Concatenate dimension' to specify the output dimension along which to concatenate the input arrays. For example, to concatenate the input arrays vertically or horizontally, specify 1 or 2, respectively, as the concatenate dimensions.

Parameters

Number of inputs: 2

Mode: Multidimensional array

Concatenate dimension: 2

Number of inputs

Specifies the number of inputs for the block.

Settings

Default: `2`

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Mode

Select the type of concatenation that this block performs.

Settings

Default: `Vector` (for the Vector Concatenate block), `Multidimensional array` (for the Matrix Concatenate block)

`Vector`
: Perform vector concatenation (see "Vector Mode" on page 1-2054 for details).

`Multidimensional array`
: Perform matrix concatenation (see "Multidimensional Array Mode" on page 1-2054 for details).

Dependency

This parameter enables **Concatenate dimension**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Concatenate dimension

Specifies the output dimension along which to concatenate the input arrays.

Settings

Default: `2`

- Enter `1` to concatenate input arrays vertically.
- Enter `2` to concatenate input arrays horizontally.
- Enter a higher dimension to perform multidimensional concatenation on the inputs.

Dependency

Selecting `Multidimensional array` for **Mode** enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

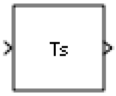
`cat` in the MATLAB reference documentation

Weighted Sample Time

Purpose Support calculations involving sample time

Library Signal Attributes

Description The Weighted Sample Time block is an implementation of the Weighted Sample Time Math block. See Weighted Sample Time Math for more information.



Weighted Sample Time Math

Purpose

Support calculations involving sample time

Library

Math Operations

Description

The Weighted Sample Time Math block adds, subtracts, multiplies, or divides its input signal, `u`, by a weighted sample time, `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.

You specify the math operation with the **Operation** parameter. Additionally, you can specify to use only the weight with either the sample time or its inverse.

Enter the weighting factor in the **Weight value** parameter. If the weight, `w`, is `1`, that value does not appear in the equation on the block icon.

The block computes its output using the precedence rules for MATLAB operators (see "Operator Precedence" 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
```

Data Type Support

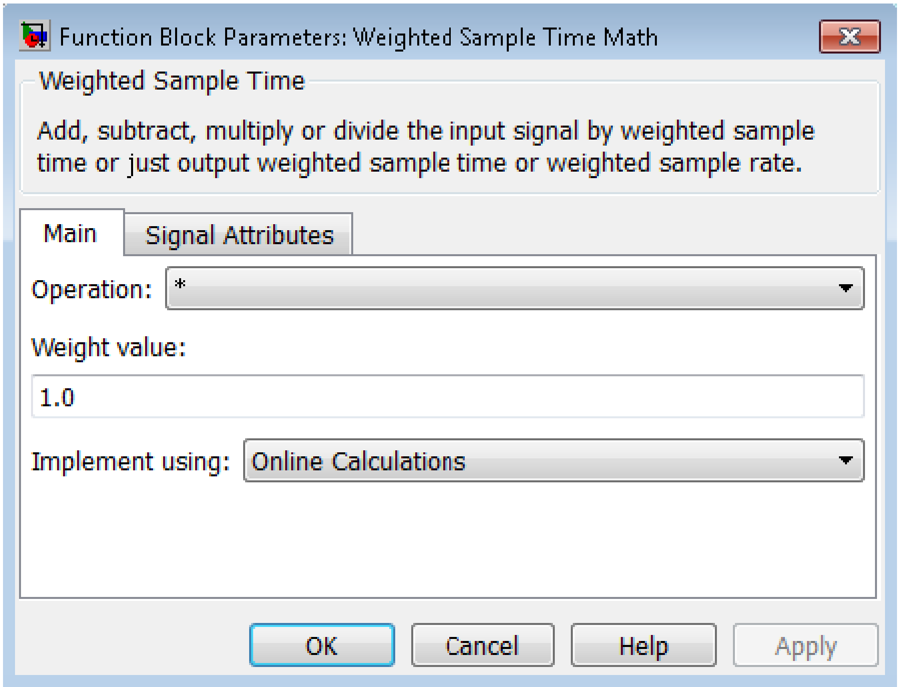
The Weighted Sample Time Math block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** pane of the Weighted Sample Time Math block dialog box appears as follows:



Function Block Parameters: Weighted Sample Time Math
Weighted Sample Time

Add, subtract, multiply or divide the input signal by weighted sample time or just output weighted sample time or weighted sample rate.

Main
Signal Attributes

Operation: *

Weight value: 1.0

Implement using: Online Calculations

OK
Cancel
Help
Apply

Operation

Specify operation to use: `+`, `-`, `*`, `/`, `Ts Only`, or `1/Ts Only`.

Weight value

Enter the weight of sample time.

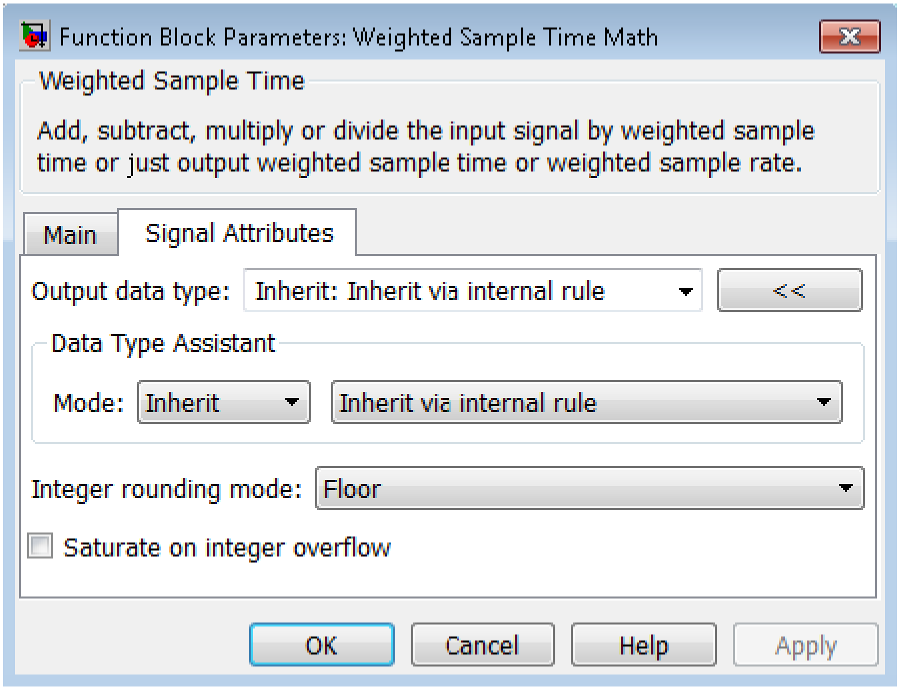
Implement using

Select one of two modes: online calculations or offline scaling adjustment. This parameter is visible only when you set **Operation** to `*` or `/`.

Result of (Ts * w)	Output Data Type of Two Modes	Block Execution
A power of 2, or an integer value	The same, when **Output data type** is `Inherit: Inherit via internal rule`	Equally efficient in both modes
Not power of 2 and not an integer value	Different	More efficient for the offline scaling mode

Note When the **Implement using** parameter is not visible, operations default to online calculations.

The **Signal Attributes** pane of the Weighted Sample Time Math block dialog box appears as follows:



Function Block Parameters: Weighted Sample Time Math

Weighted Sample Time

Add, subtract, multiply or divide the input signal by weighted sample time or just output weighted sample time or weighted sample rate.

Output data type: Inherit: Inherit via internal rule

Data Type Assistant

Mode: Inherit — Inherit via internal rule

Integer rounding mode: Floor

☐ Saturate on integer overflow

Tip The **Saturate on integer overflow** parameter is visible only if:

- The **Operation** parameter specifies + or -.
- The **Operation** parameter specifies * or / and the **Implement using** parameter specifies `Online Calculations`.

Output data type

Specify whether the block inherits the output data type by an internal rule or back propagation.

Tip If you enter an expression in the edit field, the expression must evaluate to one of the two inherit rules.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see "Rounding". in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to –128 or 127.
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 "Checking 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.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	For all math operations except `Ts` and `1/Ts`
Scalar Expansion	No, the weight is always a scalar
Zero-Crossing Detection	No

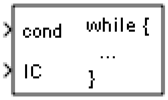
Purpose

Repeatedly execute contents of subsystem at current time step while condition is satisfied

Library

Ports & Subsystems

Description



The While Iterator block, when placed in a subsystem, repeatedly executes the contents of the subsystem at the current time step while a specified condition is true.

Note Placing a While Iterator block in a subsystem makes it an atomic subsystem if it is not already an atomic subsystem.

The output of a While Iterator subsystem cannot be a function-call signal. Otherwise, Simulink displays an error when you simulate the model or update the diagram.

You can use this block to implement the block-diagram equivalent of a C program `while` or `do-while` loop. In particular, use **While loop type** to select one of the following while loop modes:

- `do-while`

  In this mode, the While Iterator block has one input, the while condition input, whose source must reside in the subsystem. At each time step, the block runs all the blocks in the subsystem once and then checks whether the while condition input is true. If the input is true, the iterator block runs the blocks in the subsystem again. This process continues as long as the while condition input is true and the number of iterations is less than or equal to the **Maximum number of iterations**.

- `while`

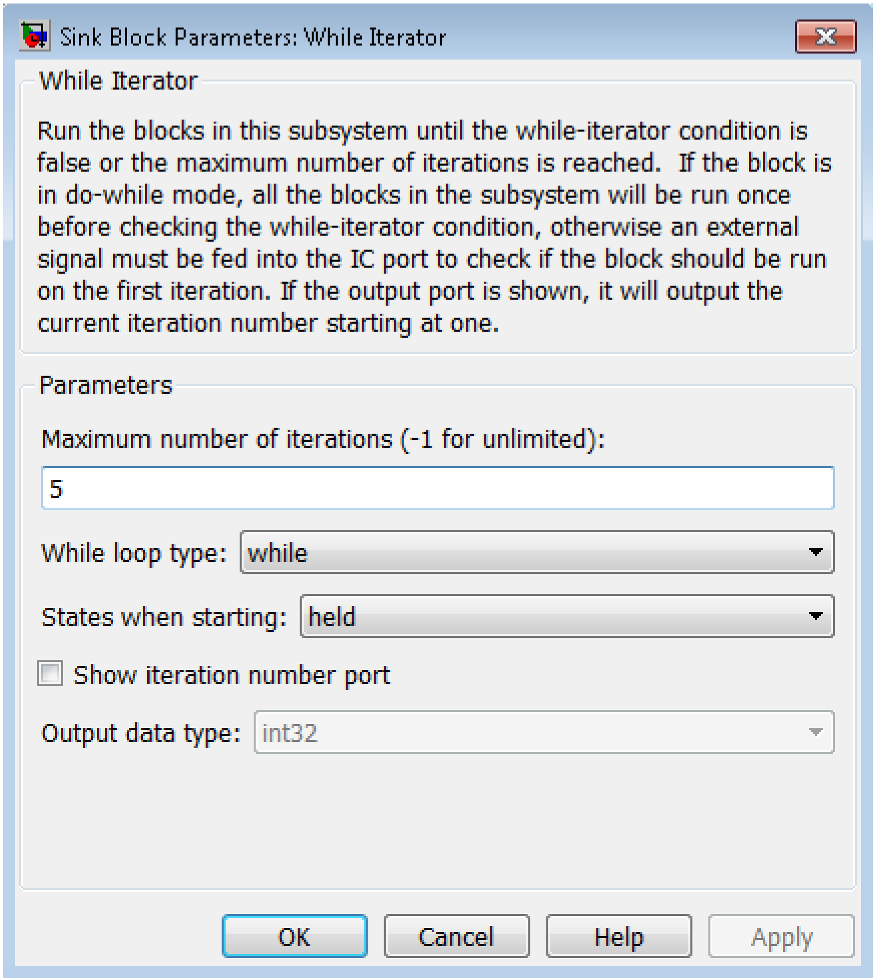
  In this mode, the iterator block has two inputs: a while condition input and an initial condition (IC) input. The source of the initial condition signal must be external to the while subsystem. At the beginning of the time step, if the IC input is true, the iterator block executes the contents of the subsystem and then checks the while condition input. If the while condition input is true, the iterator executes the subsystem again. This process continues as long as the while condition 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 at the beginning of a time step, the iterator does not execute the contents of the subsystem during the time step.

Tip Specify a maximum number of iterations to avoid infinite loops, which you can break only by terminating MATLAB.

Data Type Support

Acceptable data inputs for the condition ports are any numeric data type that Simulink supports, as well as any fixed-point type that includes a `0` value. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

The optional output port can output any of the following data types: `double`, `int32`, `int16`, or `int8`.



Sink Block Parameters: While Iterator

While Iterator

Run the blocks in this subsystem until the while-iterator condition is false or the maximum number of iterations is reached. If the block is in do-while mode, all the blocks in the subsystem will be run once before checking the while-iterator condition, otherwise an external signal must be fed into the IC port to check if the block should be run on the first iteration. If the output port is shown, it will output the current iteration number starting at one.

Parameters

Maximum number of iterations (-1 for unlimited):

5

While loop type: while

States when starting: held

☐ Show iteration number port

Output data type: int32

Parameters and Dialog Box

Maximum number of iterations

Specify the maximum number of iterations allowed. A value of -1 allows any number of iterations as long as the while condition input is true. Note that if you specify -1 and the while condition never becomes false, the simulation will run forever. In this case, the only way to stop the simulation is to terminate the MATLAB process. Therefore, do not specify –1 as the value of this parameter unless you are certain that the while condition becomes false at some point in the simulation.

While loop type

Specify the type of while loop that this block implements.

States when starting

Set this field to `reset` if you want the iterator block to reset the states of the blocks in the while subsystem to their initial values at the beginning of each time step (i.e., before executing the first loop iteration in the current time step). To cause the states of blocks in the subsystem to persist across time steps, set this field to `held` (the default).

Show iteration number port

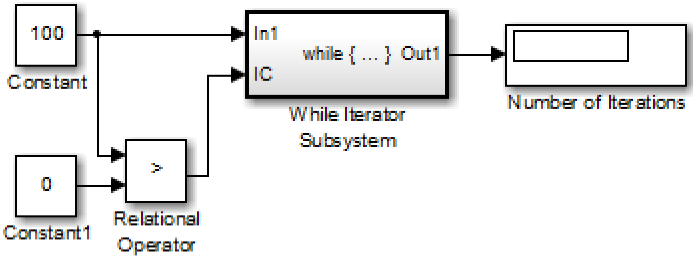
If you select this check box, the While Iterator block outputs its iteration value. This value starts at `1` and is incremented by 1 for each succeeding iteration. By default, this check box is not selected.

Output data type

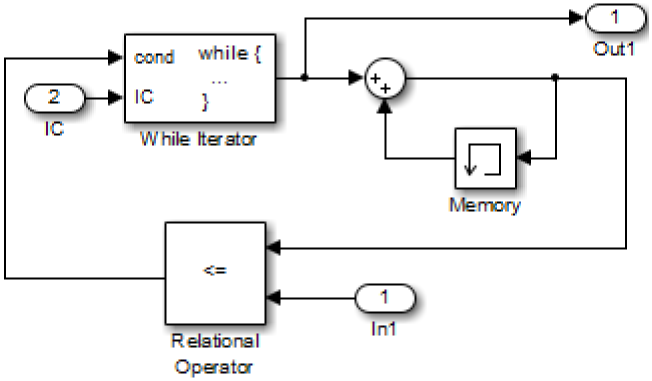
If you select the **Show iteration number port** check box (the default), this field is enabled. Use it to set the data type of the iteration number output to `int32`, `int16`, `int8`, or `double`.

Examples

The While Iterator block can optionally output the current iteration number, starting at 1. The following model uses this capability to compute N, where N is the first N integers whose sum is less than 100.


100
Constant
0
Constant1
>
Relational Operator
In1
IC
while { ... } Out1
While Iterator Subsystem
Number of Iterations

The contents of the While Iterator subsystem are:


cond
IC
while {
...
}
While Iterator
2
IC
1
Out1
+
+
Memory
<=
Relational Operator
1
In1

The While Iterator block uses the following parameter settings:

- **Maximum number of iterations** set to 20
- **States when starting** set to `reset`

The model is the diagrammatic equivalent of the following pseudocode:

```
max_sum = 100;
sum = 0;
iteration_number = 0;
cond = (max_sum > 0);
while (cond != 0) {
 iteration_number = iteration_number + 1;
 sum = sum + iteration_number;
 if (sum > max_sum OR iteration_number > max_iterations)
  cond = 0;
}
```

Characteristics

Direct Feedthrough	No
Sample Time	Inherited from the driving block
Scalar Expansion	No
Dimensionalized	No
Zero-Crossing Detection	No

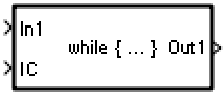
Purpose

Represent subsystem that executes repeatedly while condition is satisfied during simulation time step

Library

Ports & Subsystems

Description



The While Iterator Subsystem block is a Subsystem block that is preconfigured to serve as a starting point for creating a subsystem that executes repeatedly while a condition is satisfied during a simulation time step.

See the While Iterator block and “Control Flow Logic” for more information.

When using simplified initialization mode, you cannot place any block needing elapsed time within an Iterator Subsystem. In simplified initialization mode, Iterator subsystems do not maintain elapsed time, so Simulink will report an error if any such block (such as the Discrete-Time Integrator block) is placed within the subsystem. For more information on simplified initialization modes, see “Underspecified initialization detection”.

Width

Purpose

Output width of input vector

Library

Signal Attributes

Description

The Width block generates as output the width of its input vector.

You can use an array of buses as an input signal to a Width block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Data Type Support

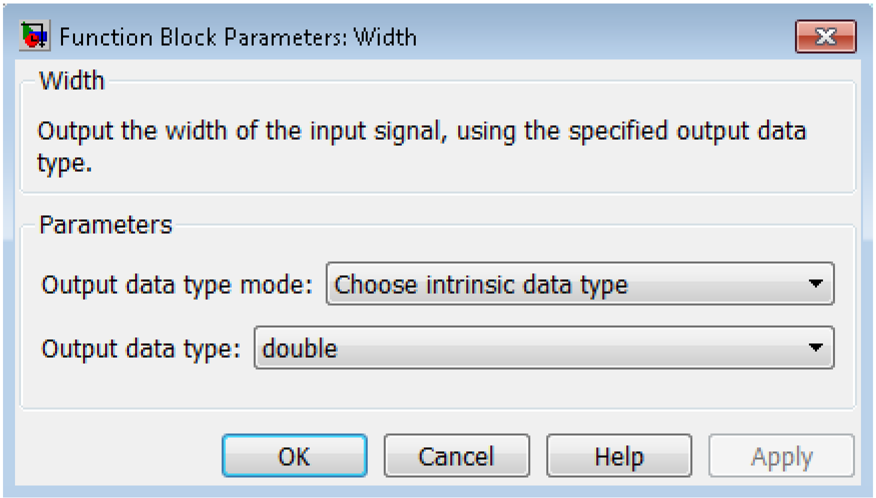
The Width block accepts real or complex signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The Width block also supports mixed-type signal vectors.

When the **Output data type mode** is not `Choose intrinsic data type`, the block supports only built-in numeric types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Function Block Parameters: Width

Width

Output the width of the input signal, using the specified output data type.

Parameters

Output data type mode: Choose intrinsic data type

Output data type: double

Note The Width block ignores the **Data type override** setting of the Fixed-Point Tool.

Output data type mode

Specify the output data type to be the same as the input, or inherit the data type by back propagation. You can also choose to specify a built-in data type from the drop-down list in the **Output data type** parameter.

Output data type

This parameter is visible when you select `Choose intrinsic data type` for **Output data type mode**. Select a built-in data type from the drop-down list.

Characteristics

Sample Time	Constant
Dimensionalized	Yes

Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

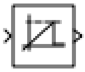
Set output to zero if input is above threshold

Library

Discontinuities

Description

The Wrap To Zero block sets the output to zero when the input is above the **Threshold** value. However, the block outputs the input when the input is less than or equal to the **Threshold**.



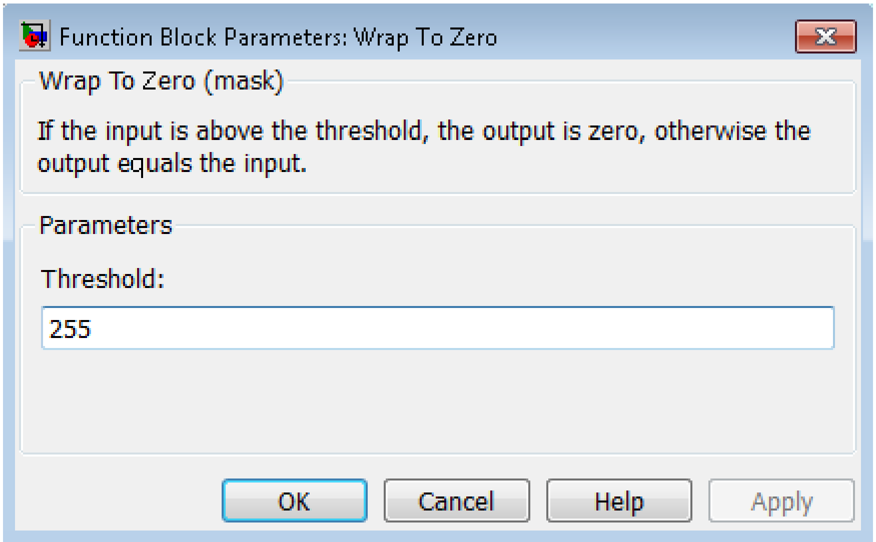
Data Type Support

The Wrap To Zero block accepts inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The block output has the same data type as the input. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Tip If the input data type cannot represent zero, parameter overflow occurs. To detect this overflow, go to the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box and set **Parameters > Detect overflow** to `warning` or `error`.


Function Block Parameters: Wrap To Zero

Wrap To Zero (mask)

If the input is above the threshold, the output is zero, otherwise the output equals the input.

Parameters

Threshold:

255

OK
Cancel
Help
Apply

Parameters and Dialog Box

Threshold

When the input exceeds the threshold, the block sets the output to zero.

Characteristics

Direct Feedthrough	Yes
Scalar Expansion	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

Purpose

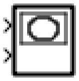
Display X-Y plot of signals using MATLAB figure window

Library

Sinks

Description

The XY Graph block displays an X-Y plot of its inputs in a MATLAB figure window.



The block has two scalar inputs. The block plots data in the first input (the *x* direction) against data in the second input (the *y* direction). (See "How to Rotate a Block" for a description of the port order for various block orientations.) This block is useful for examining limit cycles and other two-state data. Data outside the specified range does not appear.

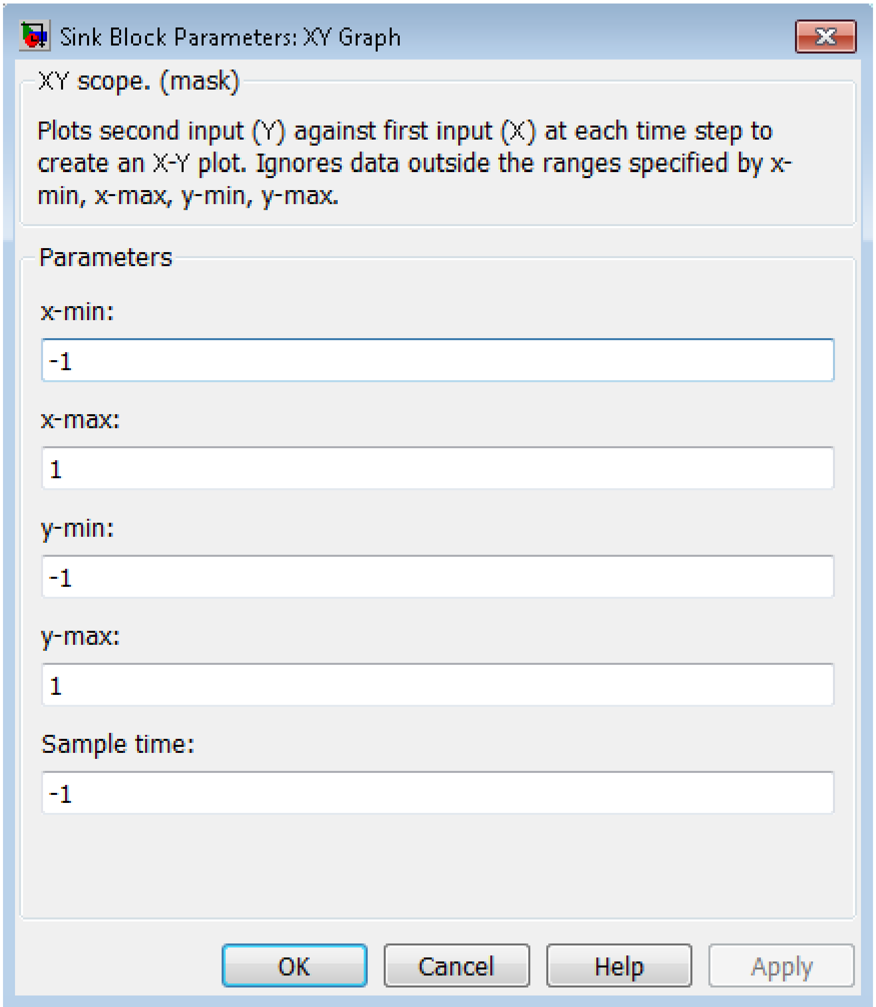
A figure window appears for each XY Graph block in the model at the start of simulation.

Data Type Support

The XY Graph block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.


Sink Block Parameters: XY Graph

XY scope. (mask)

Plots second input (Y) against first input (X) at each time step to create an X-Y plot. Ignores data outside the ranges specified by x-min, x-max, y-min, y-max.

Parameters

x-min: -1

x-max: 1

y-min: -1

y-max: 1

Sample time: -1

OK
Cancel
Help
Apply

Parameters and Dialog Box

x-min

Specify the minimum *x*-axis value. The default is -1.

x-max

Specify the maximum *x*-axis value. The default is 1.

y-min

Specify the minimum *y*-axis value. The default is `-1`.

y-max

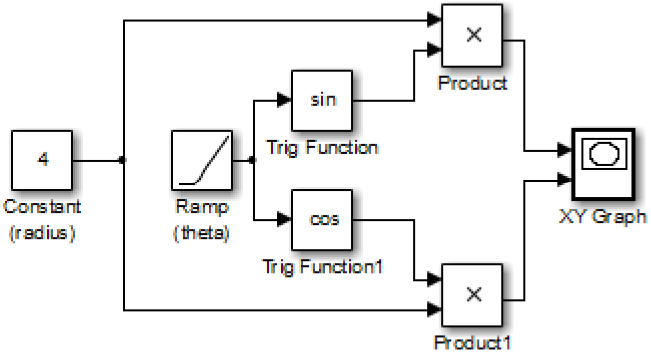
Specify the maximum *y*-axis value. The default is `1`.

Sample time

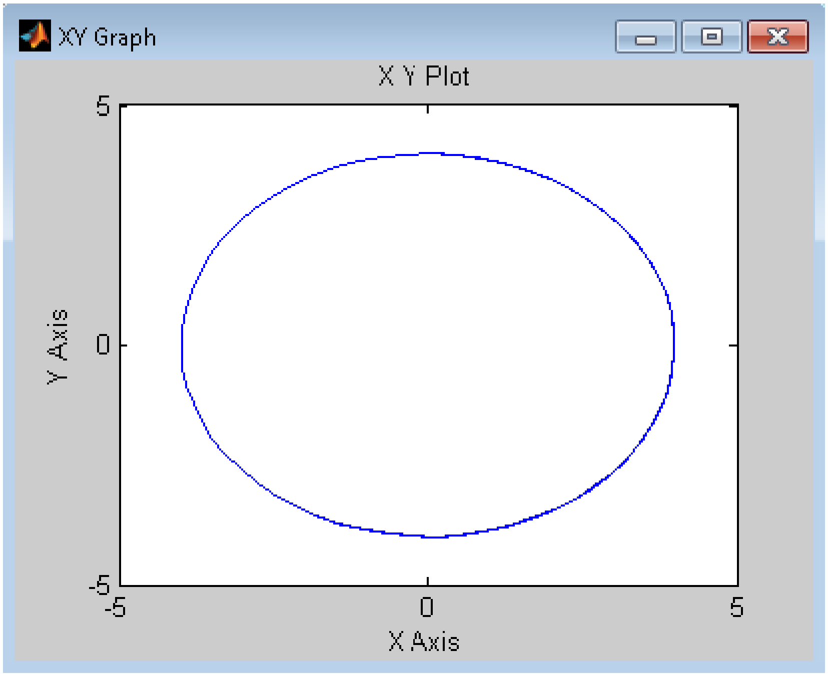
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the Simulink documentation for more information.

Examples

The following model computes the points that define a circle of radius 4, centered at the origin of the *x*-*y* plane.


4
Constant
(radius)
Ramp
(theta)
sin
Trig Function
cos
Trig Function1
×
Product
×
Product1
XY Graph

When you simulate the model, a figure window appears.


XY Graph
X Y Plot
5
0
-5
-5
0
5
Y Axis
X Axis

Characteristics

Sample Time	Specified in the **Sample time** parameter
States	0
Zero-Crossing Detection	No

Zero-Order Hold

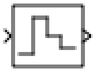
Purpose

Implement zero-order hold of one sample period

Library

Discrete

Description



The Zero-Order Hold block holds its input for the sample period you specify. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds all elements of the vector for the same sample period.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means the block inherits the **Sample time**.

Tip Do not use the Zero-Order Hold block to create a fast-to-slow transition between blocks operating at different sample rates. Instead, use the Rate Transition block.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
Unit Delay	Implement a delay using a discrete sample time that you specify. Ideally, the block accepts and outputs signals with a discrete sample time.	• `sldemo_enginewc` (Compression subsystem)
Memory	Implement a delay by one integration time step. Ideally, the block accepts	• `sldemo_bounce` • `sldemo_clutch` (Friction Mode

Block	Purpose of the Block	Reference Examples
	and outputs signals where the sample time is continuous or fixed in minor time step. For more information, see "Types of Sample Time" in the Simulink documentation.	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`

Overview of Block Capabilities

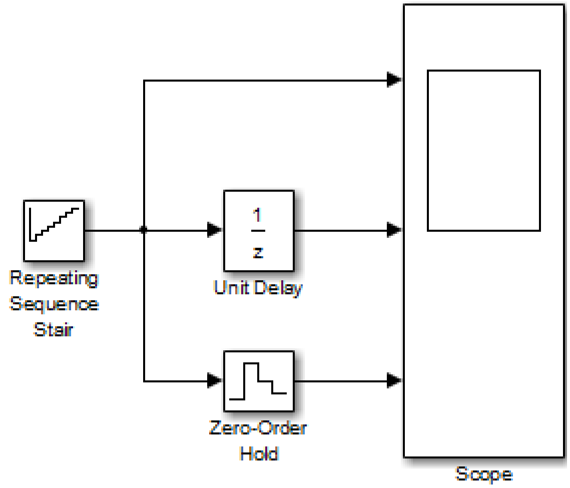
Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of initial condition	Yes	Yes	No, because the block output at time t = 0 must match the input value.
Specification of sample time	Yes	No, because the block can only inherit sample time (from the driving block or the solver used for the entire model).	Yes

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Support for frame-based signals	Yes	No	Yes
Support for state logging	Yes	No	No

Effect of Solver Specification on Block Output

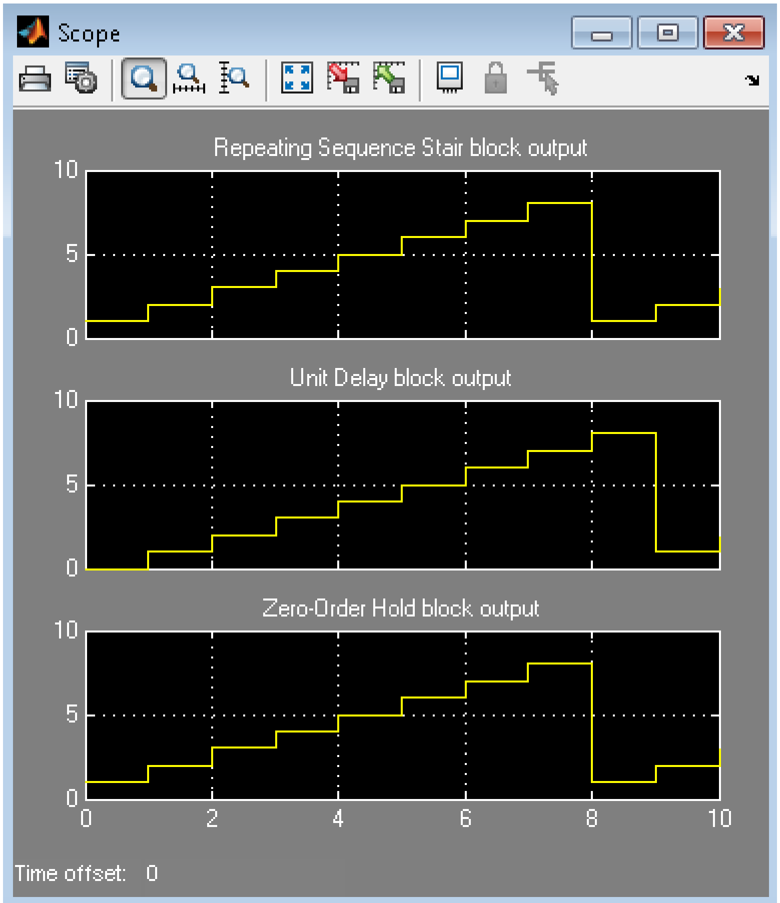
When you specify a discrete sample time in the dialog box for a Unit Delay or Zero-Order Hold block, the block output can differ depending on the solver specification for the model.

Suppose that you have a model with Unit Delay and Zero-Order Hold blocks, which both use a discrete sample time of 1:

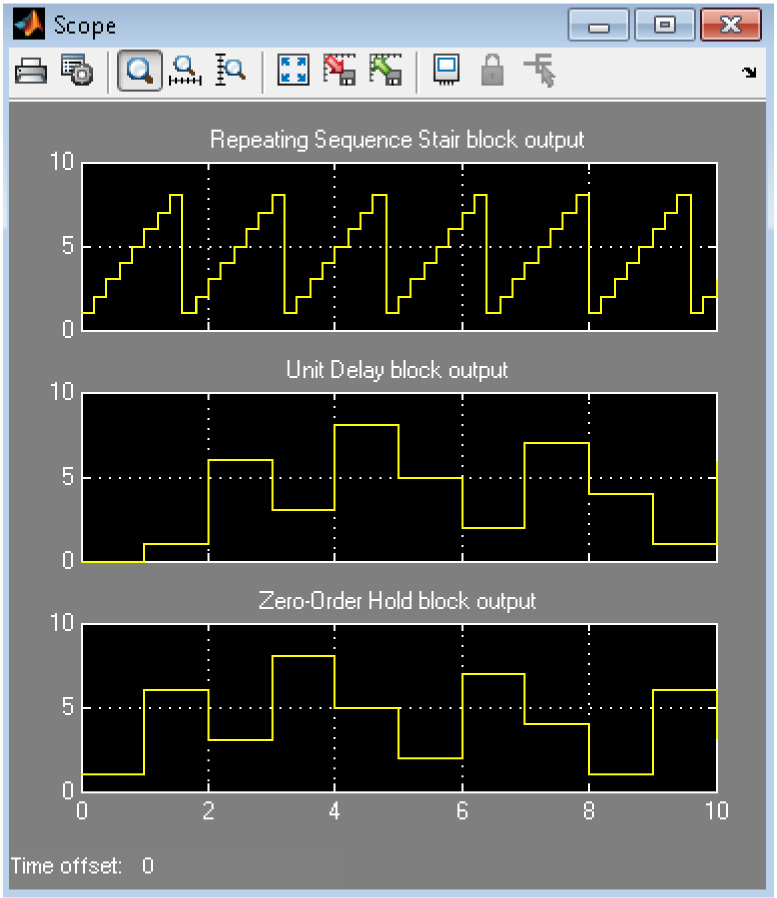


The Repeating Sequence Stair block uses a continuous sample time of 0 to provide input signals to the Unit Delay and Zero-Order Hold blocks.

If the model uses a fixed-step solver with a step size of 1, the scope shows the following simulation results:


Scope
Repeating Sequence Stair block output
10
5
0
Unit Delay block output
10
5
0
Zero-Order Hold block output
10
5
0
0
2
4
6
8
10
Time offset: 0

If the model uses a variable-step solver, the scope shows the following simulation results:


Scope
Repeating Sequence Stair block output
Unit Delay block output
Zero-Order Hold block output
10
5
0
0
2
4
6
8
10
Time offset: 0

The Zero-Order Hold block takes the input value of the Repeating Sequence Stair block at t = 0, 1, 2, ... , 9 and holds each input value for a sample period (1 second). The Unit Delay block applies the same 1-second hold to each input value of the Repeating Sequence Stair block, but also delays each value by a sample period. The **Initial conditions**

parameter specifies the output for the Unit Delay block during the first sample period. For more information about sample time, see “What Is Sample Time?” and “Specify Sample Time”.

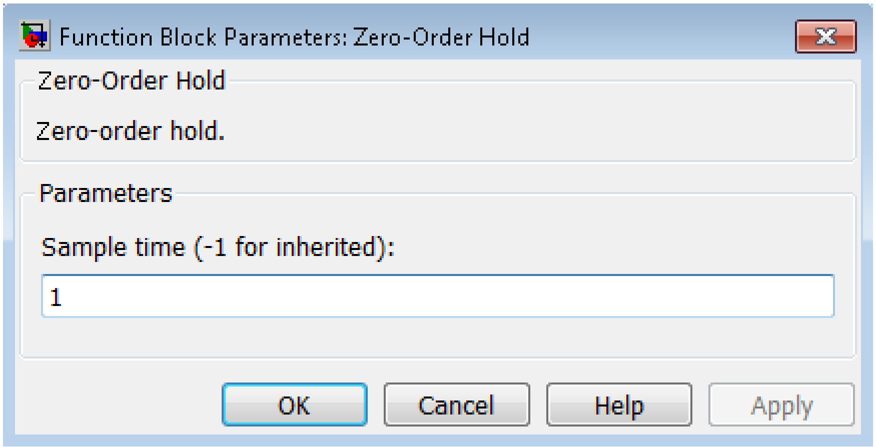
Solver specification for a model also affects the behavior of the Memory block. For details, see “Examples of Memory Block Usage” on page 1-952.

Data Type Support

The Zero-Order Hold block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box


Function Block Parameters: Zero-Order Hold

Zero-Order Hold

Zero-order hold.

Parameters

Sample time (-1 for inherited): 1

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online documentation for more information.

Do not specify a continuous sample time, either `0` or `[0,0]`. This block supports only discrete sample times. When this parameter is `-1`, the inherited sample time must be discrete and not continuous.

Bus Support

The Zero-Order Hold block is a bus-capable block. The input can be a virtual or nonvirtual bus signal. No block-specific restrictions exist. All signals in a nonvirtual bus input to a Zero-Order Hold block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “About Composite Signals” and “Bus-Capable Blocks” for more information.

You can use an array of buses as an input signal to a Zero-Order Hold block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Examples

The following models show how to use the Zero-Order Hold block:

- `sldemo_radar_eml`
- `aero_dap3dof`

Characteristics

Bus-capable	Yes
Direct Feedthrough	Yes
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	No
Dimensionalized	Yes
Zero-Crossing Detection	No

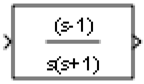
See Also

Memory, Unit Delay

Purpose Model system by zero-pole-gain transfer function

Library Continuous

Description The Zero-Pole block models a system that you define with the zeros, poles, and gain of a Laplace-domain transfer function. This block can model single-input single output (SISO) and single-input multiple-output (SIMO) systems.



Conditions for Using This Block

The Zero-Pole block assumes the following conditions:

- The transfer function has the form

  $$H(s) = K\frac{Z(s)}{P(s)} = K\frac{(s-Z(1))(s-Z(2))\ldots(s-Z(m))}{(s-P(1))(s-P(2))\ldots(s-P(n))},$$

  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**.

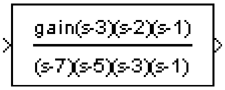
Transfer Function Display on the Block

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.

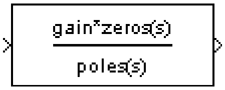
If you specify a variable in parentheses, the block evaluates the variable.

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:



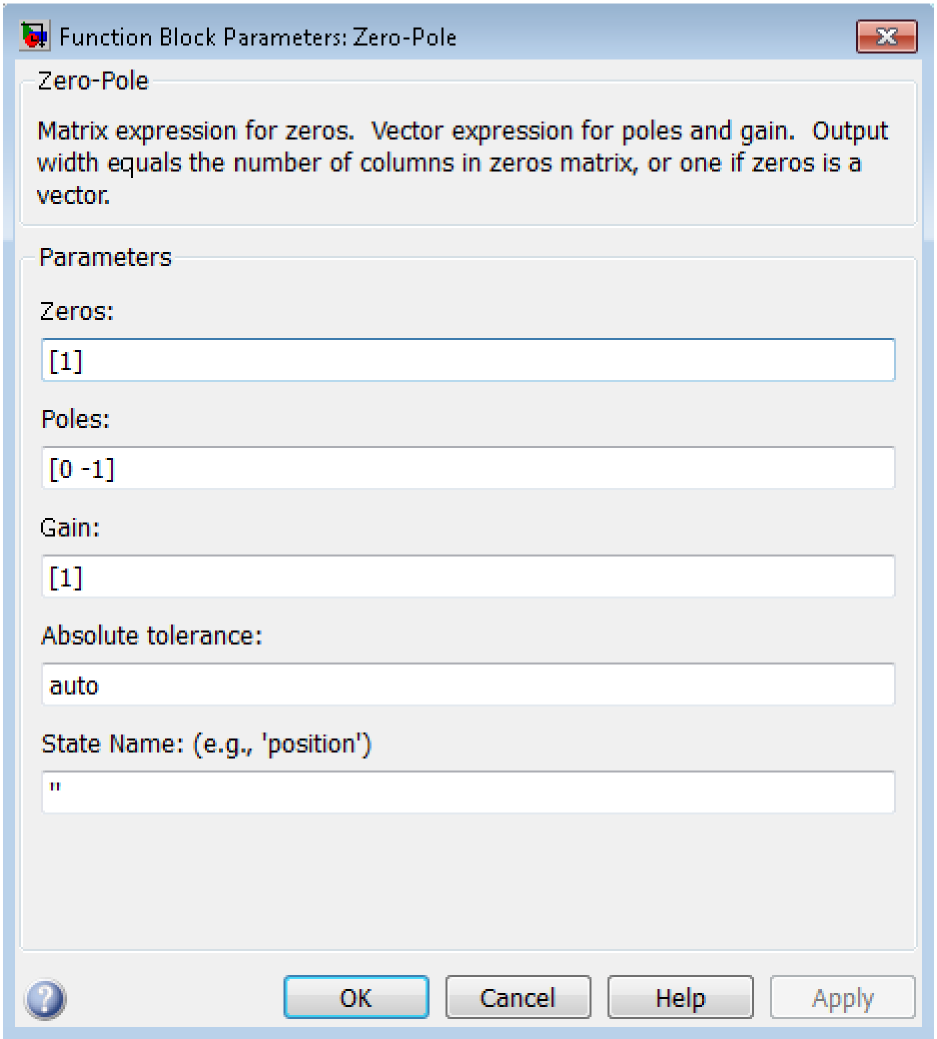
- If you specify each parameter as a variable, the block shows the variable name followed by `(s)` if appropriate.

  For example, if you specify **Zeros** as `zeros`, **Poles** as `poles`, and **Gain** as `gain`, the block looks like this:



Data Type Support

The Zero-Pole block accepts real signals of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.



Function Block Parameters: Zero-Pole

Zero-Pole

Matrix expression for zeros. Vector expression for poles and gain. Output width equals the number of columns in zeros matrix, or one if zeros is a vector.

Parameters

Zeros: [1]

Poles: [0 -1]

Gain: [1]

Absolute tolerance: auto

State Name: (e.g., 'position') ''

Parameters and Dialog Box

Zeros

Define the matrix of zeros.

Settings

Default: `[1]`

Tips

- 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Poles

Define the vector of poles.

Settings

Default: `[0 -1]`

Tips

- 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-109 for the command-line information.

Gain

Define the vector of gains.

Settings

Default: `[1]`

Tips

- 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**.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, `1`, a real scalar, or a real 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

State Name (e.g., 'position')

Assign a unique name to each state.

Settings

Default: `'  '`

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 string, cell array, or structure.

Command-Line Information

See "Block-Specific Parameters" on page 6-109 for the command-line information.

Characteristics

Direct Feedthrough	Only if the lengths of the **Poles** and **Zeros** parameters are equal
Sample Time	Continuous

Scalar Expansion	No
States	Length of the **Poles** vector
Dimensionalized	No
Zero-Crossing Detection	No

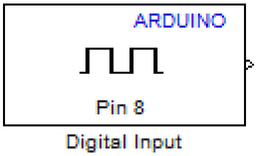
See Also

Discrete Zero-Pole

Arduino Digital Input

Purpose Get logical value of digital input pin

Library Simulink Support Package for Arduino® Hardware



Description

Get the logical value of a digital pin on the Arduino hardware:

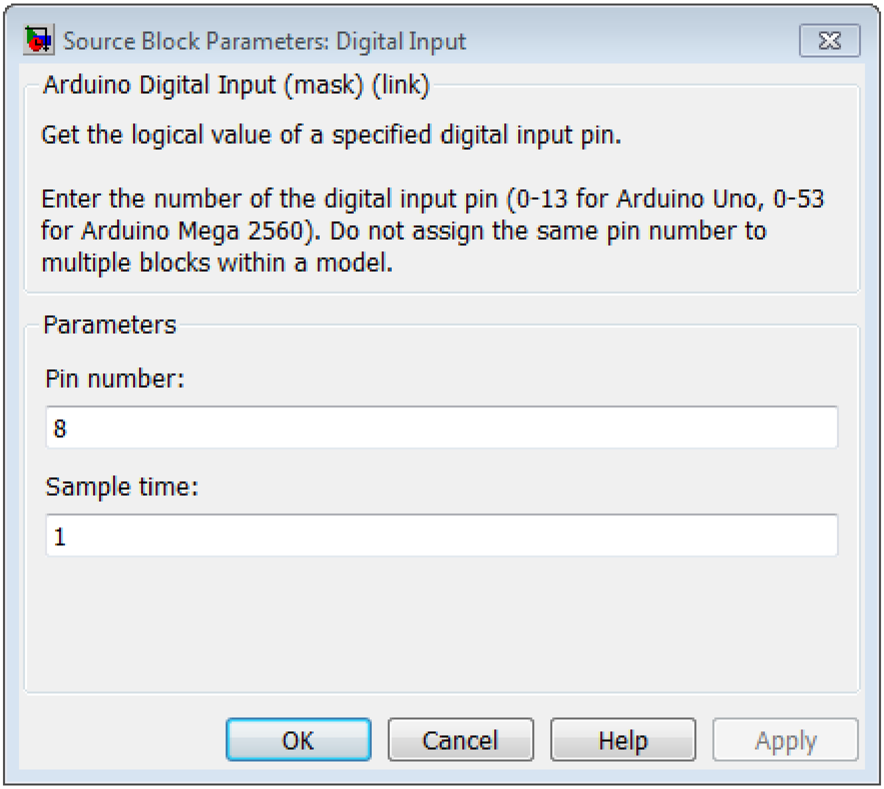
- If the logical value of the digital pin is LOW, the block outputs 0.
- If the logical value of the digital pin is HIGH, the block outputs 1.

The data type of the block output is uint8.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See "Block Produces Zeros in Simulation"

Warning

Only connect digital pins to devices that use 5 Volt TTL logic. For details, read the documentation for your Arduino hardware.



Arduino Digital Input (mask) (link)

Get the logical value of a specified digital input pin.

Enter the number of the digital input pin (0-13 for Arduino Uno, 0-53 for Arduino Mega 2560). Do not assign the same pin number to multiple blocks within a model.

Parameters

Pin number:

8

Sample time:

1

Dialog

Pin number

Enter the number of the digital pin.

Do not assign the same pin number to multiple blocks within the model.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a pin number from `0` to `53`.
- `Arduino Uno`, enter a pin number from `0` to `13`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

Sample time

Specify how often this block reads the pin value, in seconds. Enter a value greater than zero. This value defaults to a sample time of `1` second. The minimum value is `0.000001` second.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

Arduino Digital Output **|** “Install Support for Arduino Hardware” **|** “Detect and Fix Task Overruns on Arduino Hardware” **|**

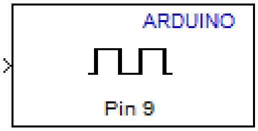
External Web Sites

- http://arduino.cc/en/Reference/DigitalRead

Arduino Digital Output

Purpose Set logical value of digital output pin

Library Simulink Support Package for Arduino Hardware



Digital Output

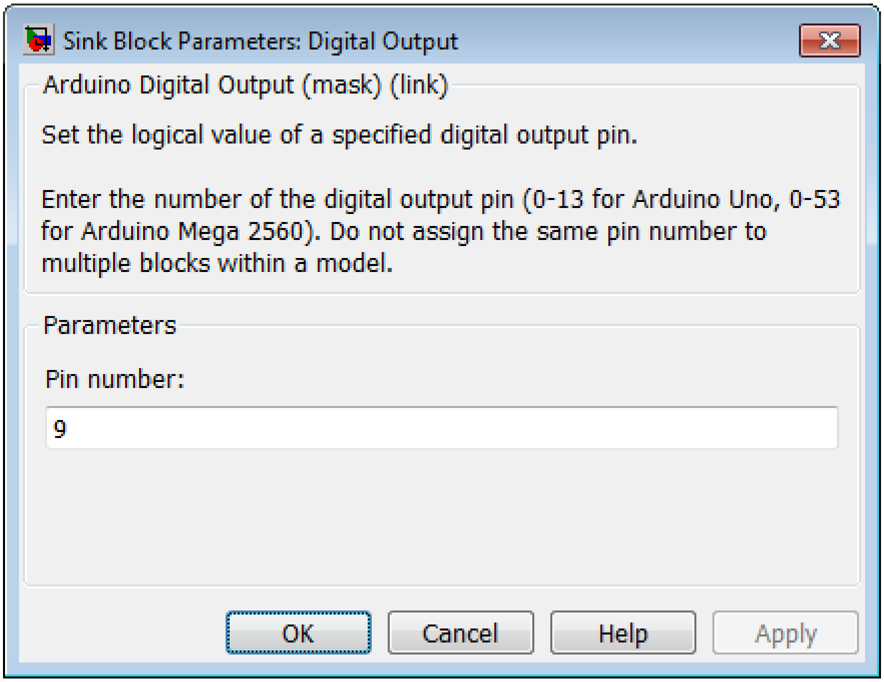
Description

Set the logical value of a digital pin on the Arduino hardware:

- Sending `1` to the block input sets the logical value of the digital pin HIGH to 5 V or 3.3 V, depending on the board voltage.
- Sending `0` to the block input sets the logical value of the digital pin LOW to 0 V.

The block input inherits the data type of the upstream block, and internally converts it to boolean.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See "Block Produces Zeros in Simulation".



Sink Block Parameters: Digital Output

Arduino Digital Output (mask) (link)

Set the logical value of a specified digital output pin.

Enter the number of the digital output pin (0-13 for Arduino Uno, 0-53 for Arduino Mega 2560). Do not assign the same pin number to multiple blocks within a model.

Parameters

Pin number: 9

Dialog

Pin number

Enter the number of the digital pin.

Do not assign the same pin number to multiple blocks within the model.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a pin number from `0` to `53`.
- `Arduino Uno`, enter a pin number from `0` to `13`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

See Also

Arduino Digital Input | "Install Support for Arduino Hardware" |

External Web Sites

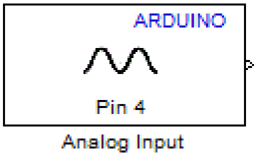
- http://arduino.cc/en/Reference/DigitalWrite

Purpose

Measure voltage of analog input pin

Library

Simulink Support Package for Arduino Hardware



Description

Measure the voltage of an analog pin relative to the analog input reference voltage on the Arduino hardware. Output the measurement as a 10-bit value that ranges from 0 to 1023.

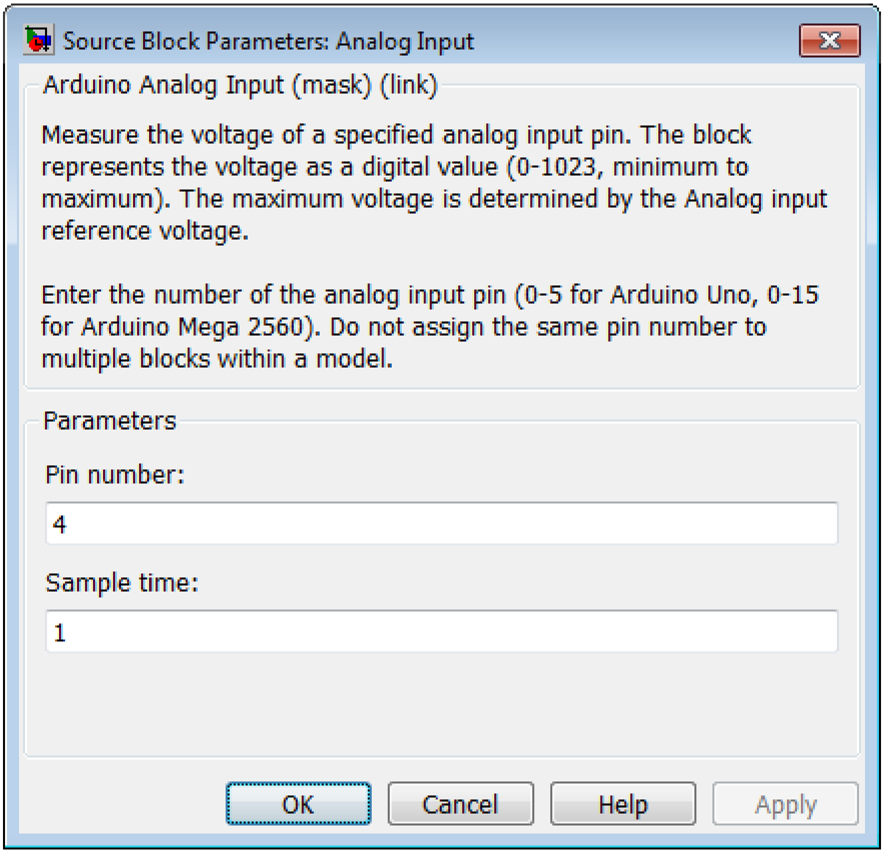
- If the measured voltage equals the ground voltage, the block outputs `0`.
- If the measured voltage equals the analog reference voltage, the block outputs `1023`.

The default value of the analog input reference voltage is 0 to 5 V. To change the **Analog input reference voltage** parameter in your model Configuration Parameters, select **Tools** > **Run on Target Hardware** > **Options...**.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”

Warning

The range of the voltage that can be applied to the analog input pin depends on the analog input reference voltage. For details, read the documentation for your Arduino hardware.



Arduino Analog Input (mask) (link)

Measure the voltage of a specified analog input pin. The block represents the voltage as a digital value (0-1023, minimum to maximum). The maximum voltage is determined by the Analog input reference voltage.

Enter the number of the analog input pin (0-5 for Arduino Uno, 0-15 for Arduino Mega 2560). Do not assign the same pin number to multiple blocks within a model.

Parameters

Pin number: 4

Sample time: 1

Dialog

Pin number

Enter the number of the analog input pin.

Do not assign the same pin number to multiple blocks within the model.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a pin number from `0` to `15`.

- `Arduino Uno`, enter a pin number from `0` to `5`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

Sample time

Specify how often this block reads the pin value. Enter a value greater than zero. This value defaults to a sample time of `1` second. The minimum value is `0.000001` second.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

Arduino PWM | “Analog input reference voltage” | “Install Support for Arduino Hardware” | “Detect and Fix Task Overruns on Arduino Hardware” |

External Web Sites

- http://arduino.cc/en/Reference/AnalogRead
- http://arduino.cc/en/Reference/AnalogReference

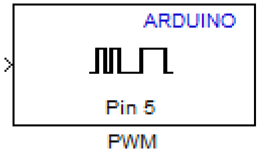
Arduino PWM

Purpose

Generate PWM waveform on analog output pin

Library

Simulink Support Package for Arduino Hardware



Description

Use pulse-width modulation (PWM) to change the duty-cycle of square-wave pulses output by a PWM pin on the Arduino hardware. PWM enables a digital output to provide a range of different power levels, similar to that of an analog output.

The value sent to the block input determines the width of the square wave, called *duty-cycle*, that the target hardware outputs on the specified PWM pin. The range of valid outputs is `0` to `255`.

For example:

- Sending the maximum value, `255`, to the block input produces 100% duty-cycle, which results in full power on a PWM pin.
- Sending the minimum value, `0`, to the block input produces 0% duty-cycle, which results in no power on a PWM pin.
- Sending an intermediate value to the block input produces a proportional duty-cycle and power output on a PWM pin. For example, sending 192 to the block input produces 75% duty cycle and power (192/256 = 0.75).
- Sending out-of-range values, such as `500` or `-500`, to the block input has the same effect as sending the maximum or minimum input values.

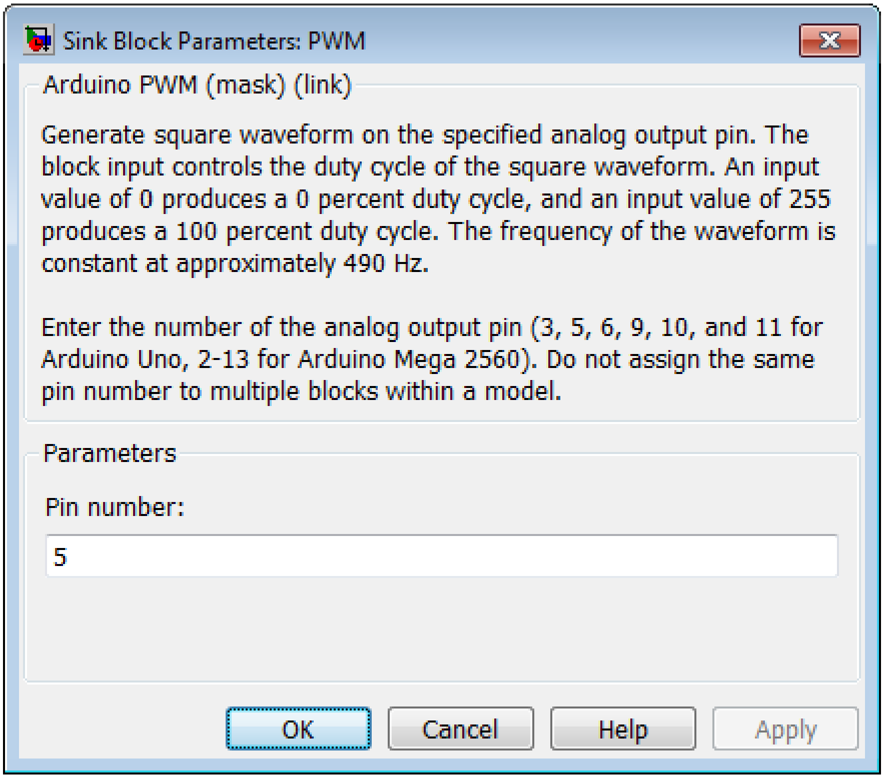
The frequency of the square wave is ~490 Hz.

The block input inherits the data type of the upstream block, and internally converts it to uint8.

Some limitations:

- With Arduino Uno hardware, the Arduino PWM block cannot use digital pins 9 or 10 when the model contains Servo blocks.
- With Arduino Mega 2560 hardware, the Arduino PWM block cannot use digital pins 11 or 12 when the model contains more than 12 Servo blocks.

Dialog



Sink Block Parameters: PWM

Arduino PWM (mask) (link)

Generate square waveform on the specified analog output pin. The block input controls the duty cycle of the square waveform. An input value of 0 produces a 0 percent duty cycle, and an input value of 255 produces a 100 percent duty cycle. The frequency of the waveform is constant at approximately 490 Hz.

Enter the number of the analog output pin (3, 5, 6, 9, 10, and 11 for Arduino Uno, 2-13 for Arduino Mega 2560). Do not assign the same pin number to multiple blocks within a model.

Parameters

Pin number: 5

Pin number
Enter the number of the PWM pin.

Do not assign the same pin number to multiple blocks within the model.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a pin number from `2` to `13`.
- `Arduino Uno`, enter one of the following pin numbers: `3`, `5`, `6`, `9`, `10`, and `11`. The board identifies these with a ~ symbol.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

See Also

"Install Support for Arduino Hardware" |

External Web Sites

- http://arduino.cc/en/Reference/AnalogWrite

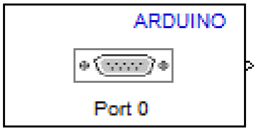
Arduino Serial Receive

Purpose

Get one byte of data from serial port

Library

Simulink Support Package for Arduino Hardware



Serial Receive

Description

Get one byte of data per sample period from the receive buffer of the specified serial port. For more information, see “Use Serial Communications with Arduino Hardware”.

The Serial Receive block has two block outputs, **Data** and **Status**.

When data is available:

- The **Data** block output emits data from the serial receive buffer.
- The **Status** block output emits `1`.

When data is not available:

- The **Data** block output emits `255`.
- The **Status** block output emits `0`.

The datatype of the **Data** block output is uint8.

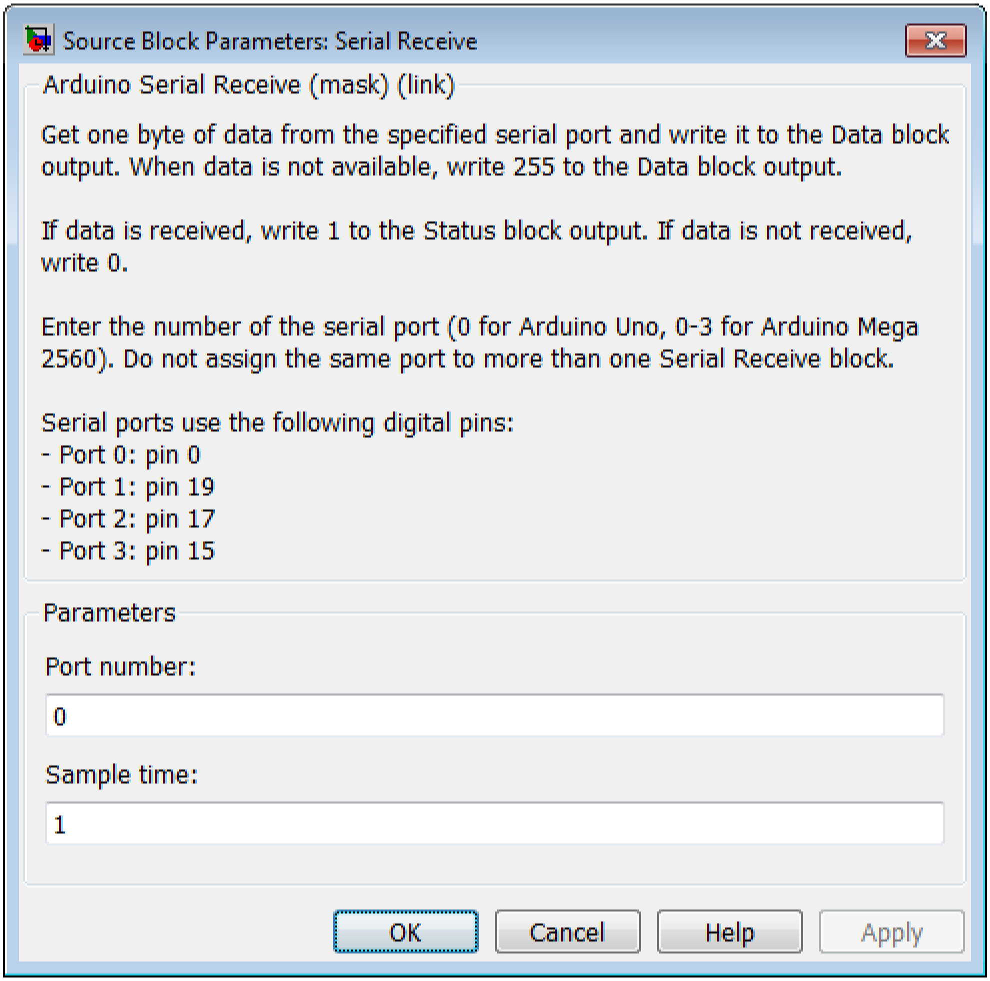
The datatype of the **Status** block output is int. You can use the Status block output to determine whether a value of `255` emitted by the **Data** block output is data, or an indication that no data was received.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”

If you use this block in models with the Standard Servo Read, Standard Servo Write, and Continuous Servo Write blocks, use longer sample times to avoid overruns.

Warning

Only connect serial port pins to devices that use 5 Volt TTL logic. Do not connect these pins to an RS-232 serial interface, such as the DE-9M connector on a computer, without limiting the voltage. The RS-232 standard allows higher voltages that can damage your hardware. For details, read the documentation for your Arduino hardware.



Arduino Serial Receive (mask) (link)

Get one byte of data from the specified serial port and write it to the Data block output. When data is not available, write 255 to the Data block output.

If data is received, write 1 to the Status block output. If data is not received, write 0.

Enter the number of the serial port (0 for Arduino Uno, 0-3 for Arduino Mega 2560). Do not assign the same port to more than one Serial Receive block.

Serial ports use the following digital pins:
- Port 0: pin 0
- Port 1: pin 19
- Port 2: pin 17
- Port 3: pin 15

Parameters

Port number: 0

Sample time: 1

Dialog

Port Number

Enter the number of the serial port.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a port number from `0` to `3`.

- `Arduino Uno`, enter `0`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

You can assign a Serial Transmit block and a Serial Receive block to the same serial port.

Do not assign more than one Serial Receive block to the same serial port.

Do not assign the pin numbers used by the serial port to other blocks within the model.

Serial port 0 is connected to the USB port through a converter. Do not use both serial port 0 and the USB port at the same time. For example, do not use serial port 0 if you intend to use External mode, because External mode requires the USB port.

Sample time

Specify how often this block reads the serial port buffer. Enter a value greater than zero. This value defaults to a sample time of `1` second. The minimum value is `0.000001` second.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

Arduino Serial Transmit | "Install Support for Arduino Hardware" | "Use Serial Communications with Arduino Hardware" | "Tune and Monitor Models Running on Arduino Mega 2560 Hardware" | "Detect and Fix Task Overruns on Arduino Hardware" |

External Web Sites

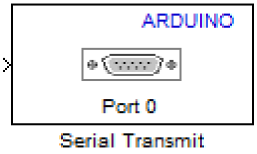
- http://arduino.cc/en/Serial/Read

Purpose

Send buffered data to serial port

Library

Simulink Support Package for Arduino Hardware



Description

Send buffered data to the specified serial port. For more information, see "Use Serial Communications with Arduino Hardware".

The Arduino Uno hardware has one serial port device, serial port 0, connected to the digital pins marked `TX 1` and `RX 0`. If you set the **Port number** parameter to `0`, this block transmits over the digital pin marked `TX 1`.

The block input accepts vector or scalar uint8 data. To convert a data source to uint8, use a Data Type Conversion block.

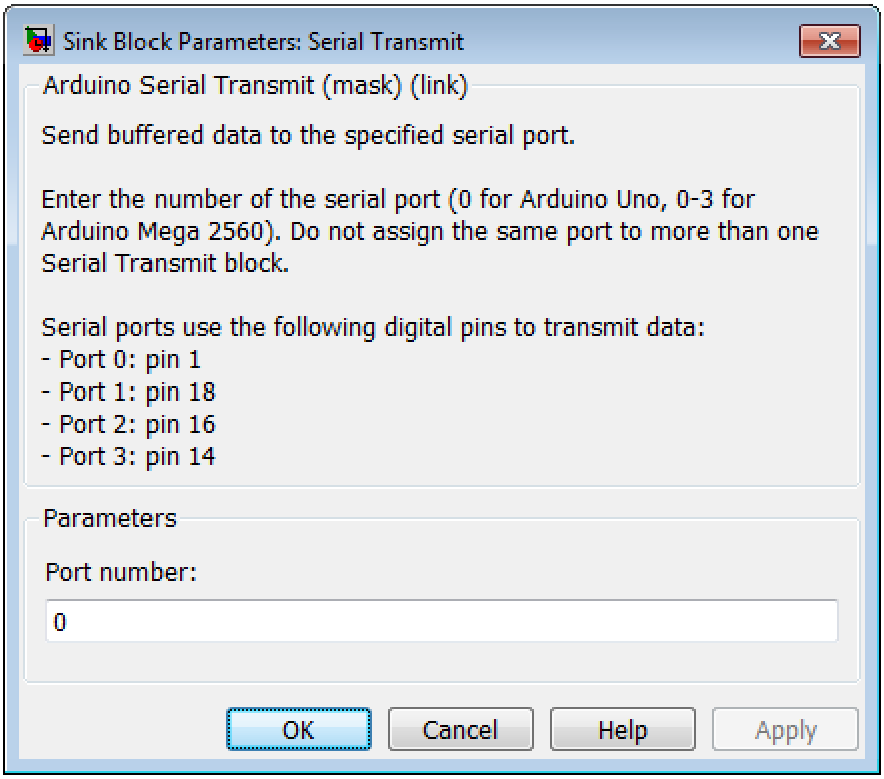
If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See "Block Produces Zeros in Simulation".

If you use this block in models with the Standard Servo Read, Standard Servo Write, and Continuous Servo Write blocks, use longer sample times to avoid overruns.

Warning

Only connect serial port pins to devices that use 5 Volt TTL logic. Do not connect these pins to an RS-232 serial interface, such as the DE-9M connector on a computer, without limiting the voltage. The RS-232 standard allows higher voltages that can damage your hardware. For details, read the documentation for your Arduino hardware.

Dialog



Arduino Serial Transmit (mask) (link)

Send buffered data to the specified serial port.

Enter the number of the serial port (0 for Arduino Uno, 0-3 for Arduino Mega 2560). Do not assign the same port to more than one Serial Transmit block.

Serial ports use the following digital pins to transmit data:
- Port 0: pin 1
- Port 1: pin 18
- Port 2: pin 16
- Port 3: pin 14

Parameters

Port number: 0

Port Number

Enter the number of the serial port.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a port number from `0` to `3`.
- `Arduino Uno`, enter `0`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

You can assign a Serial Transmit block and a Serial Receive block to the same serial port.

Do not assign multiple Serial Transmit blocks to the same serial port.

Do not assign the pin numbers used by the serial port to other blocks within the model.

Serial port 0 is connected to the USB port through a converter. Do not use both serial port 0 and the USB port at the same time. For example, do not use serial port 0 if you intend to use External mode, because External mode requires the USB port.

See Also

Arduino Serial Receive | "Install Support for Arduino Hardware" | "Use Serial Communications with Arduino Hardware" | "Tune and Monitor Models Running on Arduino Mega 2560 Hardware" |

External Web Sites

- http://arduino.cc/en/Serial/Write

Arduino Standard Servo Read

Purpose Get position of standard servo motor shaft in degrees

Library Simulink Support Package for Arduino Hardware

ARDUINO
Pin 2

Standard Servo Read

Description

Measure the angle of a standard servo motor shaft in degrees, from 0 to 180.

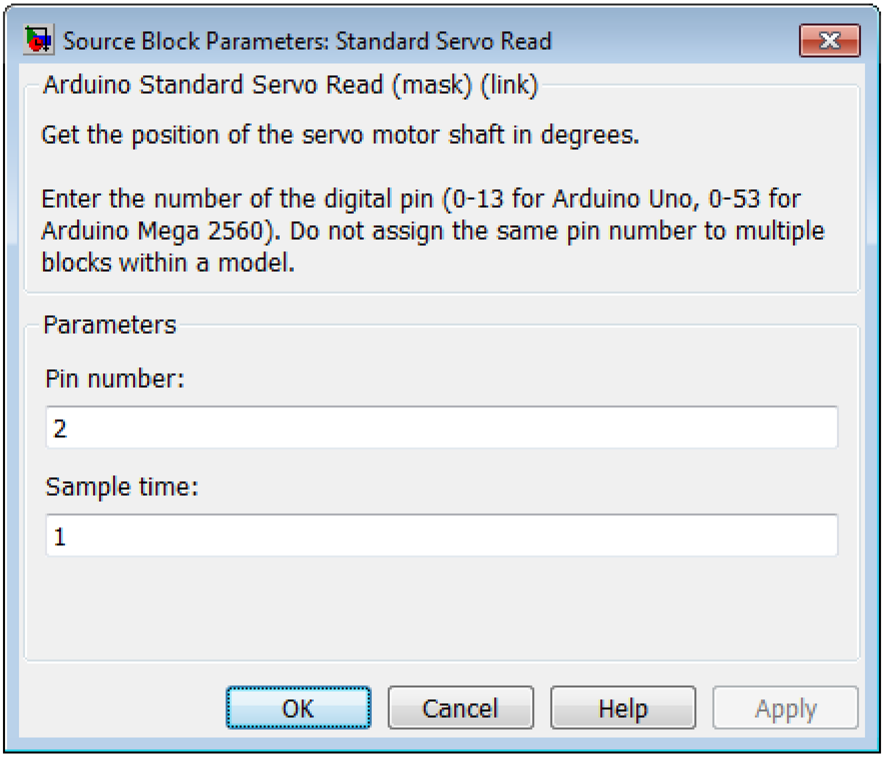
The data type of the block output is uint8.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Some limitations:

- Do not use Servo blocks with External mode.
- If you use this block in models with the Serial Receive and Serial Transmit blocks, use longer sample times to avoid overruns.
- The maximum number of Servo blocks per model is 12 for Arduino Uno hardware, and 48 for Arduino Mega 2560 hardware.
- With Arduino Uno hardware, the Arduino PWM block cannot use digital pins 9 or 10 when the model contains Servo blocks.
- With Arduino Mega 2560 hardware, the Arduino PWM block cannot use digital pins 11 or 12 when the model contains more than 12 Servo blocks.

Dialog



Source Block Parameters: Standard Servo Read

Arduino Standard Servo Read (mask) (link)

Get the position of the servo motor shaft in degrees.

Enter the number of the digital pin (0-13 for Arduino Uno, 0-53 for Arduino Mega 2560). Do not assign the same pin number to multiple blocks within a model.

Parameters

Pin number: 2

Sample time: 1

Pin number

Enter the number of the digital pin.

Do not assign the same pin number to multiple blocks within the model.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a pin number from `0` to `53`.
- `Arduino Uno`, enter a pin number from `0` to `13`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

Sample time

Specify how often this block reads the pin value. Enter a value greater than zero. This value defaults to a sample time of `1` seconds. The minimum value is `0.000001` second.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

Arduino Standard Servo Write | Arduino Continuous Servo Write | "Install Support for Arduino Hardware" | "Detect and Fix Task Overruns on Arduino Hardware" |

External Web Sites

- http://arduino.cc/en/Reference/ServoRead

Arduino Standard Servo Write

Purpose

Set shaft position of standard servo motor

Library

Simulink Support Package for Arduino Hardware

ARDUINO
Pin 9
Standard Servo Write

Description

Set the shaft position of a standard servo motor, from 0 to 180 degrees.

To rotate the servo shaft, send values from 0 to 180 to the block input.

Sending out-of-range values, such as `-5` or `200`, to the block input has the same effect as sending the maximum or minimum input values.

The block input inherits the data type of the upstream block, and internally converts it to uint8.

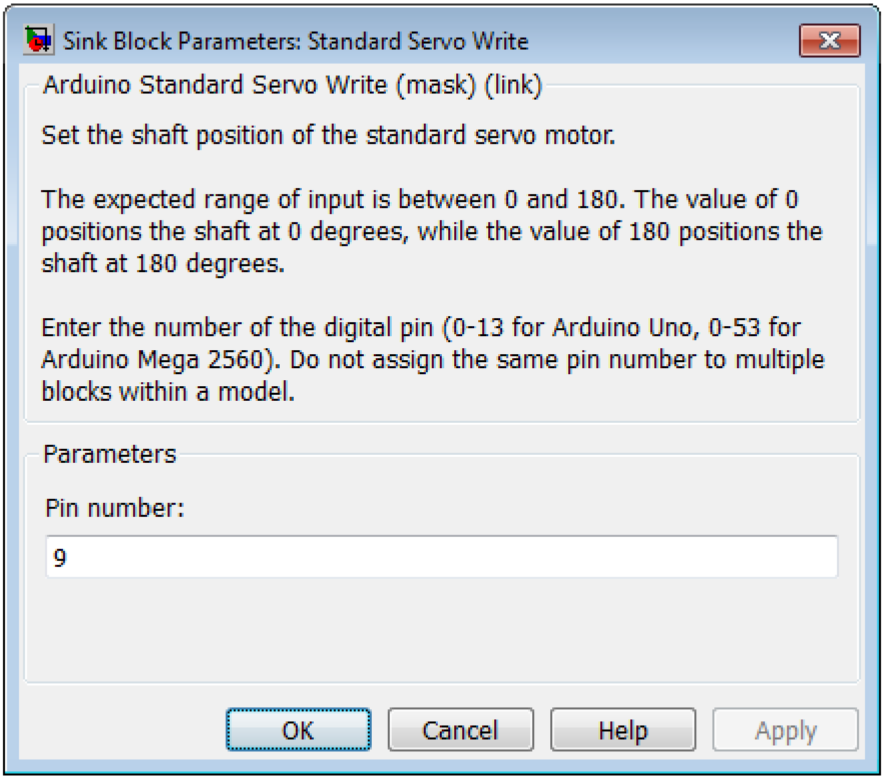
If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See "Block Produces Zeros in Simulation".

Some limitations:

- Do not use Servo blocks with External mode.
- If you use this block in models with the Serial Receive and Serial Transmit blocks, use longer sample times to avoid overruns.
- The maximum number of Servo blocks per model is 12 for Arduino Uno hardware, and 48 for Arduino Mega 2560 hardware.
- With Arduino Uno hardware, the Arduino PWM block cannot use digital pins 9 or 10 when the model contains Servo blocks.
- With Arduino Mega 2560 hardware, the Arduino PWM block cannot use digital pins 11 or 12 when the model contains more than 12 Servo blocks.

Arduino Standard Servo Write

Dialog



Arduino Standard Servo Write (mask) (link)

Set the shaft position of the standard servo motor.

The expected range of input is between 0 and 180. The value of 0 positions the shaft at 0 degrees, while the value of 180 positions the shaft at 180 degrees.

Enter the number of the digital pin (0-13 for Arduino Uno, 0-53 for Arduino Mega 2560). Do not assign the same pin number to multiple blocks within a model.

Parameters

Pin number:

9

Pin number

Enter the number of the digital pin.

Do not assign the same pin number to multiple blocks within the model.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a pin number from `0` to `53`.
- `Arduino Uno`, enter a pin number from `0` to `13`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

See Also

Arduino Standard Servo Read **|** Arduino Continuous Servo Write **|** "Install Support for Arduino Hardware" **|**

External Web Sites

- http://arduino.cc/en/Reference/ServoWrite

Arduino Continuous Servo Write

Purpose Set shaft speed of continuous rotation servo motor

Library Simulink Support Package for Arduino Hardware

Description

ARDUINO
Pin 2

Continuous Servo Write

Set the direction and speed of a continuous rotation servo motor:

- Sending -90 to the block input produces the maximum rate of rotation in one direction.
- Sending `90` to the block input produces the maximum rate of rotation in the opposite direction.
- Sending 0 to the block input stops the servo motor.
- Sending out-of-range values, such as `-95` or `200`, to the block input has the same effect as sending the maximum or minimum input values.

The characteristics of some motors cause them to continue rotating when the block input value is 0. In that case, you can experiment to find an offset value that stops the motor.

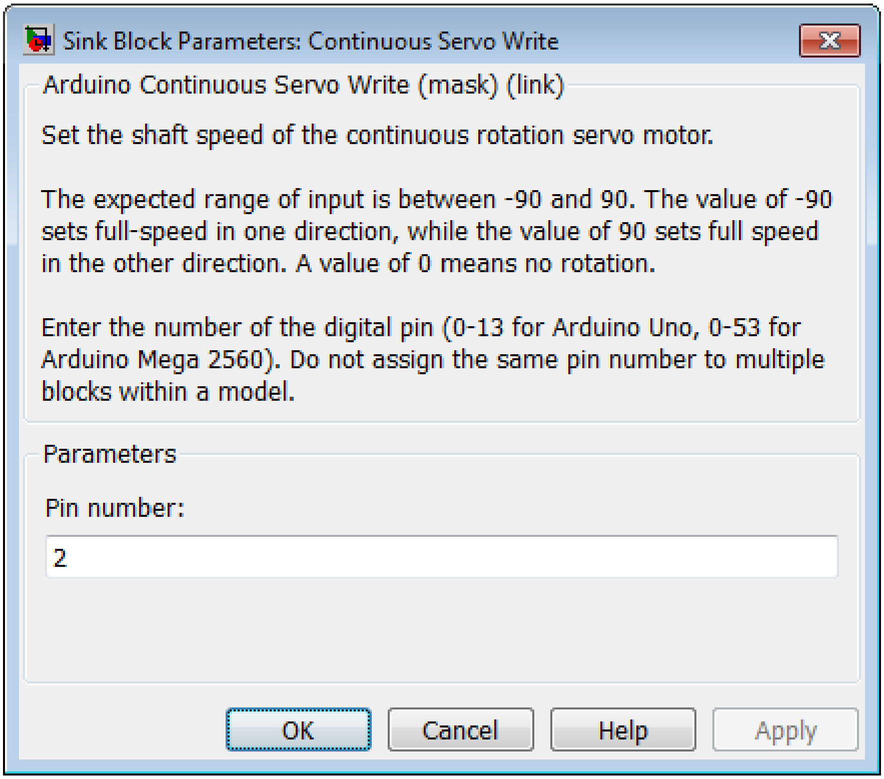
With Arduino Mega 2560 hardware, you can use External mode to determine the offset while your model is running on the hardware. See "Tune and Monitor Models Running on Arduino Mega 2560 Hardware".

The block input inherits the data type of the upstream block, and internally converts it to uint8.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See "Block Produces Zeros in Simulation".

Some limitations:

- Do not use Servo blocks with External mode.
- If you use this block in models with the Serial Receive and Serial Transmit blocks, use longer sample times to avoid overruns.
- The maximum number of Servo blocks per model is 12 for Arduino Uno hardware, and 48 for Arduino Mega 2560 hardware.
- With Arduino Uno hardware, the Arduino PWM block cannot use digital pins 9 or 10 when the model contains Servo blocks.
- With Arduino Mega 2560 hardware, the Arduino PWM block cannot use digital pins 11 or 12 when the model contains more than 12 Servo blocks.



Arduino Continuous Servo Write (mask) (link)

Set the shaft speed of the continuous rotation servo motor.

The expected range of input is between -90 and 90. The value of -90 sets full-speed in one direction, while the value of 90 sets full speed in the other direction. A value of 0 means no rotation.

Enter the number of the digital pin (0-13 for Arduino Uno, 0-53 for Arduino Mega 2560). Do not assign the same pin number to multiple blocks within a model.

Parameters

Pin number: 2

Dialog

Pin number

Enter the number of the digital pin.

Do not assign the same pin number to multiple blocks within the model.

If you set the **Target hardware** parameter to:

- `Arduino Mega 2560`, enter a pin number from `0` to `53`.
- `Arduino Uno`, enter a pin number from `0` to `13`.

Note To change the **Target hardware** parameter, select **Tools > Run on Target Hardware > Options**

See Also

Arduino Standard Servo Read **|** Arduino Standard Servo Write **|** "Install Support for Arduino Hardware" **|**

External Web Sites

- http://arduino.cc/en/Reference/ServoWrite

BeagleBoard ALSA Audio Capture

Purpose

Capture audio from sound card using ALSA

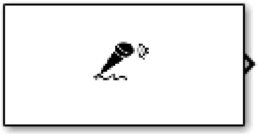
Library

Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

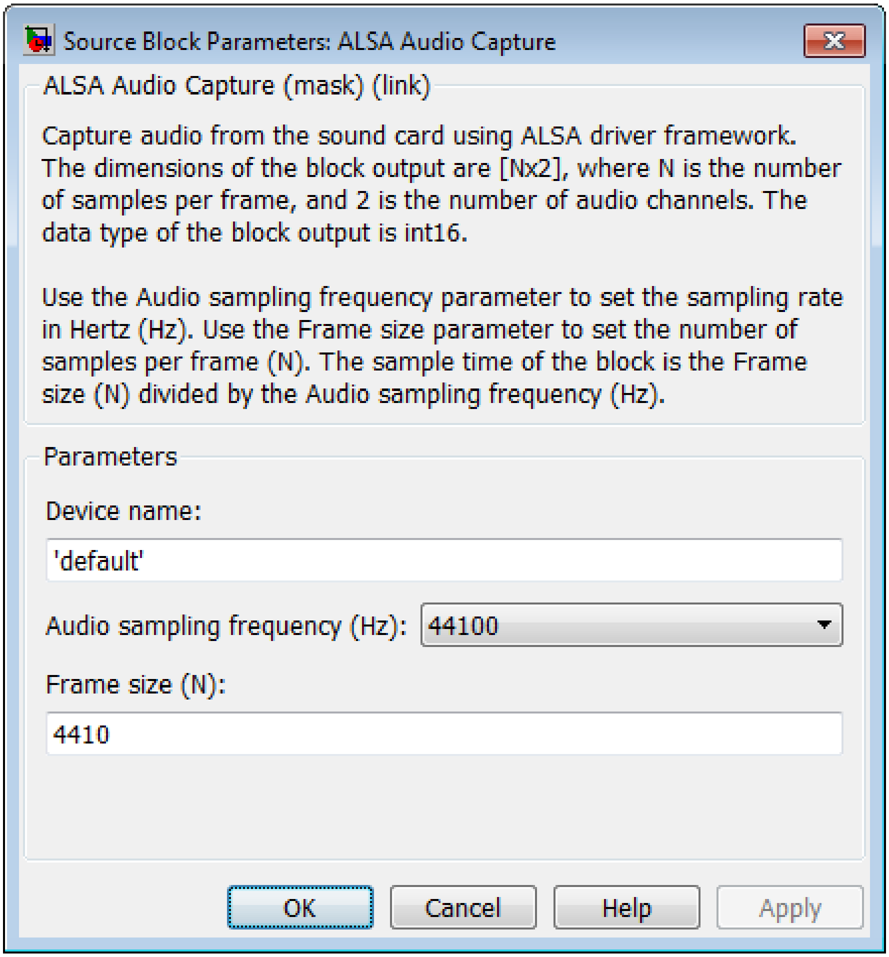
Simulink Support Package for Raspberry Pi Hardware



ALSA Audio Capture

Description

Capture the left and right audio channels from the AUDIO IN connector or a stereo sound device using the Advanced Linux Sound Architecture (ALSA) driver framework. The block output, **Out**, emits an N-by-2 frame of int16 values. The width of the frame, 2, corresponds to the left and right audio channels. To determine the sample rate of the block output, divide the **Frame size (N)** by the **Audio sampling frequency (Hz)**. For example, if **Frame size (N)** is 4410 and **Audio sampling frequency (Hz)** is 44100, then the sample rate is 0.1 seconds.



Source Block Parameters: ALSA Audio Capture

ALSA Audio Capture (mask) (link)

Capture audio from the sound card using ALSA driver framework. The dimensions of the block output are [Nx2], where N is the number of samples per frame, and 2 is the number of audio channels. The data type of the block output is int16.

Use the Audio sampling frequency parameter to set the sampling rate in Hertz (Hz). Use the Frame size parameter to set the number of samples per frame (N). The sample time of the block is the Frame size (N) divided by the Audio sampling frequency (Hz).

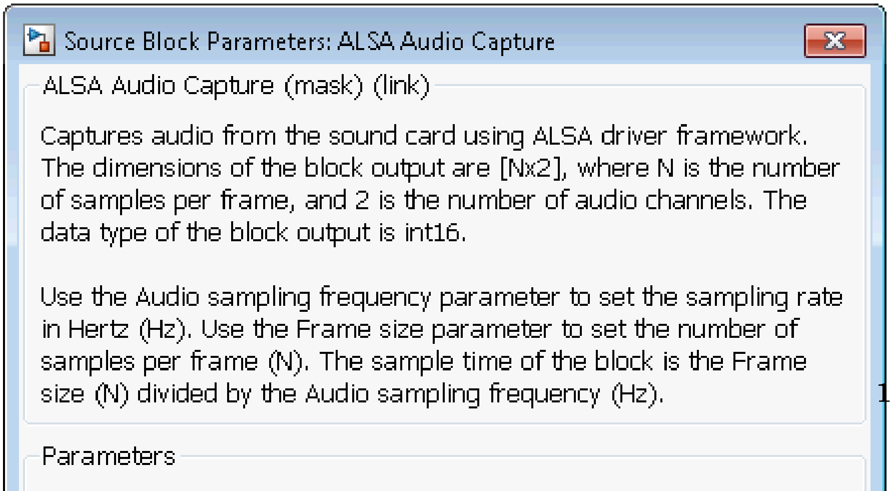
Parameters

Device name: 'default'

Audio sampling frequency (Hz): 44100

Frame size (N): 4410

Dialog



Source Block Parameters: ALSA Audio Capture

ALSA Audio Capture (mask) (link)

Captures audio from the sound card using ALSA driver framework. The dimensions of the block output are [Nx2], where N is the number of samples per frame, and 2 is the number of audio channels. The data type of the block output is int16.

Use the Audio sampling frequency parameter to set the sampling rate in Hertz (Hz). Use the Frame size parameter to set the number of samples per frame (N). The sample time of the block is the Frame size (N) divided by the Audio sampling frequency (Hz).

Parameters

Device name

For Raspberry Pi™ hardware, leave this value set to `'hw:1,0'`.

For other types of hardware, use the default ALSA device, or specify an audio input device by name.

To use the default sound device specified by the ALSA configuration file, leave this parameter set to `'default'`.

To find out which sound device the ALSA configuration file specifies, connect to the board and open the `/usr/share/alsa/alsa.conf` file. In the following example, the `alsa.conf` file specifies device 2 on card 0 as the default device:

```
pcm.!default {
 type hw
 card 0
 device 2
}
```

To specify an audio input device by name, connect to the board and open the `/proc/asound/cards` file. In the following example, the `cards` file names two devices, `VirMIDI` and `AudioPCI`:

```
$ cat /proc/asound/cards 0 [Dummy  ]: Dummy - Dummy
     Dummy 1

1 [VirMIDI ]: VirMIDI - VirMIDI
     Virtual MIDI Card 1

2 [AudioPCI ]: ENS1371 - Ensoniq AudioPCI
     Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

This parameter value defaults to `'default'`.

Audio sampling frequency (Hz)

Enter the sampling frequency of the ALSA Audio Capture (input) device.

By default, the sampling frequency of ALSA Audio Capture is the same as the sampling frequency of ALSA Audio Playback.

This parameter value defaults to `44100` Hz (44.1 kHz). The maximum rate equals the sampling rate of the audio capture device.

Frame size (N)

Enter the number of samples this block will output to your model. This parameter value defaults to `4410`.

See Also

Raspberry Pi ALSA Audio Playback | "Connect to Serial Port on BeagleBoard Hardware" | "Install Support for BeagleBoard Hardware" |

External Web Sites

- http://www.alsa-project.org
- http://en.wikipedia.org/wiki/Advanced_Linux_Sound_Architecture
- http://beagleboard.org/

BeagleBoard ALSA Audio Playback

Purpose

Send audio to sound card for playback using ALSA

Library

Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

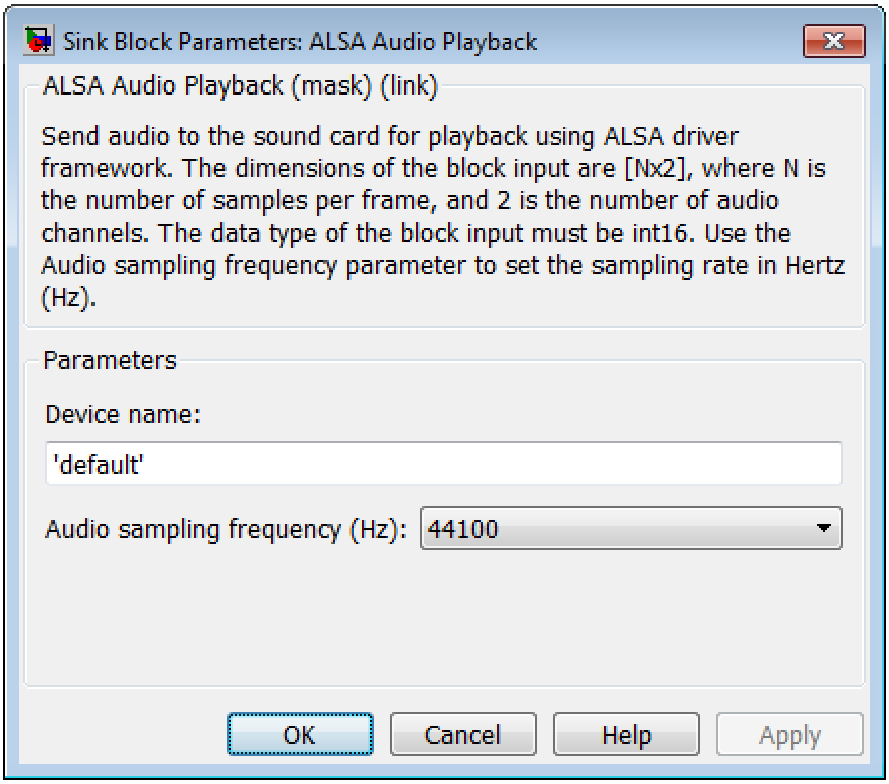
Simulink Support Package for Raspberry Pi Hardware

ALSA Audio Playback

Description

Send audio to the sound card for playback using the Advanced Linux Sound Architecture (ALSA) driver framework. The dimensions of the block input, **In**, are N-by-2, where N is the number of samples per frame, and 2 is the number of audio channels. The data type of the block input must be int16. Use the Audio sampling frequency parameter to set the sampling rate in Hertz (Hz).

Note When you use the software volume controls to lower the volume of one channel, crosstalk from the other channel may be audible.



Sink Block Parameters: ALSA Audio Playback

ALSA Audio Playback (mask) (link)

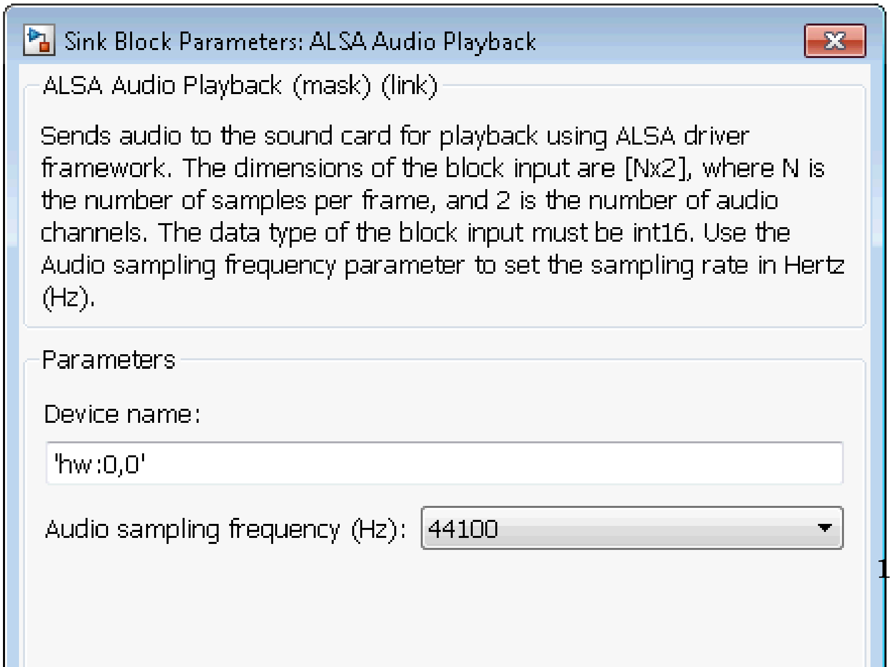
Send audio to the sound card for playback using ALSA driver framework. The dimensions of the block input are [Nx2], where N is the number of samples per frame, and 2 is the number of audio channels. The data type of the block input must be int16. Use the Audio sampling frequency parameter to set the sampling rate in Hertz (Hz).

Parameters

Device name: 'default'

Audio sampling frequency (Hz): 44100

Dialog



Sink Block Parameters: ALSA Audio Playback

ALSA Audio Playback (mask) (link)

Sends audio to the sound card for playback using ALSA driver framework. The dimensions of the block input are [Nx2], where N is the number of samples per frame, and 2 is the number of audio channels. The data type of the block input must be int16. Use the Audio sampling frequency parameter to set the sampling rate in Hertz (Hz).

Parameters

Device name: 'hw:0,0'

Audio sampling frequency (Hz): 44100

Device name

For Raspberry Pi hardware, leave this value set to `'hw:0,0'`.

For other types of hardware, use the default ALSA device, or specify an audio input device by name.

To use the default sound device specified by the ALSA configuration file, leave this parameter set to `'default'`.

To find out which sound device the ALSA configuration file specifies, connect to the board and open the `/usr/share/alsa/alsa.conf` file. In the following example, the `alsa.conf` file specifies device 2 on card 0 as the default device:

```
pcm.!default {
 type hw
 card 0
 device 2
}
```

To specify an audio input device by name, connect to the board and open the `/proc/asound/cards` file. In the following example, the `cards` file names two devices, `VirMIDI` and `AudioPCI`:

```
$ cat /proc/asound/cards 0 [Dummy  ]: Dummy - Dummy
     Dummy 1

1 [VirMIDI ]: VirMIDI - VirMIDI
     Virtual MIDI Card 1

2 [AudioPCI ]: ENS1371 - Ensoniq AudioPCI
     Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

This parameter value defaults to `'default'`.

Audio sampling frequency (Hz)

Enter the sample frequency of the ALSA Audio Playback (output) device.

By default, the sample frequency of ALSA Audio Playback is the same as the sample frequency of ALSA Audio Capture.

This parameter value defaults to `44100` Hz (44.1 kHz). The maximum rate equals the sampling rate of the audio capture device.

See Also

Raspberry Pi ALSA Audio Capture | “Connect to Serial Port on BeagleBoard Hardware” | “Install Support for BeagleBoard Hardware” |

External Web Sites

- http://www.alsa-project.org
- http://en.wikipedia.org/wiki/Advanced_Linux_Sound_Architecture
- http://beagleboard.org/
- http://pandaboard.org/

BeagleBoard eSpeak Text to Speech

Purpose

Convert text to speech for output on default audio device

Library

Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

Simulink Support Package for Raspberry Pi Hardware

eSpeak Text to Speech1

Description

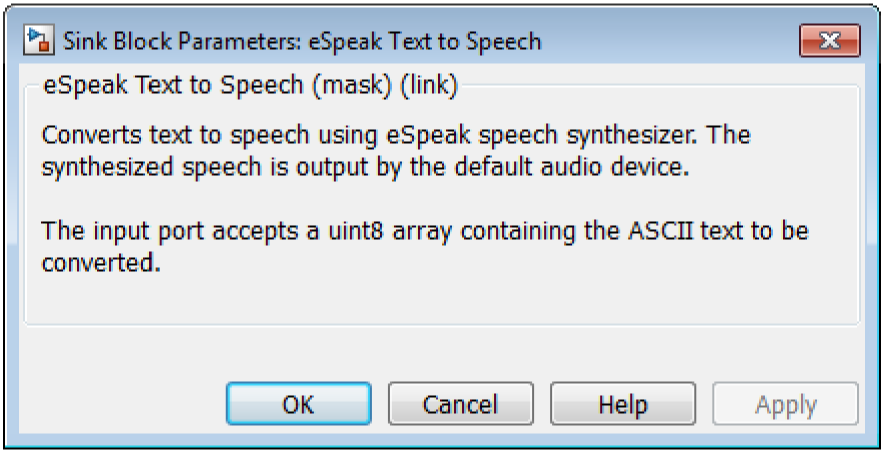
Convert text to speech using the eSpeak speech synthesizer. The synthesized speech is output by the default audio device.

The input port accepts a uint8 array containing the ASCII text to be converted: `uint8('`*`string`*`')`.

For example, you can send `uint8('Hello world')` or `uint8('1 2 3')` to the input. Sending `[1 2 3]` to the input does not work.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See "Block Produces Zeros in Simulation".

Dialog


Sink Block Parameters: eSpeak Text to Speech
eSpeak Text to Speech (mask) (link)
OK
Cancel
Help
Apply

Converts text to speech using eSpeak speech synthesizer. The synthesized speech is output by the default audio device.

The input port accepts a uint8 array containing the ASCII text to be converted.

The dialog does not contain any parameters.

See Also

"Install Support for BeagleBoard Hardware" **|**

External Web Sites

- http://beagleboard.org/
- http://pandaboard.org/
- http://en.wikipedia.org/wiki/Pull-up_resistor

BeagleBoard GPIO Read

Purpose Read logical value of GPIO pin configured as input

Library Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

Simulink Support Package for Raspberry Pi Hardware

GPIO 4
GPIO Read

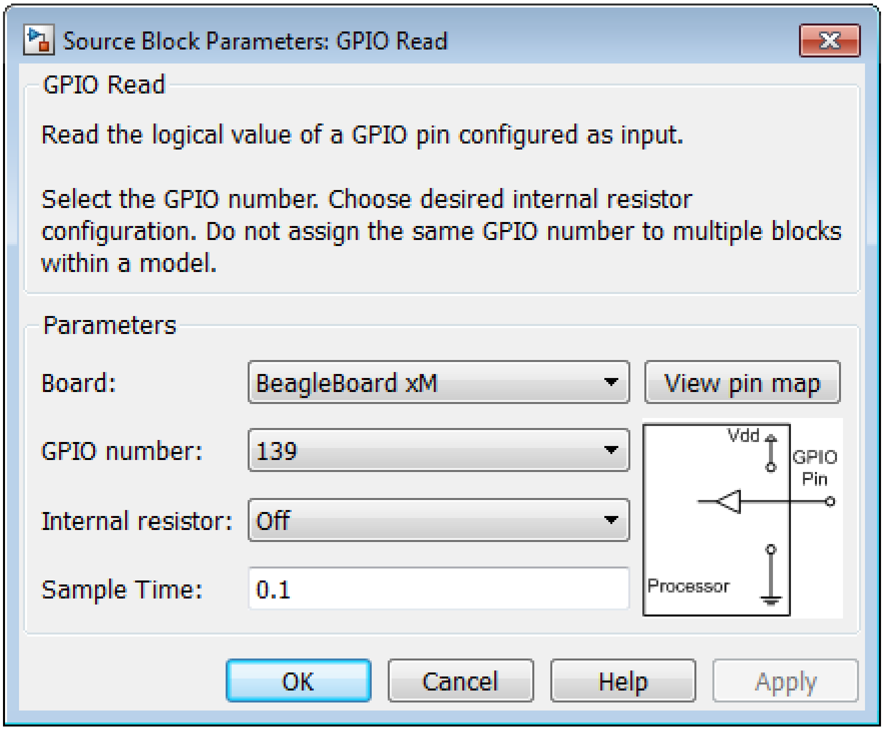
Description Use this block to:

- Create a physical input from a GPIO pin.
- Read the logical value of the GPIO pin at intervals that are determined by **Sample time**.
- Send the logical value to other blocks in the model as a stream of boolean values.
- The maximum input voltage of the GPIO pin is 1.8 V.

Warning

You can damage your target hardware by:

- Shorting a GPIO pin to ground.
- Shorting a GPIO pin with high voltage to another GPIO pin with low voltage.
- Connecting a GPIO pin to a voltage source greater than the maximum voltage.
- Exceeding the maximum current by driving multiple GPIO pins high simultaneously.



Source Block Parameters: GPIO Read

GPIO Read

Read the logical value of a GPIO pin configured as input.

Select the GPIO number. Choose desired internal resistor configuration. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

Board: BeagleBoard xM

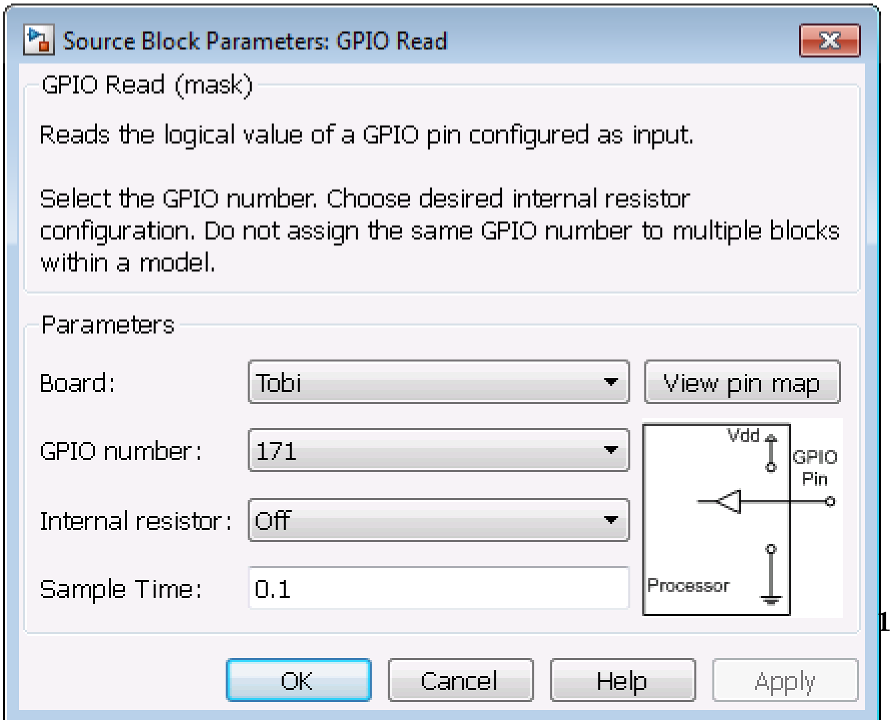
GPIO number: 139

Internal resistor: Off

Sample Time: 0.1

View pin map
Vdd
GPIO
Pin
Processor
OK
Cancel
Help
Apply

Dialog



Source Block Parameters: GPIO Read

GPIO Read (mask)

Reads the logical value of a GPIO pin configured as input.

Select the GPIO number. Choose desired internal resistor configuration. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

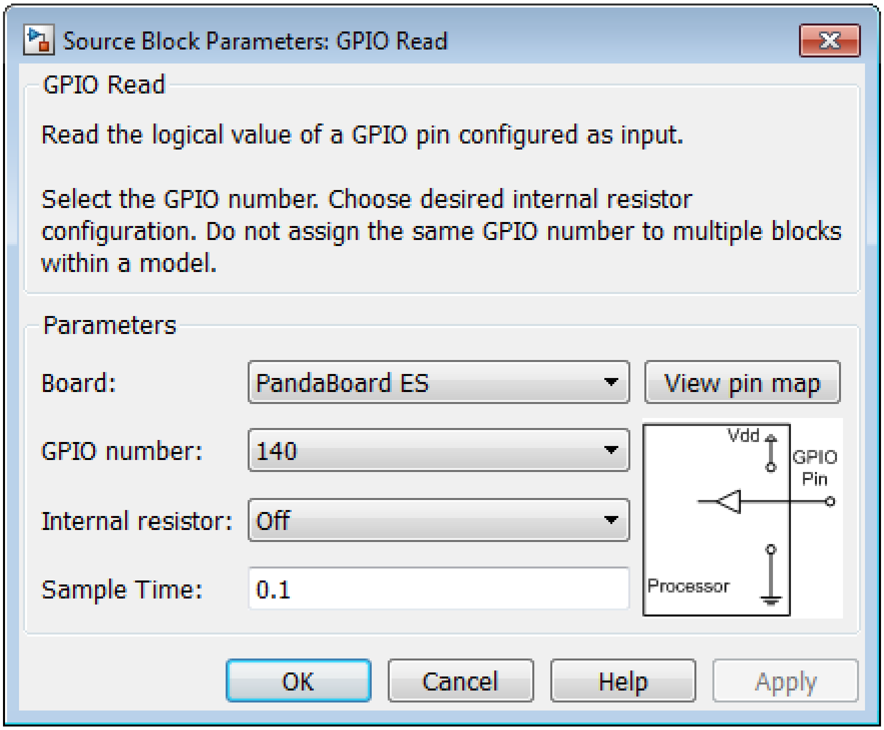
Board: Tobi

GPIO number: 171

Internal resistor: Off

Sample Time: 0.1

View pin map
Vdd
GPIO
Pin
Processor
OK
Cancel
Help
Apply



Source Block Parameters: GPIO Read

GPIO Read

Read the logical value of a GPIO pin configured as input.

Select the GPIO number. Choose desired internal resistor configuration. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

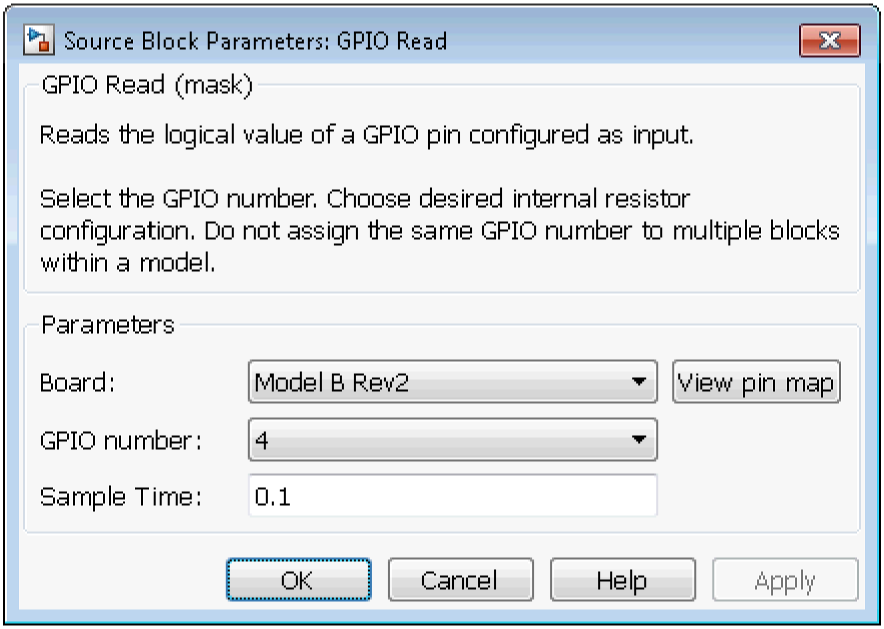
Board: PandaBoard ES

GPIO number: 140

Internal resistor: Off

Sample Time: 0.1

View pin map
Vdd
GPIO Pin
Processor
OK
Cancel
Help
Apply



GPIO Read (mask)

Reads the logical value of a GPIO pin configured as input.

Select the GPIO number. Choose desired internal resistor configuration. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

Board: Model B Rev2

GPIO number: 4

Sample Time: 0.1

Board

Select the specific board you are using. You can find this information on the target hardware, product packaging, or product documentation.

Changing **Board** updates the **GPIO number** choices and the **View pin map** diagram.

You can print the pin map to refer to it later.

GPIO number

After selecting the **Board**, select the number of the GPIO pin. Do not use the same GPIO number in multiple blocks within a model.

To see the locations of the GPIO pins on the target hardware, click the **View pin map** button.

Internal resistor

This parameter is not available for Raspberry Pi hardware.

Set the GPIO internal resistor to:

- `Off` — Do not use a pull-up or pull-down resistor.
- `Pull-up` — Use the internal pull-up resistor. When the physical input is inactive, disconnected, or has high impedance, the voltage of the GPIO pin goes high, settling at the voltage of *Vdd* on the processor.
- `Pull-down` — Use the internal pull-up resistor. When the physical input is inactive, disconnected, or has high impedance, the voltage of the GPIO pin goes low, settling at the voltage of *ground* on the processor.

Changing this parameter updates the diagram to the right of the parameter.

Sample time

Select the sample time for reading the state of the GPIO pin. This parameter value defaults to `1/10` second.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

“Install Support for BeagleBoard Hardware” **|** “Tune and Monitor Model Running on BeagleBoard Hardware” **|** “Install Support for Gumstix® Overo® Hardware” **|** “Tune and Monitor Model Running on Gumstix Overo Hardware” **|** “Install Support for PandaBoard Hardware” **|** “Tune and Monitor Model Running on PandaBoard Hardware” **|**

External Web Sites

- http://beagleboard.org/
- http://pandaboard.org/
- http://en.wikipedia.org/wiki/Pull-up_resistor

BeagleBoard GPIO Write

Purpose

Write logical value of GPIO pin configured as output

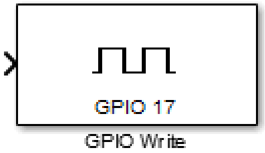
Library

Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

Simulink Support Package for Raspberry Pi Hardware



Description

Configure a GPIO pin on the target hardware to function as a physical output.

Send a logical value from your model to the GPIO pin. The sample time is determined by an upstream block.

The block input accepts boolean inputs.

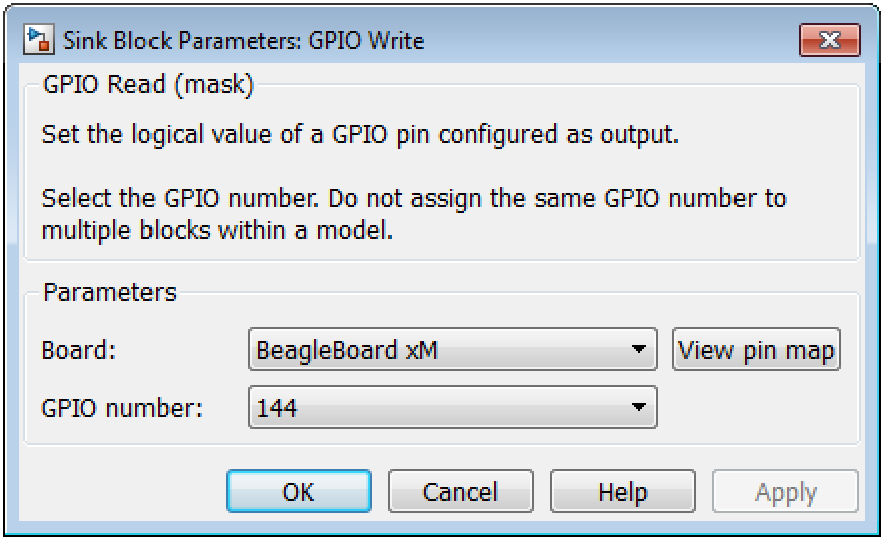
Do not exceed the maximum voltage of the GPIO pin.

Warning

You can damage your target hardware by:

- Shorting a GPIO pin to ground.
- Shorting a GPIO pin with high voltage to another GPIO pin with low voltage.
- Connecting a GPIO pin to a voltage source greater than the maximum voltage.
- Exceeding the maximum current by driving multiple GPIO pins high simultaneously.

Dialog



Sink Block Parameters: GPIO Write

GPIO Read (mask)

Set the logical value of a GPIO pin configured as output.

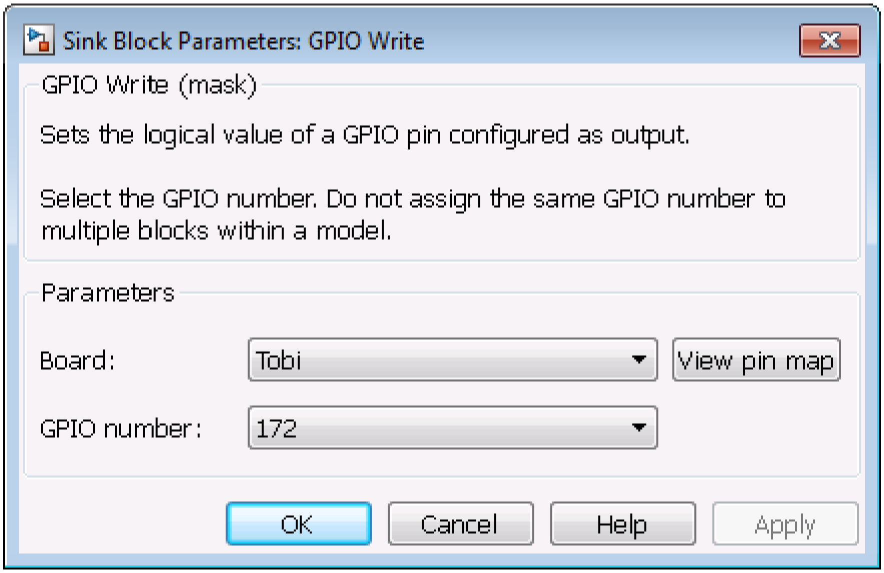
Select the GPIO number. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

Board: BeagleBoard xM

GPIO number: 144

View pin map
OK
Cancel
Help
Apply



Sink Block Parameters: GPIO Write

GPIO Write (mask)

Sets the logical value of a GPIO pin configured as output.

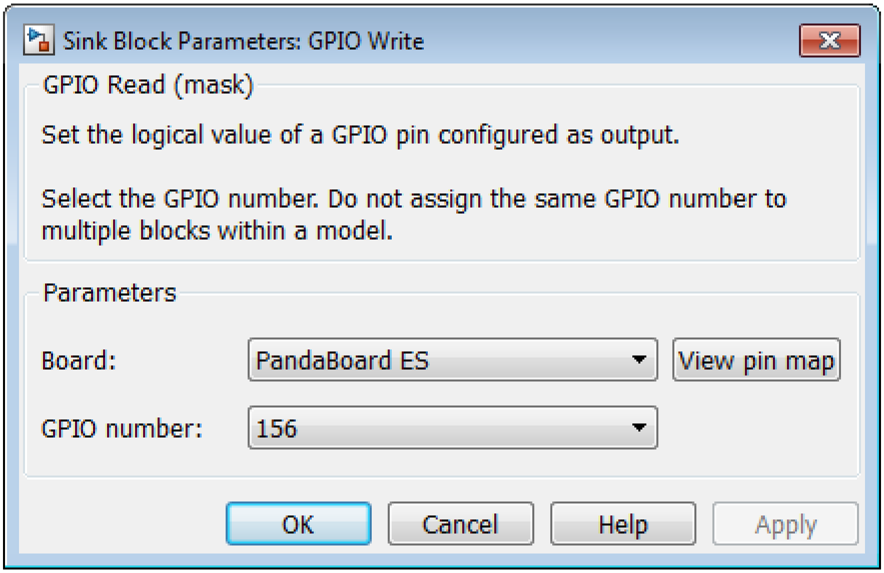
Select the GPIO number. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

Board: Tobi

GPIO number: 172

View pin map
OK
Cancel
Help
Apply


Sink Block Parameters: GPIO Write

GPIO Read (mask)

Set the logical value of a GPIO pin configured as output.

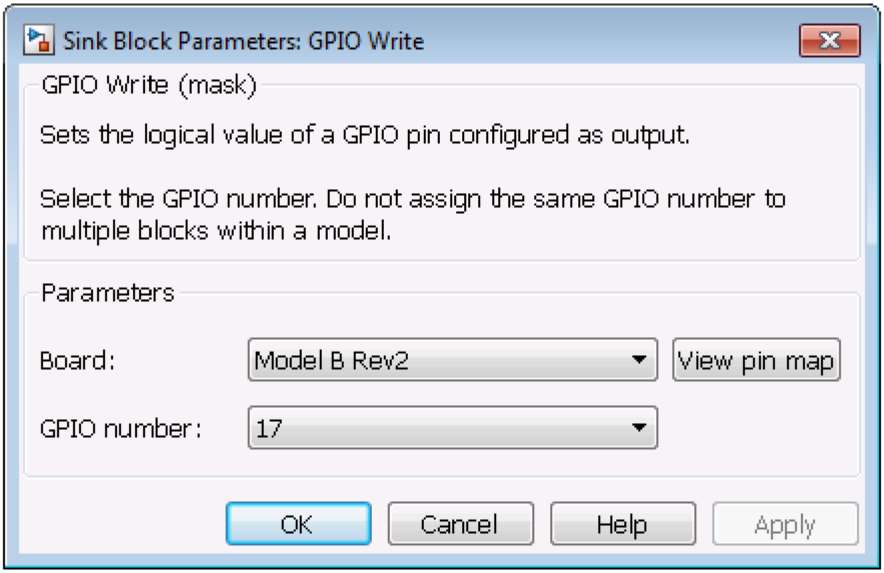
Select the GPIO number. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

Board: PandaBoard ES

GPIO number: 156

View pin map
OK
Cancel
Help
Apply


Sink Block Parameters: GPIO Write

GPIO Write (mask)

Sets the logical value of a GPIO pin configured as output.

Select the GPIO number. Do not assign the same GPIO number to multiple blocks within a model.

Parameters

Board: Model B Rev2

GPIO number: 17

View pin map
OK
Cancel
Help
Apply

Board

Select the specific board you are using.

You can find this information on the target hardware, product packaging, or product documentation.

Changing **Board** updates the **GPIO number** choices and the **View pin map** diagram.

You can print the pin map to refer to it later.

GPIO number

After selecting the **Board**, select the number of the GPIO pin. Do not use the same GPIO number in multiple blocks within a model.

To see the locations of the GPIO pins on the target hardware, click the **View pin map** button.

See Also

"Install Support for BeagleBoard Hardware" |

External Web Sites

- http://beagleboard.org/
- http://pandaboard.org/

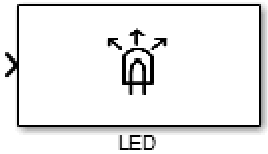
Purpose Illuminate built-in LED

Library Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

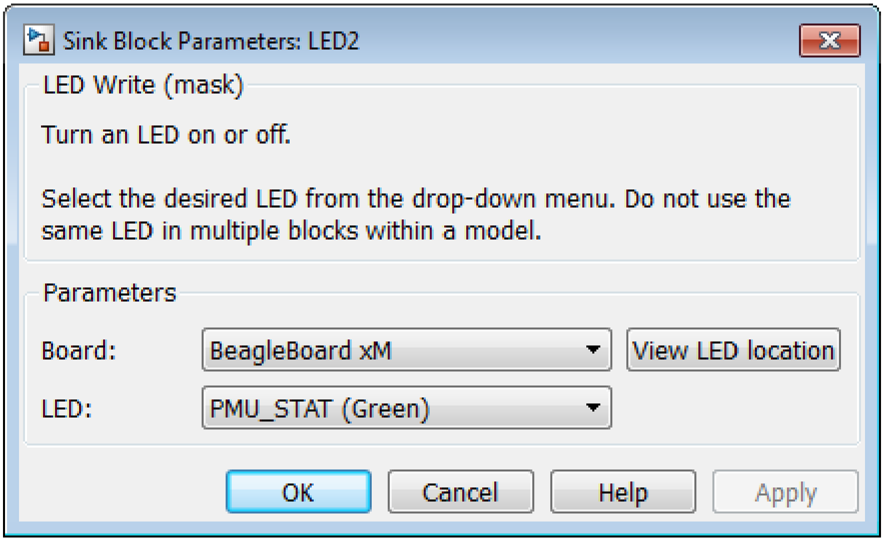
Simulink Support Package for Raspberry Pi Hardware



Description

Use this block to illuminate a built-in LED on the target hardware

Dialog



Sink Block Parameters: LED2

LED Write (mask)

Turn an LED on or off.

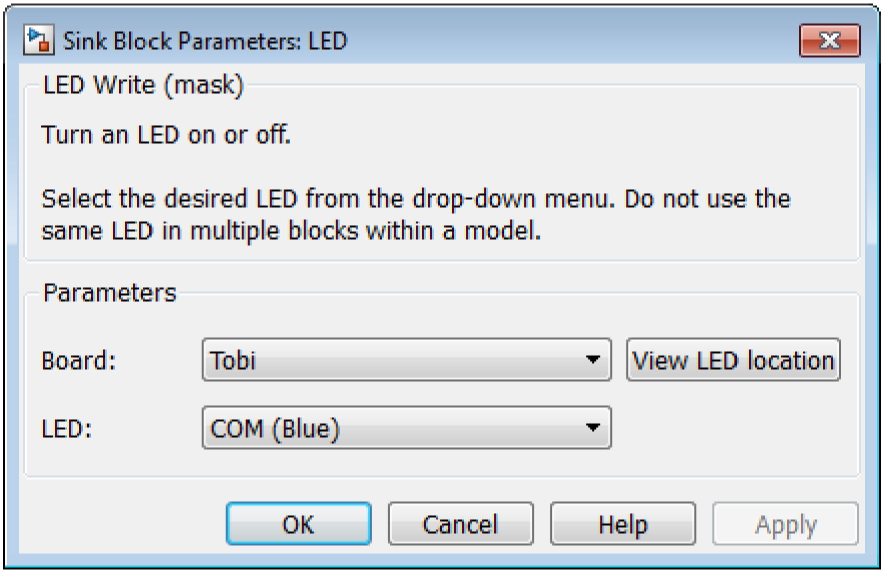
Select the desired LED from the drop-down menu. Do not use the same LED in multiple blocks within a model.

Parameters

Board: BeagleBoard xM

LED: PMU_STAT (Green)

View LED location
OK
Cancel
Help
Apply



Sink Block Parameters: LED

LED Write (mask)

Turn an LED on or off.

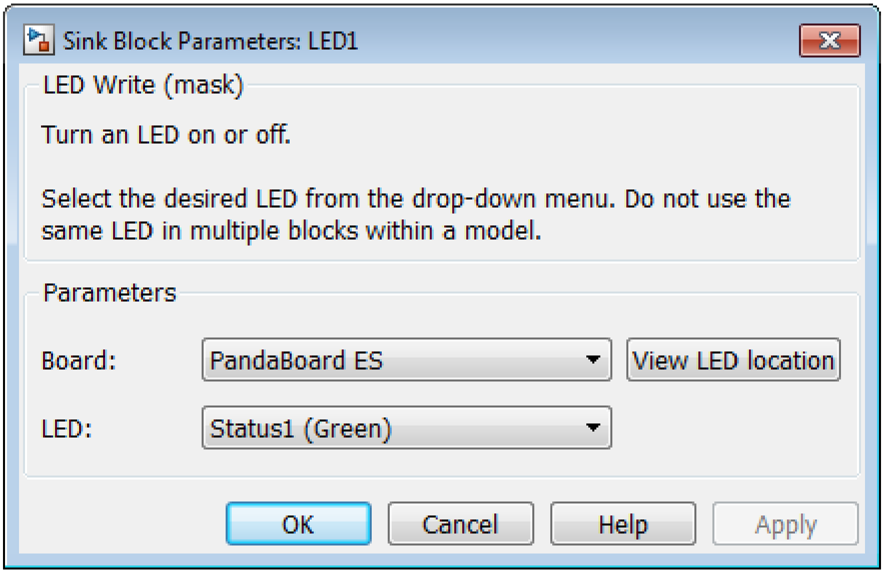
Select the desired LED from the drop-down menu. Do not use the same LED in multiple blocks within a model.

Parameters

Board: Tobi

LED: COM (Blue)

View LED location
OK
Cancel
Help
Apply


Sink Block Parameters: LED1

LED Write (mask)

Turn an LED on or off.

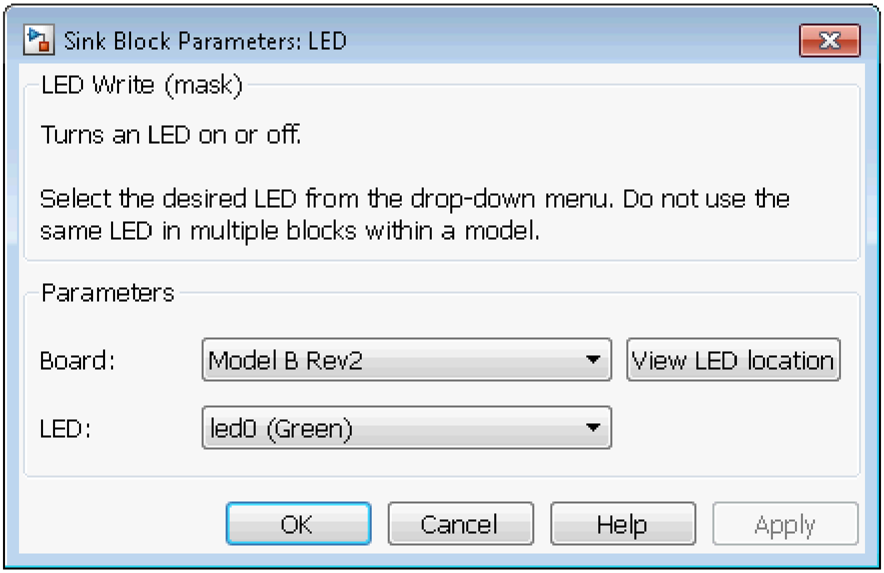
Select the desired LED from the drop-down menu. Do not use the same LED in multiple blocks within a model.

Parameters

Board: PandaBoard ES

LED: Status1 (Green)

View LED location
OK
Cancel
Help
Apply


Sink Block Parameters: LED

LED Write (mask)

Turns an LED on or off.

Select the desired LED from the drop-down menu. Do not use the same LED in multiple blocks within a model.

Parameters

Board: Model B Rev2

LED: led0 (Green)

View LED location
OK
Cancel
Help
Apply

Board

Select the specific board you are using. You can find this information on the target hardware, product packaging, or product documentation.

Changing **Board** updates the **LED** choices.

LED

After selecting the **Board**, select the LED. Do not use the same LED in multiple blocks within a model.

To see the locations of the LED on the target hardware, click the **View LED location** button. You can print the LED Locations diagram for reference.

See Also

"Install Support for BeagleBoard Hardware" |

External Web Sites

- http://beagleboard.org/
- http://pandaboard.org/

Purpose

Display video using SDL

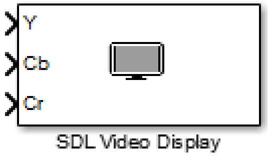
Library

Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

Simulink Support Package for Raspberry Pi Hardware

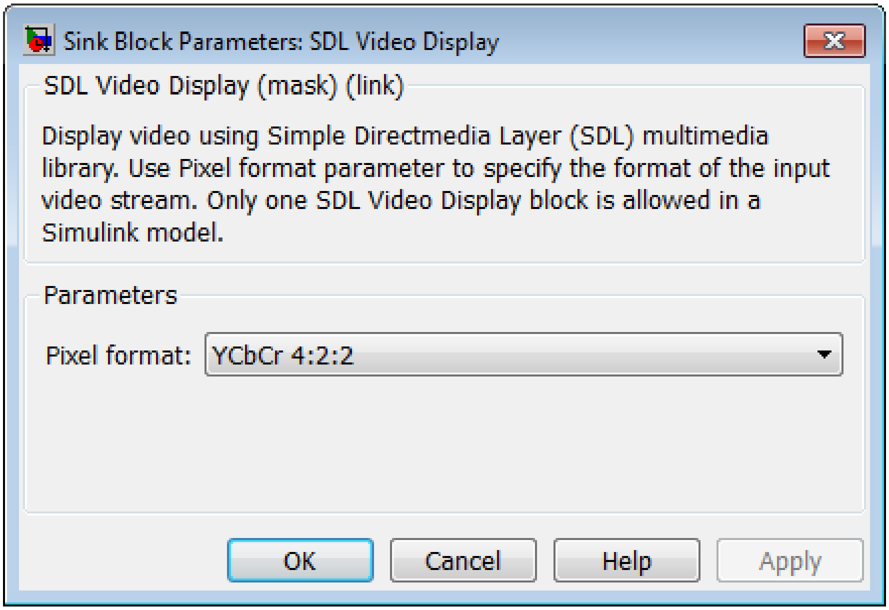


Description

Display video data using the Simple Directmedia Layer (SDL) multimedia library. A Simulink model can contain only one SDL Video Display block.

This block takes planar video data in YCbCr 4:2:2 or RGB formats.

Dialog



SDL Video Display (mask) (link)

Display video using Simple Directmedia Layer (SDL) multimedia library. Use Pixel format parameter to specify the format of the input video stream. Only one SDL Video Display block is allowed in a Simulink model.

Parameters

Pixel format: YCbCr 4:2:2

Pixel format

Specify the format of the input video stream: `YCbCr 4:2:2` or `RGB`.

`YCbCr 4:2:2` uses three channels to represent color image data for each pixel:

- Y is the luma component (essentially a grayscale signal).
- Cb is the blue-difference chroma component.
- Cr is the red-difference chroma component.

Cb and Cr are sampled at half the rate of Y. Therefore, if the dimensions of Y are M*N, the dimensions of Cb and Cr are each (M/2) by N.

`RGB` uses three 8-bit values to represent the red, green, and blue components of each pixel.

This parameter value defaults to `YCbCr 4:2:2`.

See Also

Raspberry Pi V4L2 Video Capture | “Install Support for BeagleBoard Hardware” |

External Web Sites

- http://en.wikipedia.org/wiki/Chroma_subsampling
- http://beagleboard.org/
- http://pandaboard.org/
- https://groups.google.com/forum/#!topic/beagleboard/x8MT086mKe0
- http://processors.wiki.ti.com/index.php/Beagle_Board_Challenge:_TurtleCam

BeagleBoard UDP Receive

Purpose

Receive UDP packets over IP network

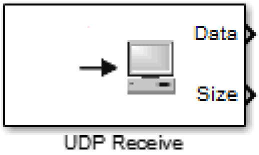
Library

Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

Simulink Support Package for Raspberry Pi Hardware



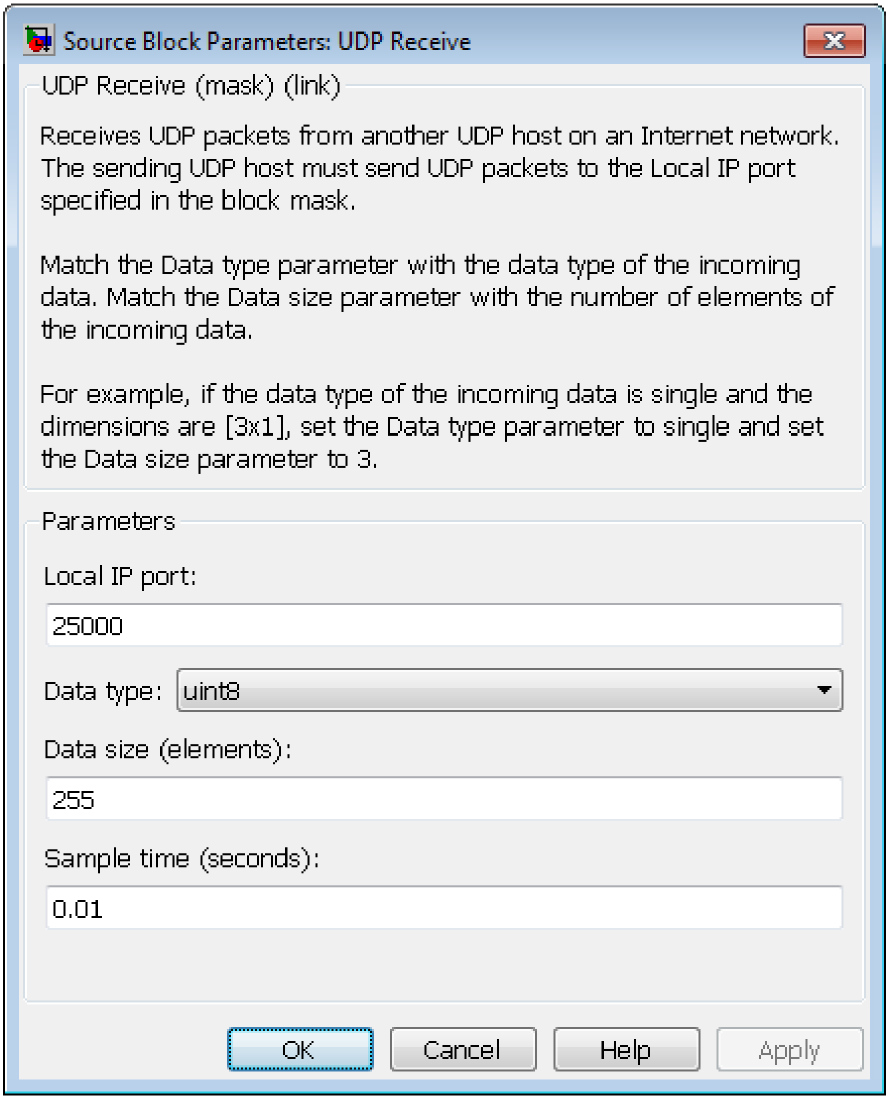
Description

Receive UDP packets from the local network. The block output, **data**, emits UDP packet data as a one-dimensional vector of a specified data type. The block output, **size**, emits the size the data in the UDP buffer.

While UDP Receive waits for new data to arrive, **size** emits a stream of zeros. When new data arrives, **size** changes to a non-zero value.

If you simulate a model that contains the UDP Receive block on your host computer (e.g., Simulation > Normal), and send UDP packets to that model from a board, the **data** output emits a disproportionately large number of zeros. This is because Simulink software simulates the model as a free-running application with a shorter period than the real-time application running on the board. In other words, the free-running simulation outputs zeros because it is waiting for the "slower" real-time application on the board to send data. If both applications were running on boards, this mismatch would not occur.

Note If you are having trouble using UDP to communicate with a computer, investigate whether antivirus or firewall software might be blocking UDP traffic. If so, try reconfiguring the software to allow UDP traffic for a specific IP port number.



UDP Receive (mask) (link)

Receives UDP packets from another UDP host on an Internet network. The sending UDP host must send UDP packets to the Local IP port specified in the block mask.

Match the Data type parameter with the data type of the incoming data. Match the Data size parameter with the number of elements of the incoming data.

For example, if the data type of the incoming data is single and the dimensions are [3x1], set the Data type parameter to single and set the Data size parameter to 3.

Parameters

Local IP port: 25000

Data type: uint8

Data size (elements): 255

Sample time (seconds): 0.01

Dialog

Local IP port

The number of the IP port on the local device, which can range from 1 to 65535. This value defaults to 25000. The “local device” is the board running the model.

Data type

Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Data size (elements)

Set the number of data elements in each UDP packet. Match the data size used to create the UDP packets. This value defaults to `255` elements.

Sample time (seconds)

Specify how often this block reads the port buffer. Enter a value greater than zero. This value defaults to a sample time of `0.01` seconds.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

Raspberry Pi UDP Send **|** "Install Support for BeagleBoard Hardware" **|** "Detect and Fix Task Overruns on BeagleBoard Hardware" **|**

External Web Sites

- http://en.wikipedia.org/wiki/User_Datagram_Protocol
- http://beagleboard.org/
- http://pandaboard.org/

BeagleBoard UDP Send

Purpose

Send UDP packets over IP network

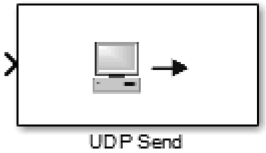
Library

Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

Simulink Support Package for Raspberry Pi Hardware

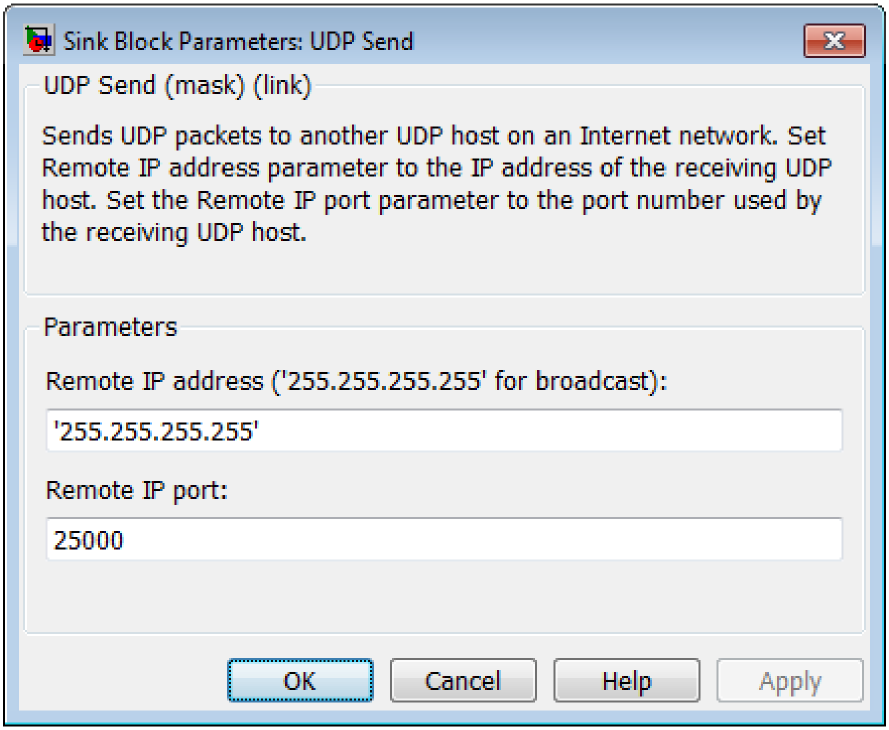


Description

Send a UDP packet to a network address identified by the remote IP address and remote IP port parameters.

Note If you are having trouble using UDP to communicate with a computer, investigate whether antivirus or firewall software might be blocking UDP traffic. If so, try reconfiguring the software to allow UDP traffic for a specific IP port number.

Dialog



UDP Send (mask) (link)

Sends UDP packets to another UDP host on an Internet network. Set Remote IP address parameter to the IP address of the receiving UDP host. Set the Remote IP port parameter to the port number used by the receiving UDP host.

Parameters

Remote IP address ('255.255.255.255' for broadcast): '255.255.255.255'

Remote IP port: 25000

Remote IP address ('255.255.255.255' for broadcast)

Enter the address or hostname of the IP node on the remote device.

To broadcast the UDP message to all devices, use the default parameter value, `'255.255.255.255'`. Use this approach sparingly to avoid creating unnecessary broadcast traffic. Often, IP routers are configured to drop broadcast UDP packets sent to `'255.255.255.255'`.

Remote IP port

The number of the IP port on the remote device, which can range from 1 to 65535. This value defaults to `25000`.

See Also

Raspberry Pi UDP Receive | “Install Support for BeagleBoard Hardware” |

Related Examples

- Get the IP Address of the BeagleBoard Hardware

External Web Sites

- http://en.wikipedia.org/wiki/User_Datagram_Protocol
- http://beagleboard.org/
- http://pandaboard.org/

BeagleBoard V4L2 Video Capture

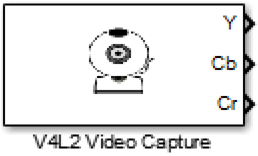
Purpose Capture video from USB camera using V4L2

Library Simulink Support Package for BeagleBoard Hardware

Simulink Support Package for Gumstix Overo Hardware

Simulink Support Package for PandaBoard Hardware

Simulink Support Package for Raspberry Pi Hardware



Description

Capture live video from a USB video camera, using the V4L2 (Video for Linux Two API) driver framework. The output is in row major format. During simulation, the model outputs a moving colorbar image.

The Ubuntu Linux image used on the target hardware supports video capture from USB cameras listed as Universal Video Class (UVC). In addition to being a UVC class camera, the camera itself must support data acquisition in YUYV mode.

The following cameras have been tested:

- Logitech QuickCam Pro 9000
- Logitech QuickCam Pro 3000
- Logitech Webcam C600
- Logitech HD Webcam C310

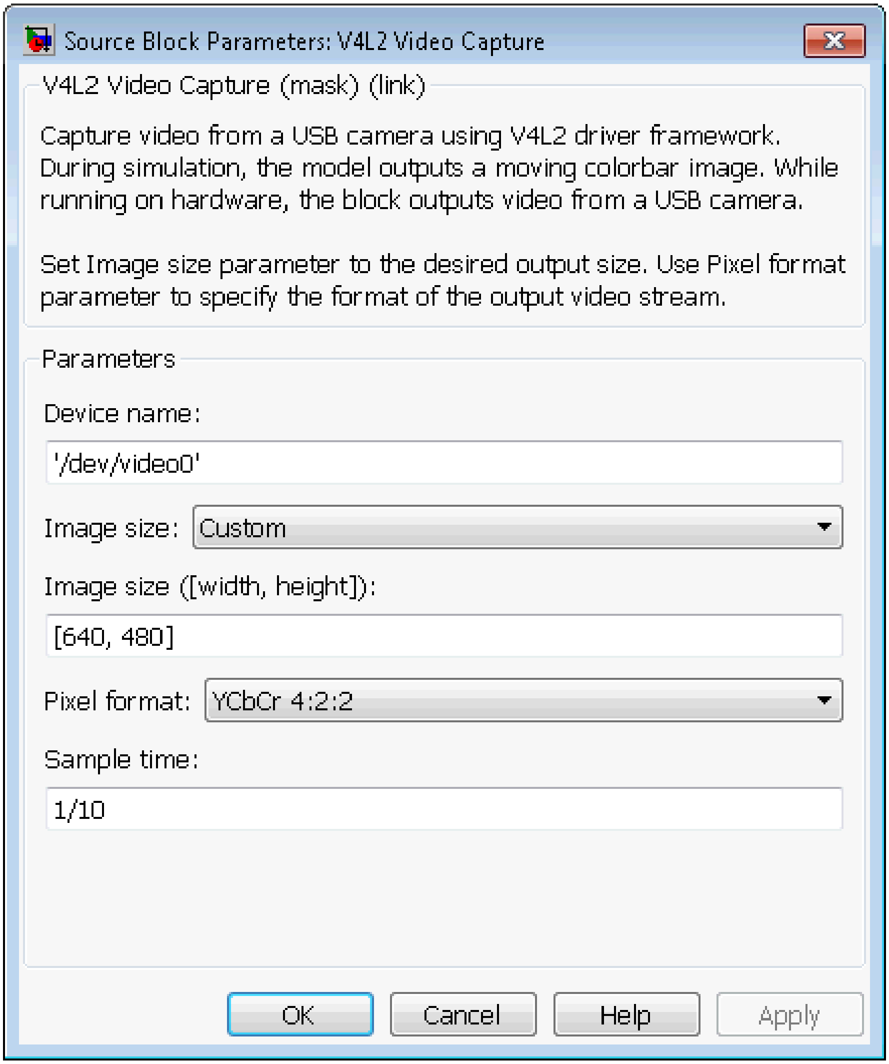
For a list of Logitech webcams that support UVC, see http://logitech-en-amr.custhelp.com/app/answers/detail/a_id/6471. For a list of UVC cameras supported by Ubuntu, see http://www.ideasonboard.org/uvc/.

For BeagleBoard hardware only: If you connect any USB camera to a USB Host port on the board, the video output contains artifacts that look like small dark scratches. This is a known hardware issue, and is independent of the Linux distribution. To work around this issue, connect the camera to the BeagleBoard's USB OTG port using a "Micro USB OTG to USB 2.0 Adapter". The USB OTG port is labelled "OTG" and is located next to the BeagleBoard's "5V" or "5VDC" connector. This issue is discussed at:

- http://processors.wiki.ti.com/index.php/Beagle_Board_Challenge:_TurtleCam
- https://groups.google.com/forum/#!topic/beagleboard/x8MT086mKe0

Dialog

Device Configuration



Source Block Parameters: V4L2 Video Capture

V4L2 Video Capture (mask) (link)

Capture video from a USB camera using V4L2 driver framework. During simulation, the model outputs a moving colorbar image. While running on hardware, the block outputs video from a USB camera.

Set Image size parameter to the desired output size. Use Pixel format parameter to specify the format of the output video stream.

Parameters

Device name: '/dev/video0'

Image size: Custom

Image size ([width, height]): [640, 480]

Pixel format: YCbCr 4:2:2

Sample time: 1/10

OK
Cancel
Help
Apply

Device name

Enter the path and name of the video device. This parameter value defaults to `'/dev/video0'`.

The Linux kernel creates a video device file when you connect a supported USB video camera to the board. By default, the Linux kernel supports all USB video class (UVC) devices.

To see the list of video device files, open a command line session with the board and enter: `ls -al /dev/video*`.

Image size ([width, height])

Specify the width in pixels and height in lines of the image to capture.

This parameter value defaults to `[640, 480]`.

Pixel format

Select the video format of the video device, `RGB` or `YCbCr 4:2:2`.

`RGB` represents the red, green, and blue components of a pixel using an 8-bit value. RGB color space is device-dependent.

`YCbCr 4:2:2` uses three channels to represent color image data for each pixel:

- `Y` is the luma component (essentially a grayscale signal).
- `Cb` is the blue-difference chroma component.
- `Cr` is the red-difference chroma component.

The Cb and Cr components are sampled at half the sample rate of Y. See http://en.wikipedia.org/wiki/Chroma_subsampling.

This parameter value defaults to `YCbCr 4:2:2`.

Sample time

Select the sample time of the video device. This parameter value defaults to `1/10`

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

Raspberry Pi SDL Video Display | "Install Support for BeagleBoard Hardware" | "Detect and Fix Task Overruns on BeagleBoard Hardware" |

External Web Sites

- http://en.wikipedia.org/wiki/Chroma_subsampling
- http://logitech-en-amr.custhelp.com/app/answers/detail/a_id/6471
- http://www.ideasonboard.org/uvc/
- http://beagleboard.org/
- http://pandaboard.org/
- http://processors.wiki.ti.com/index.php/Beagle_Board_Challenge:_TurtleCam
- https://groups.google.com/forum/#!topic/beagleboard/x8MT086mKe0

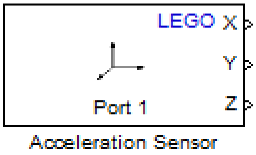
LEGO MINDSTORMS NXT Acceleration Sensor

Purpose

Measure acceleration along three axes

Library

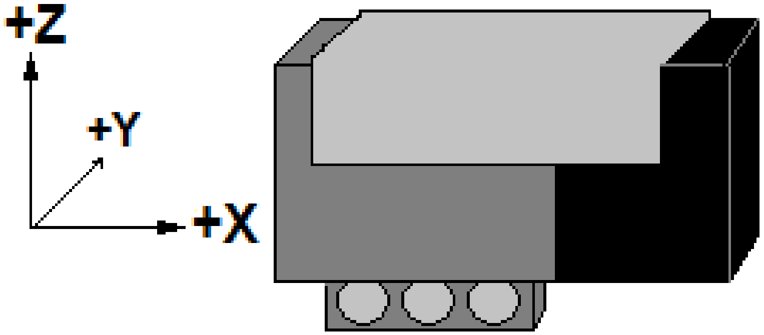
Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

Measure the linear acceleration along the X, Y, and Z axes of the NAC1040 HiTechnic NXT Acceleration / Tilt Sensor.

The axes of the sensor are oriented as shown here.



For each axis, the sensor measures -2 g to +2 g of acceleration. The sensor outputs approximately 200 counts per g.

Note The SI equivalent of 1 g is 9.8 meters per second squared.

You can use this sensor to:

- Determine the orientation of the sensor, for example in a self-levelling robot.
- Measure small and moderate acceleration forces, such as road surface vibration and acceleration in a vehicle.
- Detect larger acceleration events, such as a collision.

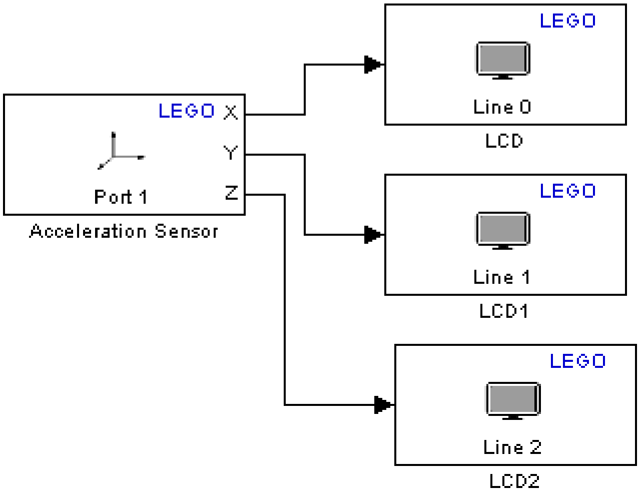
If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Use the Acceleration Sensor

View the changing outputs of an acceleration sensor as you change the orientation of the sensor.

Open the `legonxtlib` block library and copy the blocks shown to a new model.

Connect the Acceleration Sensor block to the LCD blocks, as shown.



Double click each LCD block to open the block masks for each block.

Use the **Display at line** parameters to select a different line number for each block.

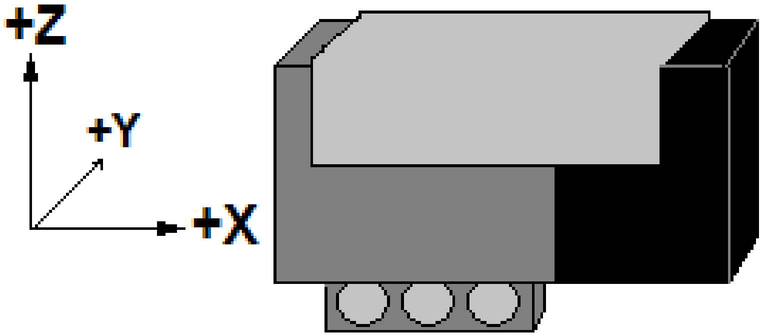
Use the **Display label** parameters to add an `X`, `Y`, or `Z` label for each line.

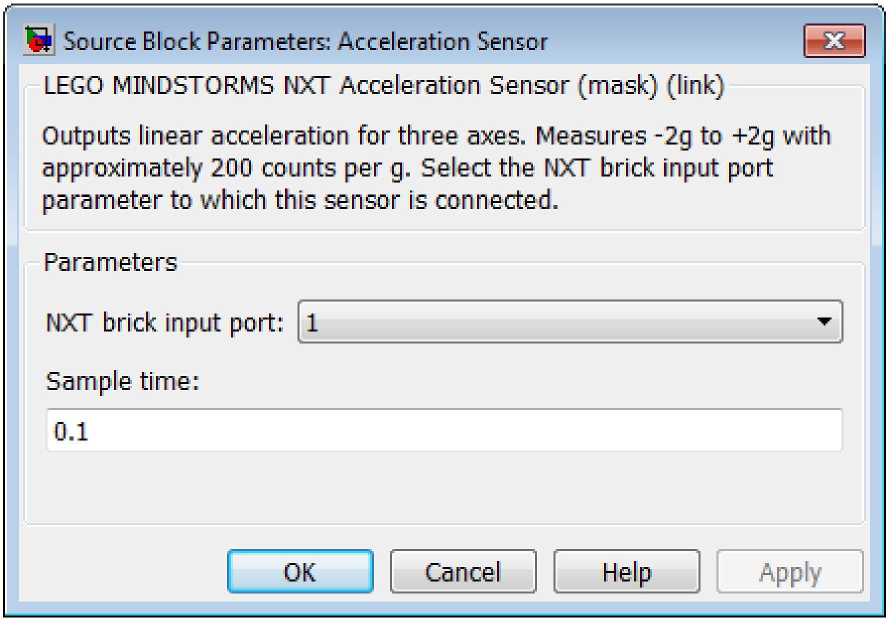
Connect the Acceleration sensor to Port 1 on the NXT brick.

Prepare and run the model on the NXT brick.

Move the Acceleration sensor about and observe the outputs on the LCD.

Position the Acceleration sensor on a table and systematically change the orientation of the sensor. The horizontal axes, such as X and Y in the following illustration, output values near zero. The vertical axis, such as Z in the following illustration, measures Earth's gravitational acceleration and therefore outputs a value near 200. If you invert the Z axis from the position shown, the sensor outputs a value near `-200`.




Source Block Parameters: Acceleration Sensor

LEGO MINDSTORMS NXT Acceleration Sensor (mask) (link)

Outputs linear acceleration for three axes. Measures -2g to +2g with approximately 200 counts per g. Select the NXT brick input port parameter to which this sensor is connected.

Parameters

NXT brick input port: 1

Sample time:

0.1

Dialog Box

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are `1`, `2`, `3`, or `4`.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

"Install Support for LEGO® MINDSTORMS® NXT Hardware" | "Detect and Fix Task Overruns on NXT Brick" |

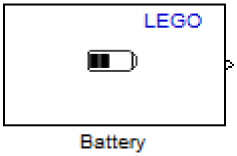
External Web Sites

- HiTechnic NXT Acceleration / Tilt Sensor (NAC1040)

LEGO MINDSTORMS NXT Battery

Purpose Measure voltage of battery in NXT brick

Library Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

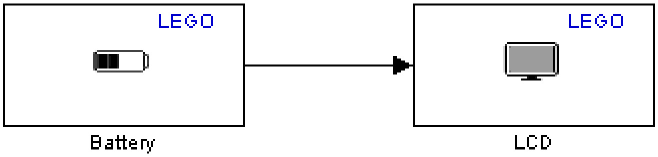
This block measures the voltage of the battery located in the NXT brick, and outputs the value in millivolts. For example, when the six 1.5 volt batteries are fully charged, the output value is approximately `9000`.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Measure Battery Voltage

Measure the voltage of the batteries in the NXT brick.

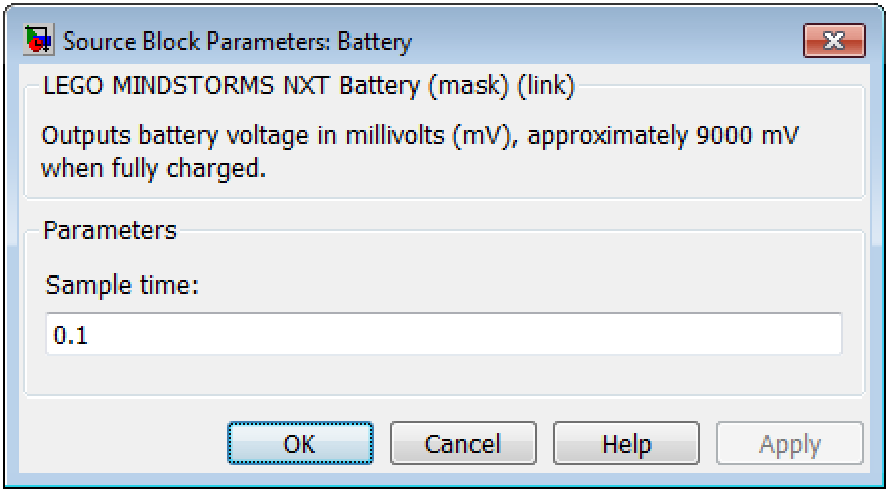
Open the `legonxtlib` block library and copy the blocks shown to a new model.



Connect the Battery block to the LCD block, as shown.

Prepare and run the model on the NXT brick.

Observe the battery voltage, shown in millivolts, on the LCD display.



Source Block Parameters: Battery

LEGO MINDSTORMS NXT Battery (mask) (link)

Outputs battery voltage in millivolts (mV), approximately 9000 mV when fully charged.

Parameters

Sample time: 0.1

Dialog Box

Sample time

Specify how often the block reads battery values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Battery | LEGO MINDSTORMS NXT Button | LEGO MINDSTORMS NXT LCD | LEGO MINDSTORMS NXT Speaker | LEGO MINDSTORMS NXT Timer | “Install Support for LEGO MINDSTORMS NXT Hardware” | “Detect and Fix Task Overruns on NXT Brick” |

External Web Sites

- http://mindstorms.lego.com

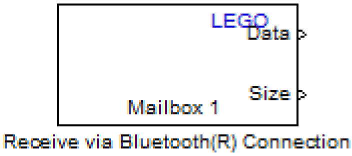
LEGO MINDSTORMS NXT Receive via Bluetooth Connection

Purpose

Receive data from another NXT brick over Bluetooth

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

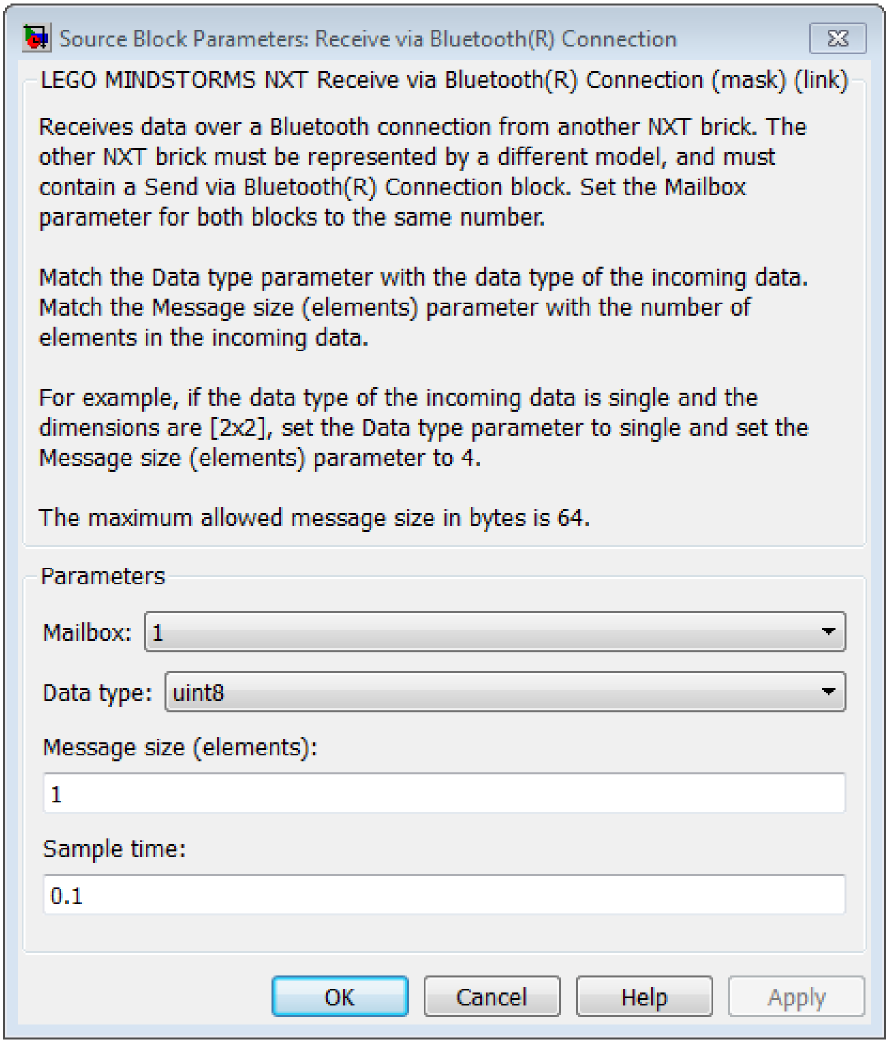
Receive data from another NXT brick over the built-in Bluetooth® device. The model running on the other NXT brick must contain a Send via Bluetooth Connection block.

To establish multiple channels between the two NXT bricks, use multiple pairs of Bluetooth blocks, and set the Mailbox parameters for each pair to a unique number. This approach cannot be used to establish communication between more than two NXT bricks.

Note Because External mode relies on a Bluetooth connection between the host computer and the NXT brick, you cannot use External mode with models that contain a Send via Bluetooth Connection block or a Receive via Bluetooth Connection block.

LEGO MINDSTORMS NXT Receive via Bluetooth Connection

Dialog Box



Source Block Parameters: Receive via Bluetooth(R) Connection

LEGO MINDSTORMS NXT Receive via Bluetooth(R) Connection (mask) (link)

Receives data over a Bluetooth connection from another NXT brick. The other NXT brick must be represented by a different model, and must contain a Send via Bluetooth(R) Connection block. Set the Mailbox parameter for both blocks to the same number.

Match the Data type parameter with the data type of the incoming data. Match the Message size (elements) parameter with the number of elements in the incoming data.

For example, if the data type of the incoming data is single and the dimensions are [2x2], set the Data type parameter to single and set the Message size (elements) parameter to 4.

The maximum allowed message size in bytes is 64.

Parameters

Mailbox: 1

Data type: uint8

Message size (elements): 1

Sample time: 0.1

Mailbox

Set the Mailbox parameter in both the Send via Bluetooth Connection and Receive via Bluetooth Connection blocks to the same number.

Data type

Match the **Data type** parameter with the data type of the incoming data.

Message size (elements)

Match the **Message size (elements)** parameter with the number of elements in the incoming data. For example, if the data type of the incoming data is single and the dimensions are [2x2], set the **Message size (elements)** parameter to `single` and set the **Message size (elements)** to `4`.

Sample time

Specify how often the block reads buffer values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Send via Bluetooth Connection **|** "Install Support for LEGO MINDSTORMS NXT Hardware" **|** "Detect and Fix Task Overruns on NXT Brick" **|**

Related Examples

- "Exchange Data Between Two NXT Bricks"

External Web Sites

- http://mindstorms.lego.com

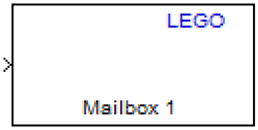
LEGO MINDSTORMS NXT Send via Bluetooth Connection

Purpose

Send data to another NXT brick over Bluetooth

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Send via Bluetooth(R) Connection

Description

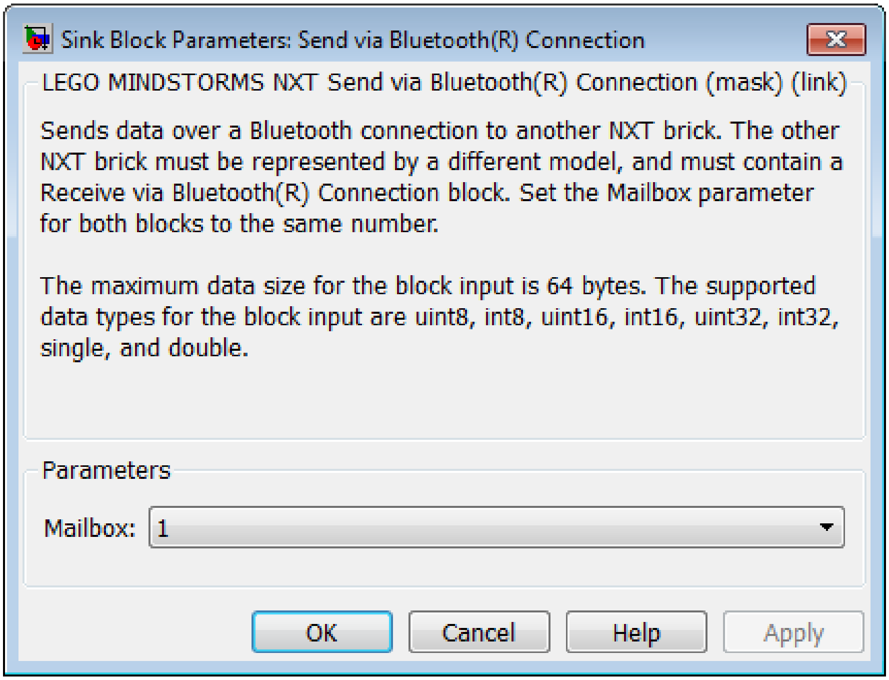
Send data to another NXT brick over the built-in Bluetooth device. The model running on the other NXT brick must contain a Receive via Bluetooth Connection block.

To establish multiple channels between the two NXT bricks, use multiple pairs of Bluetooth blocks, and set the Mailbox parameters for each pair to a unique number. This approach cannot be used to establish communication between more than two NXT bricks.

The maximum size for the block input is 64 bytes. The supported data types for the block input are uint8, int8, uint16, int16, uint32, int32, single, and double.

Note Because External mode relies on a Bluetooth connection between the host computer and the NXT brick, you cannot use External mode with models that contain a Send via Bluetooth Connection block or a Receive via Bluetooth Connection block.

Dialog Box



LEGO MINDSTORMS NXT Send via Bluetooth(R) Connection (mask) (link)

Sends data over a Bluetooth connection to another NXT brick. The other NXT brick must be represented by a different model, and must contain a Receive via Bluetooth(R) Connection block. Set the Mailbox parameter for both blocks to the same number.

The maximum data size for the block input is 64 bytes. The supported data types for the block input are uint8, int8, uint16, int16, uint32, int32, single, and double.

Parameters

Mailbox: 1

Mailbox

Set the Mailbox parameter in both the Send via Bluetooth Connection and Receive via Bluetooth Connection blocks to the same number.

See Also

LEGO MINDSTORMS NXT Receive via Bluetooth Connection | “Install Support for LEGO MINDSTORMS NXT Hardware” |

Related Examples

- “Exchange Data Between Two NXT Bricks”

External Web Sites

- http://mindstorms.lego.com

LEGO MINDSTORMS NXT Button

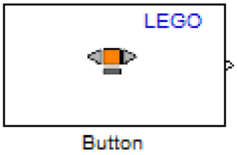
Purpose

Output state of button on NXT brick

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware

Description



Get the state of the “middle orange” or “right arrow” button on the NXT brick: `1` while pressed and `0` while released.

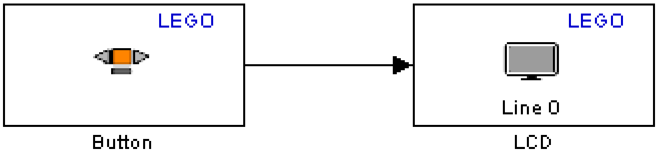
If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Use the Button

Measure the voltage of the batteries in the NXT brick.

Open the `legonxtlib` block library and copy the blocks shown to a new model.

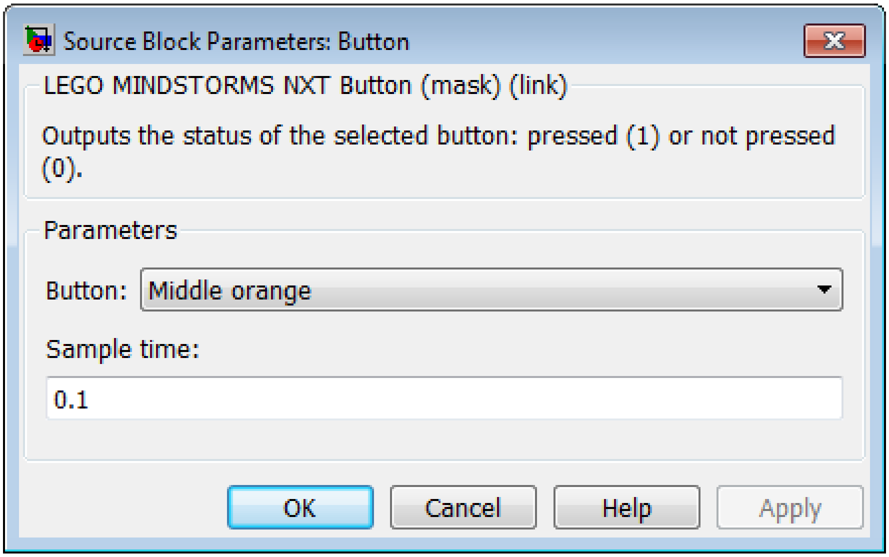
Connect the Button block to the LCD block, as shown.



Prepare and run the model on the NXT brick.

Press the orange button on the NXT brick, and observe `0` on the LCD display turn to `1`.

Dialog Box



Source Block Parameters: Button

LEGO MINDSTORMS NXT Button (mask) (link)

Outputs the status of the selected button: pressed (1) or not pressed (0).

Parameters

Button: Middle orange

Sample time: 0.1

Button

Select the button for which to output status. The options are `Middle orange` and `Right arrow`.

Sample time

Specify how often the block reads the state of the button.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Battery | LEGO MINDSTORMS NXT Button | LEGO MINDSTORMS NXT LCD | LEGO MINDSTORMS NXT Speaker | LEGO MINDSTORMS NXT Timer | “Install Support for LEGO MINDSTORMS NXT Hardware” | “Detect and Fix Task Overruns on NXT Brick” |

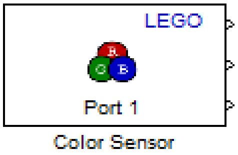
External Web Sites

- http://mindstorms.lego.com

LEGO MINDSTORMS NXT Color Sensor

Purpose Measure color or light intensity

Library Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

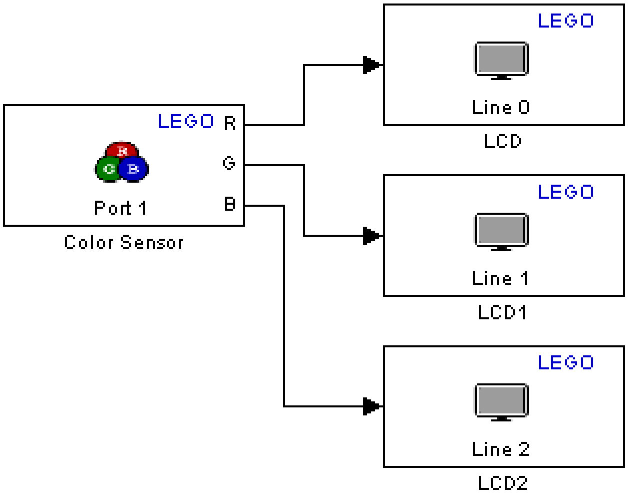
Measure color, or light intensity (luminance) from the LEGO 10286 Color Sensor. The sensor measures color using separate red, green, and blue (RGB) components.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Use the Color Sensor

Use the Color Sensor block to measure the color or light intensity of an object.

Open the `legonxtlib` block library and copy the blocks shown to a new model.



Connect the Color Sensor block to the LCD blocks.

Double click each LCD block to open the block masks for each block.

Use the **Display at line** parameters to select a different line number for each block.

Use the **Display label** parameters to add an `R`, `G`, or `B` label for each line.

Connect the color sensor to Port 1 on the NXT brick.

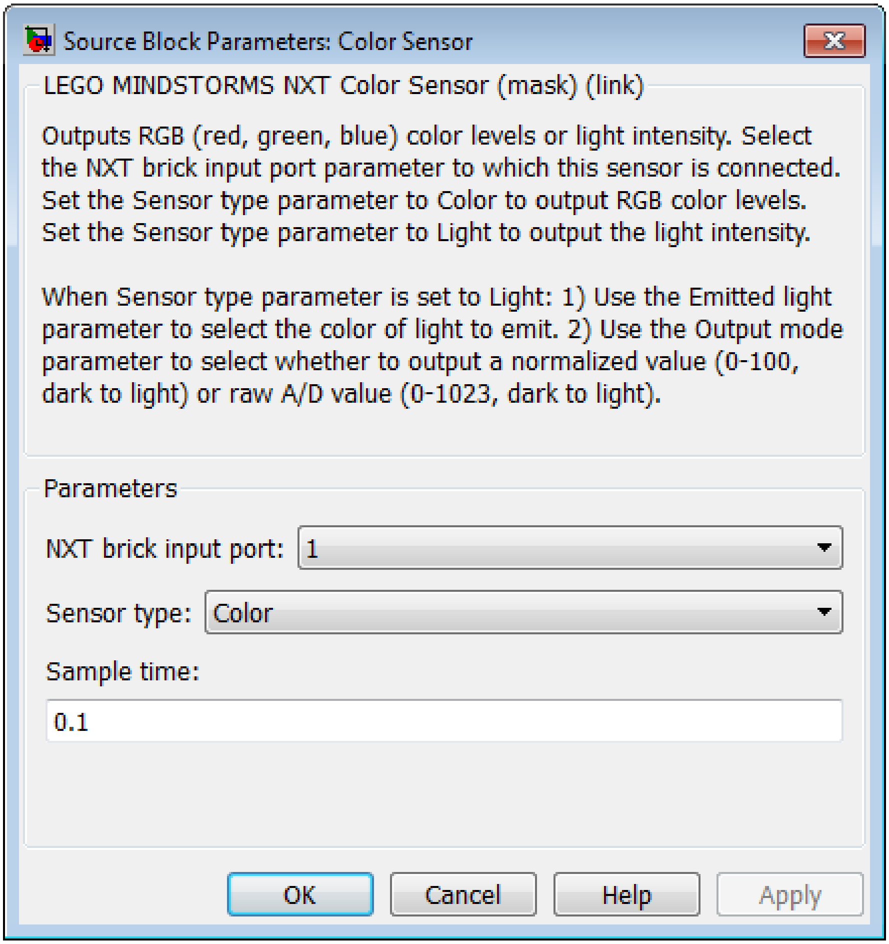
Prepare and run the model on the NXT brick.

Point the color sensor at a variety of well-lit colored objects and observe the measured red, green, and blue components of that color on the LCD. When you point the sensor at a bright white light, all three output values are near `255`. When you point the sensor at a dark shadow, all three output values are low, nearer to `0`.

In the Color Sensor block, change the **Sensor type** parameter to `Light`. This action changes the block output for the red component, **R**, into an unlabeled block output for light measurements.

Delete the broken connectors and unused LCD blocks from the model, and then run the model on the NXT brick.

Point the color sensor at a variety of well-lit colored objects and observe the light measurements. When you point the sensor at a bright white light, the block output values are high, near `90`. When you point the sensor at a dark shadow, all three output values are low, near `6`.



Source Block Parameters: Color Sensor

LEGO MINDSTORMS NXT Color Sensor (mask) (link)

Outputs RGB (red, green, blue) color levels or light intensity. Select the NXT brick input port parameter to which this sensor is connected. Set the Sensor type parameter to Color to output RGB color levels. Set the Sensor type parameter to Light to output the light intensity.

When Sensor type parameter is set to Light: 1) Use the Emitted light parameter to select the color of light to emit. 2) Use the Output mode parameter to select whether to output a normalized value (0-100, dark to light) or raw A/D value (0-1023, dark to light).

Parameters

NXT brick input port: 1

Sensor type: Color

Sample time: 0.1

Dialog Box

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are 1, 2, 3, or 4.

Sensor type

Select the type of measurement to perform, `Color` (the default) or `Light`.

When you select `Color`, the block outputs a 1-by-3 matrix of RGB values. Each value ranges from 0 (darkest) to 255 (brightest).

When you select `Light`, the block displays two new parameters, **Emitted light** and **Output mode**.

Emitted light

This parameter is only available when the **Sensor type** parameter is set to `Light`.

Use **Emitted light** to choose the color of light the LED on the sensor emits, or to turn the LED off. The options are `Red`, `Green`, `Blue`, `White`, and `None`.

Output mode

This parameter is only available when the **Sensor type** parameter is set to `Light`.

Select the range the block outputs, `Normalized (0-100)` or `Raw A/D (0-1023)`.

When you select `Normalized (0-100)`, the block outputs light intensity as normalized values that range from 0 (darkest) to 100 (brightest).

When you select `Raw A/D (0-1023)`, the block outputs light intensity as raw values that ranges from 0 (darkest) to 1023 (brightest).

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Light Sensor | “Install Support for LEGO MINDSTORMS NXT Hardware” | “Detect and Fix Task Overruns on NXT Brick” |

External Web Sites

- http://mindstorms.lego.com

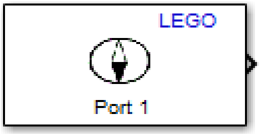
LEGO MINDSTORMS NXT Compass Sensor

Purpose

Read magnetic heading of compass sensor

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Compass Sensor

Description

Read the magnetic heading of the compass sensor. The block outputs the heading as a int8 value, from 0 to 359, where:

- 0 is North
- 90 is East
- 180 is South
- 270 is West

The sensor reads the heading 100 times per second, and is accurate to within 1 degree.

This block supports the following sensors:

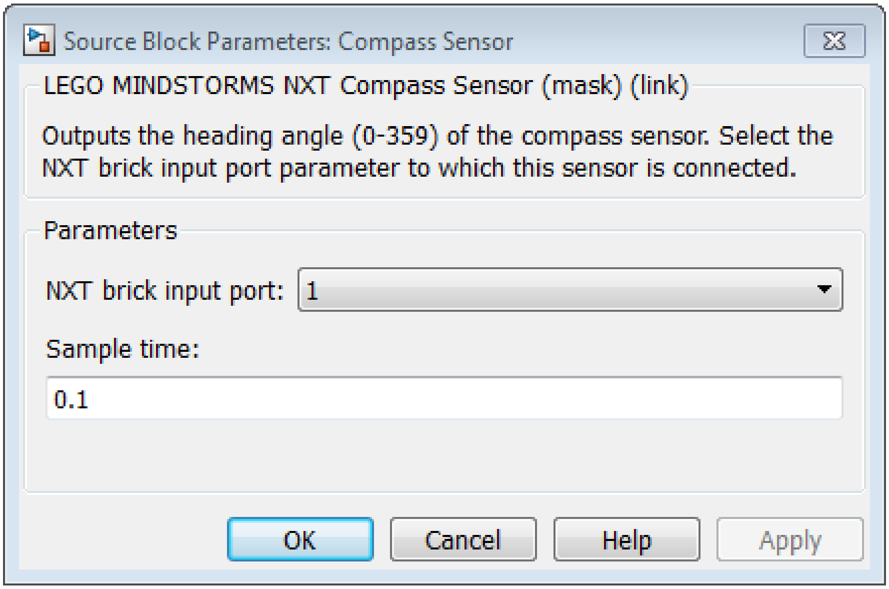
- HiTechnic NXT Compass Sensor (NMC1034)
- LEGO MINDSTORMS NXT Compass Sensor, Item: #10285

To configure the block, select the NXT brick input port parameter to which this sensor is connected.

Position the sensor away from sources of magnetic interference, such as motors and speakers. This block does not provide a calibration mode to reduce the effects of magnetic interference from nearby objects.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Dialog Box



Source Block Parameters: Compass Sensor

LEGO MINDSTORMS NXT Compass Sensor (mask) (link)

Outputs the heading angle (0-359) of the compass sensor. Select the NXT brick input port parameter to which this sensor is connected.

Parameters

NXT brick input port: 1

Sample time: 0.1

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are `1`, `2`, `3`, or `4`.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

“Install Support for LEGO MINDSTORMS NXT Hardware” **|** “Detect and Fix Task Overruns on NXT Brick” **|**

External Web Sites

- http://mindstorms.lego.com

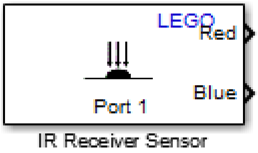
LEGO MINDSTORMS NXT IR Receiver Sensor

Purpose

Receive infrared signals from LEGO Power Functions IR Speed Remote Control

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

Receive infrared signals from LEGO Power Functions IR Speed Remote Control.

The remote control has two channels:

- The left dial and stop button are “red”.
- The right dial and stop button are “blue”.

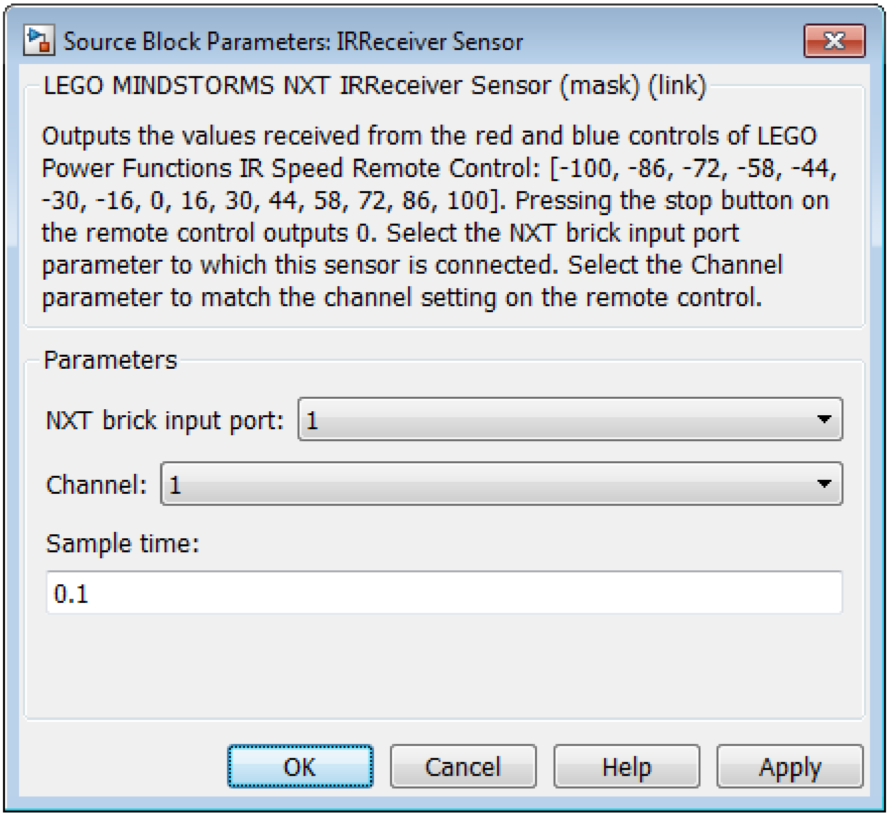
For each channel on the remote control:

- Turning the transmitter on initializes the output value to 0.
- Rotating the dial either way increases or decreases the output value to: [-100, -86, -72, -58, -44, -30, -16, 0, 16, 30, 44, 58, 72, 86, 100]
- Changing the position of the black switch inverts polarity of the dial.
- Pressing the red “stop” button resets the output value to 0.

With the block switch in the down position, turning a dial clockwise outputs positive values.

To configure the block, select the NXT brick input port to which this sensor is connected. Select the Channel parameter to match the channel setting on the remote control.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.



Source Block Parameters: IRReceiver Sensor

LEGO MINDSTORMS NXT IRReceiver Sensor (mask) (link)

Outputs the values received from the red and blue controls of LEGO Power Functions IR Speed Remote Control: [-100, -86, -72, -58, -44, -30, -16, 0, 16, 30, 44, 58, 72, 86, 100]. Pressing the stop button on the remote control outputs 0. Select the NXT brick input port parameter to which this sensor is connected. Select the Channel parameter to match the channel setting on the remote control.

Parameters

NXT brick input port: 1

Channel: 1

Sample time: 0.1

Dialog Box

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are `1`, `2`, `3`, or `4`.

Channel

Update this value to match the channel setting on the remote control.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Light Sensor | "Install Support for LEGO MINDSTORMS NXT Hardware" | "Detect and Fix Task Overruns on NXT Brick" |

External Web Sites

- http://mindstorms.lego.com

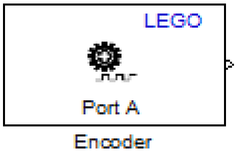
LEGO MINDSTORMS NXT Encoder

Purpose

Measure encoder rotation from NXT Interactive Servo Motor

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

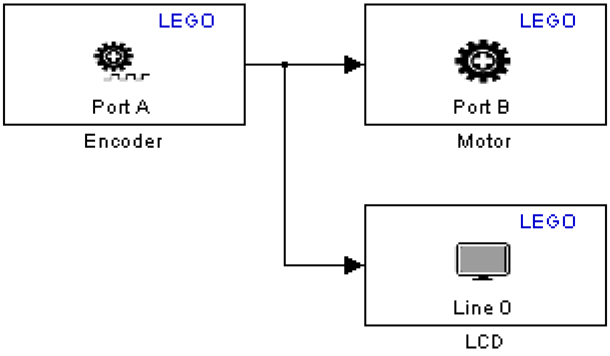
This block uses the encoder in the LEGO 9842 Interactive Servo Motor to measure the rotation of the wheel in degrees. Driving the wheel forward increments the measurement, and driving the wheel in reverse decrements the measurement.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Use the Encoder

Use the Encoder block to measure encoder rotation from an NXT Interactive Servo Motor

Open the `legonxtlib` block library and copy the blocks shown to a new model.



Connect the Encoder block to the Motor block and the LCD block, as shown.

Double click the Motor block to open the block mask, and change the **NXT brick output port** parameter to `B`.

Connect one servo motor to Port A on the NXT brick. Connect another servo motor to Port B on the NXT brick.

Prepare and run the model on the NXT brick.

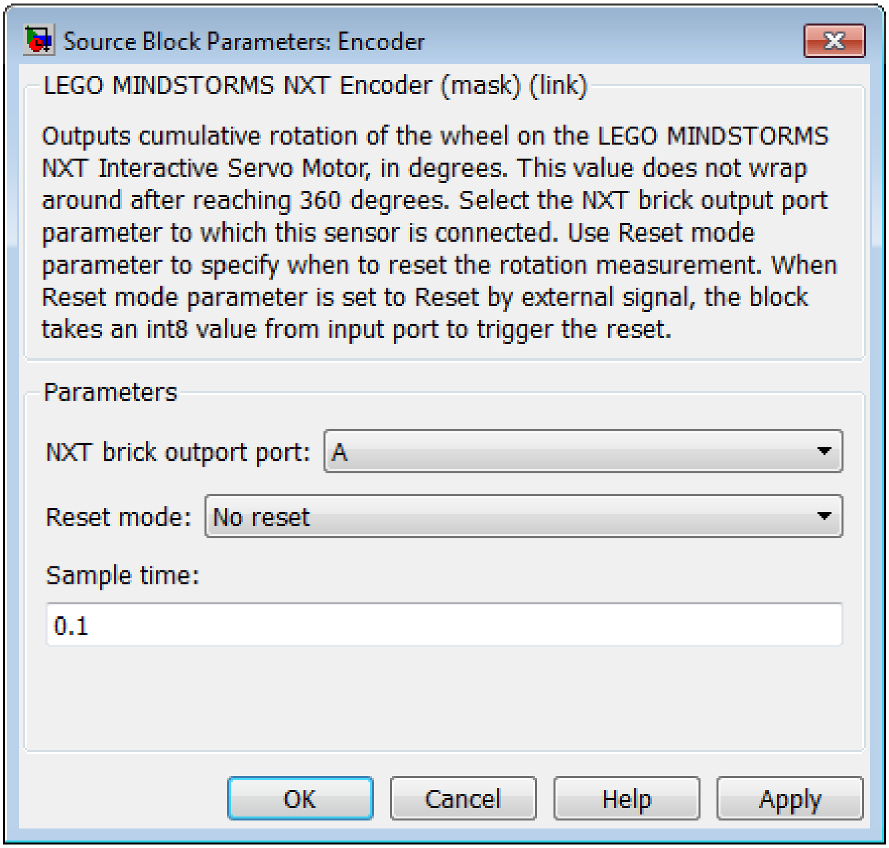
Slowly rotate the wheel of servo motor connected to Port A in each direction. Observe the servo motor connected to Port B drive in one direction and then in the other direction as you input positive or negative values. Using the input value displayed on the LCD, observe that values larger than 100 or -100 do not increase the motor speed.

In the Encoder block, experiment with changing the **Reset mode** parameter to `Reset at each sample time`, and changing the **Sample time** parameter to `1`.

Run the model on the NXT brick again.

Observe that the block output values remain lower and reset to zero every second.

Dialog Box



Source Block Parameters: Encoder

LEGO MINDSTORMS NXT Encoder (mask) (link)

Outputs cumulative rotation of the wheel on the LEGO MINDSTORMS NXT Interactive Servo Motor, in degrees. This value does not wrap around after reaching 360 degrees. Select the NXT brick output port parameter to which this sensor is connected. Use Reset mode parameter to specify when to reset the rotation measurement. When Reset mode parameter is set to Reset by external signal, the block takes an int8 value from input port to trigger the reset.

Parameters

NXT brick outport port: A

Reset mode: No reset

Sample time:

0.1

NXT brick output port

Select the NXT brick output port to which the encoder is connected. The options are `A`, `B`, or `C`.

The LEGO 9842 Interactive Servo Motor contains both a motor and an encoder. If your model uses a Motor block and an Encoder block for the same LEGO 9842 Interactive Servo Motor, set the **NXT brick output port** parameter in both of those blocks to the same port.

Reset mode

Choose whether to reset the measurement value:

- `No reset` — Do not reset the measurement value. Measure the total degrees of rotation.
- `Reset at each sample time` — For each sample period, output the measurement value and then reset it to 0.
- `Reset by external signal` — Create a block input that can reset the measurement value to 0. When you send a value other than 0 to the block input, reset the measurement value to 0. For each sample period, output the measurement value since the last reset.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Motor | "Install Support for LEGO MINDSTORMS NXT Hardware" | "Detect and Fix Task Overruns on NXT Brick" |

External Web Sites

- http://mindstorms.lego.com

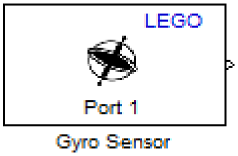
LEGO MINDSTORMS NXT Gyro Sensor

Purpose

Measure rate of rotation

Library

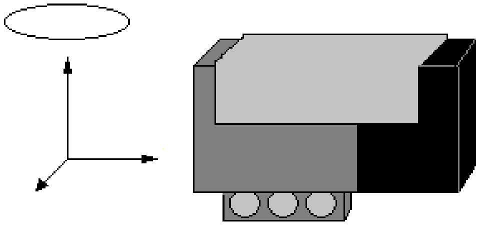
Simulink Support Package for LEGO MINDSTORMS NXT Hardware

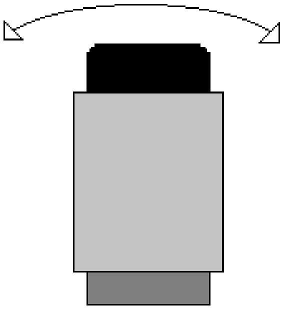


Description

This block measures rate of rotation (yaw) along the a single axis of the HiTechnic NXT Gyro Sensor (NGY1044). This yaw is indicated by a circle above the vertical axis in the image on the left, and by a double-arrow in the image on the right.

Before using this sensor for precise measurements, use the offset value to remove small positive or negative bias in the output values, as described in the example below.



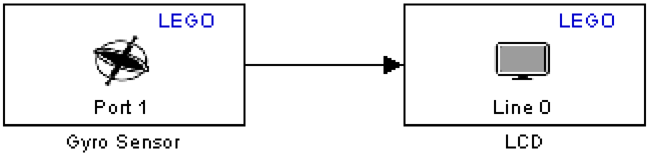


If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Calibrate the Gyro Sensor

Use the Gyro Sensor block to perform zero-calibration for the gyro sensor

Open the `legonxtlib` block library and copy the blocks shown to a new model.



Connect the Gyro Sensor block to the LCD block, as shown.

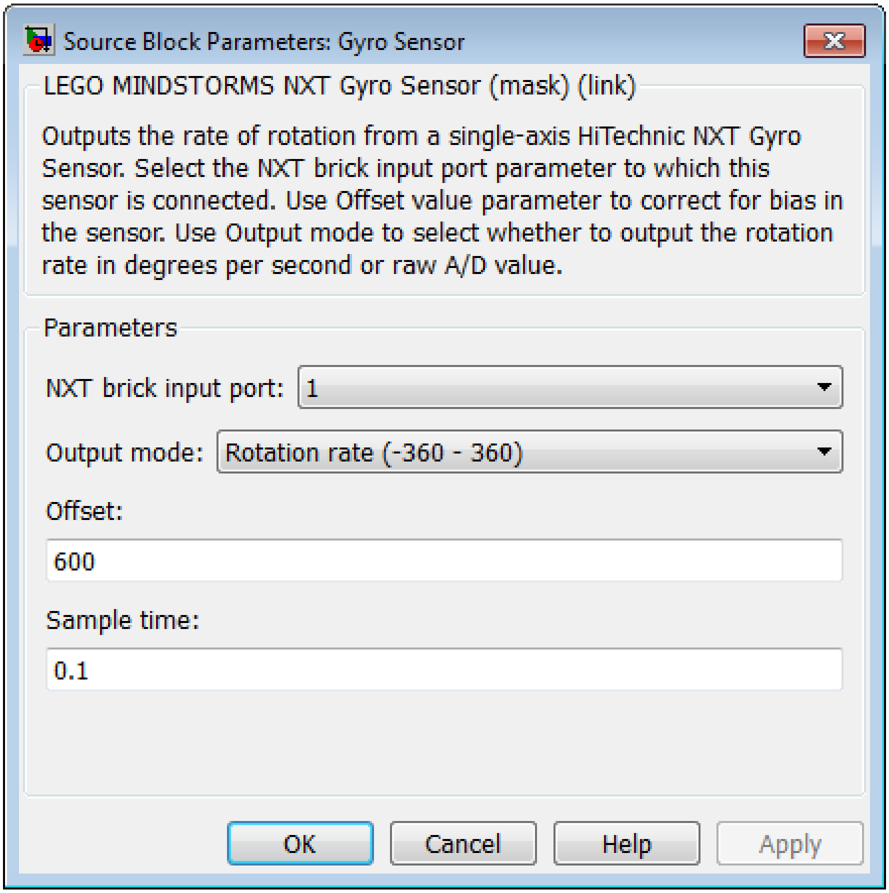
Connect the NXT Gyro Sensor to the NTX brick.

Prepare and run the model on the NXT brick.

Observe any bias in the measurement values from the Gyro sensor visible on the LCD display of the NXT brick.

Double-click the Gyro Sensor block. In the block mask that opens, increase or decrease the offset value to reduce the bias.

Save the changes to your model and run the model again. Repeat these last two steps until the bias is nearly zero.



Source Block Parameters: Gyro Sensor

LEGO MINDSTORMS NXT Gyro Sensor (mask) (link)

Outputs the rate of rotation from a single-axis HiTechnic NXT Gyro Sensor. Select the NXT brick input port parameter to which this sensor is connected. Use Offset value parameter to correct for bias in the sensor. Use Output mode to select whether to output the rotation rate in degrees per second or raw A/D value.

Parameters

NXT brick input port: 1

Output mode: Rotation rate (-360 - 360)

Offset:

600

Sample time:

0.1

Dialog Box

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are 1, 2, 3, or 4.

Output mode

Select the type of rotation data to output:

`Rotation rate (-360 - 360)` outputs the rate of rotation in degrees per second. When this option is selected, the block displays the **Offset** parameter.

`Raw A/D (0 - 1023)` outputs the rate of rotation in raw analog-to-digital values per second. When this option is selected, the block hides the **Offset** parameter.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Acceleration Sensor | "Install Support for LEGO MINDSTORMS NXT Hardware" | "Detect and Fix Task Overruns on NXT Brick" |

External Web Sites

- HiTechnic NXT Gyro Sensor (NGY1044)

LEGO MINDSTORMS NXT LCD

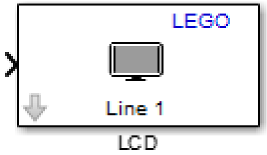
Purpose

Display number on single line of NXT LCD screen

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware

Description



This block displays the value of the block input on the LCD located on the NXT brick. This block lets you enter a label for the value, and select the line of the LCD upon which to display the label/value.

If the value displayed changes faster than the refresh rate of the LCD, the displayed values may be blurry and hard to read.

If the model running on the NXT brick does not use the LCD, the LCD displays a default message that it is running a program.

Avoid configuring other LCD blocks to display information on the same line number.

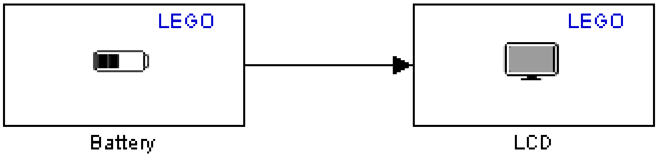
This block automatically converts the data type of the data it receives. You do not need to perform data-type conversion on the input signals.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See “Block Produces Zeros in Simulation”.

Use the LCD Display

Use the LCD block to display a number on a single line of the LCD display on the NXT brick

Open the `legonxtlib` block library and copy the blocks shown to a new model.



Connect the Battery block to the LCD block, as shown.

Prepare and run the model on the NXT brick.

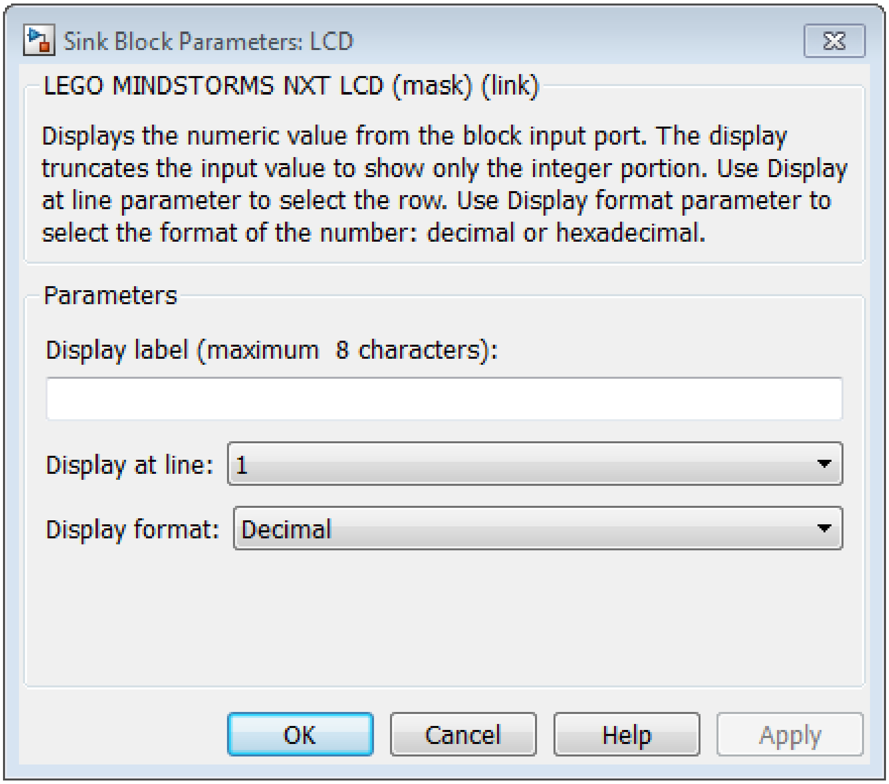
Observe the battery voltage, shown in millivolts, on line 1 of the LCD display.

Double-click the LCD block in your model.

In the block mask that opens, set the **Display label** parameter to `Mv` and the **Display at line** parameter to `3`.

Run the model again. On the NXT brick, observe the new label and position of the battery output on the LCD screen.

Dialog Box



Sink Block Parameters: LCD

LEGO MINDSTORMS NXT LCD (mask) (link)

Displays the numeric value from the block input port. The display truncates the input value to show only the integer portion. Use Display at line parameter to select the row. Use Display format parameter to select the format of the number: decimal or hexadecimal.

Parameters

Display label (maximum 8 characters):

Display at line: 1

Display format: Decimal

Display label (maximum 8 characters)

Enter a prefix or label for the input to display. the LCD displays the prefix to the left of the input value. The prefix cannot be longer than 8 characters. For example, if you are displaying the state of the Button block, enter `Button:`.

Display at line

Select the line upon which to display the number. Line `1` is at the top of the LCD. Line `8` is at the bottom.

Display format

Select the format of the number. The options are `Decimal` or `Hex` (hexadecimal).

See Also

LEGO MINDSTORMS NXT Battery | LEGO MINDSTORMS NXT Button | LEGO MINDSTORMS NXT LCD | LEGO MINDSTORMS NXT Speaker | LEGO MINDSTORMS NXT Timer | "Install Support for LEGO MINDSTORMS NXT Hardware" |

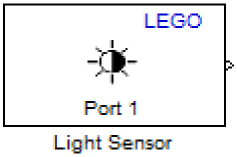
External Web Sites

- http://mindstorms.lego.com

LEGO MINDSTORMS NXT Light Sensor

Purpose Measure light intensity

Library Simulink Support Package for LEGO MINDSTORMS NXT Hardware

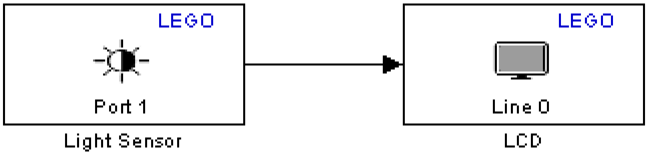


Description

This block outputs the intensity of light detected by the NXT Light Sensor. The sensor includes an illuminating LED to actively measure the reflectance of nearby surfaces. For example, the light-intensity under a desk measures approximately 20 while a the light-intensity of fluorescent light is approximately 85.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See "Block Produces Zeros in Simulation".

Use the Light Sensor



1. Open the `legonxtlib` block library and copy the blocks shown to a new model.
2. Connect the Light Sensor block to the LCD block, as shown.
3. Connect the Light sensor to Port 1 on the NXT brick.
4. Prepare and run the model on the NXT brick.

5 Point the Light sensor at a variety of well-lit objects and observe the measurement on the LCD.

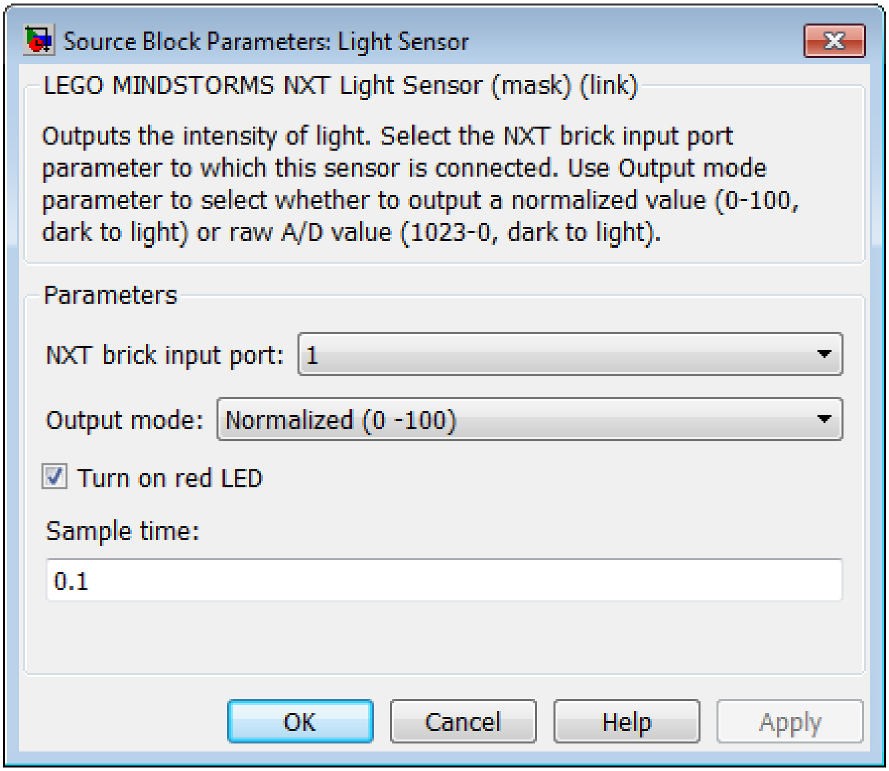
When you point the sensor at a bright light, the measurement value is higher.

When you point the sensor at a dark shadow, the measurement value is lower.

6 Hold the Light sensor near a piece of white paper in a dark location, such as under your desk. The red LED in the sensor illuminates the paper. Observe the measurements on the LCD.

When you hold the sensor nearer to the paper, the measurement value is higher.

When you hold the sensor further away from the paper, the measurement value is lower.



Source Block Parameters: Light Sensor

LEGO MINDSTORMS NXT Light Sensor (mask) (link)

Outputs the intensity of light. Select the NXT brick input port parameter to which this sensor is connected. Use Output mode parameter to select whether to output a normalized value (0-100, dark to light) or raw A/D value (1023-0, dark to light).

Parameters

NXT brick input port: 1

Output mode: Normalized (0 -100)

☑ Turn on red LED

Sample time:

0.1

Dialog Box

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are `1`, `2`, `3`, or `4`.

Output mode

Select the type of light intensity data to output:

`Light intensity (0 - 100)` outputs the intensity of the light on a scale from 0 to 100. 0 is the weakest intensity. 100 is the strongest intensity.

`Raw A/D (1023 - 0)` outputs the intensity of the light on a scale from 1023 to 0. 1023 is the weakest intensity. 0 is the strongest intensity.

Turn on red LED

Emit red light from the LED located next to the light sensor. Under low light conditions, the sensor can use this capability to measure the amount of red light nearby objects reflect.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Color Sensor | "Install Support for LEGO MINDSTORMS NXT Hardware" | "Detect and Fix Task Overruns on NXT Brick" |

External Web Sites

- http://mindstorms.lego.com

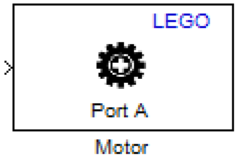
LEGO MINDSTORMS NXT Motor

Purpose

Set motor speed and stopping action

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

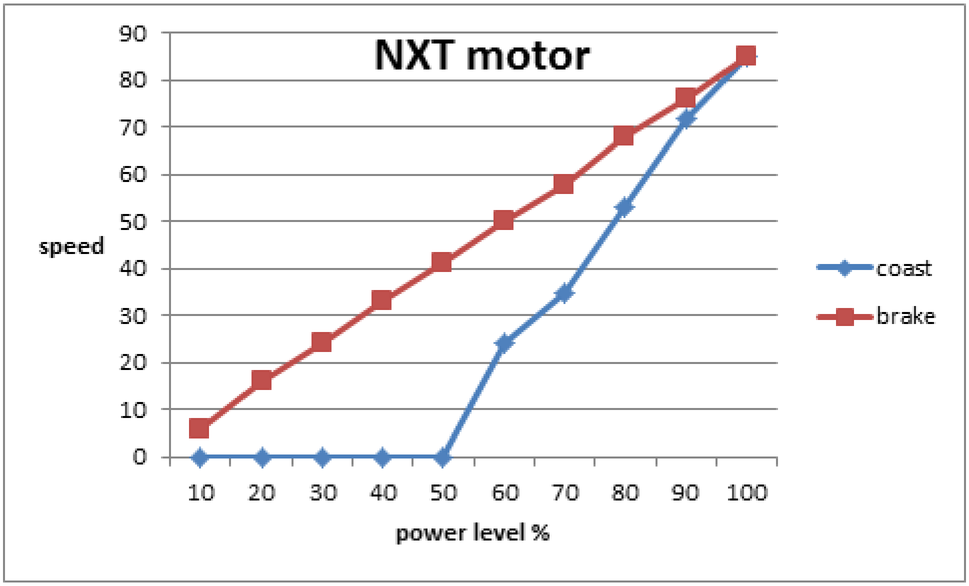
This blocks controls the speed, direction, and stopping action of the LEGO 9842 Interactive Servo Motor.

You can use this block with the LEGO MINDSTORMS NXT Encoder block, which outputs the rotation of the servo motor.

The Motor block input value controls the motor speed and direction.

- Forward: 1 to 100 (slowest to fastest)
- Reverse: –1 to –100 (slowest to fastest)
- Stop: 0

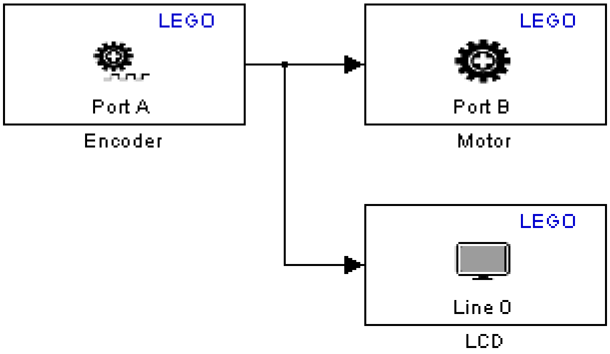
Using the **Stop action** parameter in the block, you can configure the motor to coast or brake when it receives an input of 0 (stop). As a side effect, when you change **Stop action** from `Brake` to `Coast`, the servo motor requires higher block input values to start moving and to maintain speed. The following graph shows this relationship.



This block automatically converts the data type of the data it receives. You do not need to perform data-type conversion on the input signals.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See "Block Produces Zeros in Simulation".

Use the Motor



1 Open the `legonxtlib` block library and copy the blocks shown to a new model.

2 Connect the Encoder block to the Motor block and the LCD block, as shown.

3 Double click the Motor block to open the block mask, and change the **NXT brick output port** parameter to `B`.

4 Connect one servo motor to Port A on the NXT brick. Connect another servo motor to Port B on the NXT brick.

5 Prepare and run the model on the NXT brick.

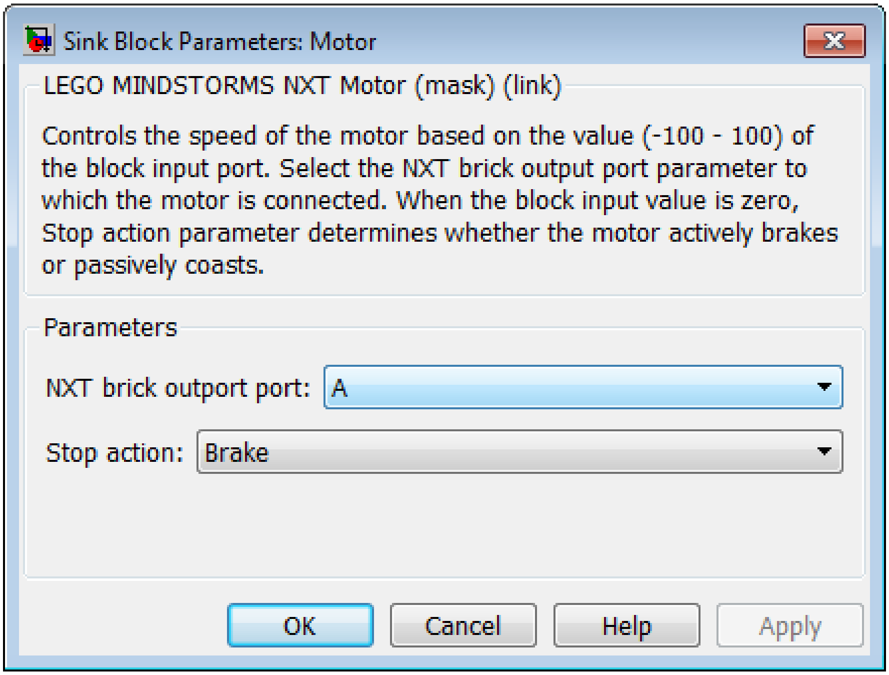
6 Slowly rotate the wheel of servo motor connected to Port A in each direction. Observe the servo motor connected to Port B drive in one direction and then in the other direction as you input positive or negative values. Using the input value displayed on the LCD, observe that values larger than 100 or -100 do not increase the motor speed.

7 In the Motor block, experiment with changing the **Stop action** parameter to `Coast`.

Run the model on the NXT brick again.

Observe that the motor requires higher block input values both to start and to achieve the same speed as when the **Stop action** parameter was set to `Brake`.

Dialog Box



Sink Block Parameters: Motor

LEGO MINDSTORMS NXT Motor (mask) (link)

Controls the speed of the motor based on the value (-100 - 100) of the block input port. Select the NXT brick output port parameter to which the motor is connected. When the block input value is zero, Stop action parameter determines whether the motor actively brakes or passively coasts.

Parameters

NXT brick outport port: A

Stop action: Brake

NXT brick output port

Select the NXT brick output port to which the encoder is connected. The options are `A`, `B`, or `C`.

The LEGO 9842 Interactive Servo Motor contains both a motor and an encoder. If your model uses a Motor block and an Encoder block for the same LEGO 9842 Interactive Servo Motor, configure the **NXT brick output port** parameter in both of those blocks to use the same NXT brick output port.

Stop action

Select the stopping behavior of the motor:

- `Coast` stops the motor gradually through loss of momentum. When **Stop action** is set to coast, getting the motor to start and run requires higher values at the block input than when **Stop action** is brake.
- `Brake` stops the motor immediately.

See Also

LEGO MINDSTORMS NXT Encoder | "Install Support for LEGO MINDSTORMS NXT Hardware" |

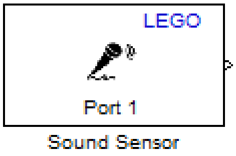
External Web Sites

- http://mindstorms.lego.com

LEGO MINDSTORMS NXT Sound Sensor

Purpose Measure sound level

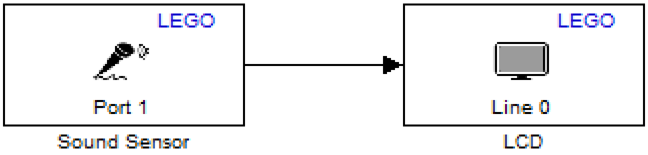
Library Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description Measure sound level using the LEGO MINDSTORMS NXT Sound Sensor.

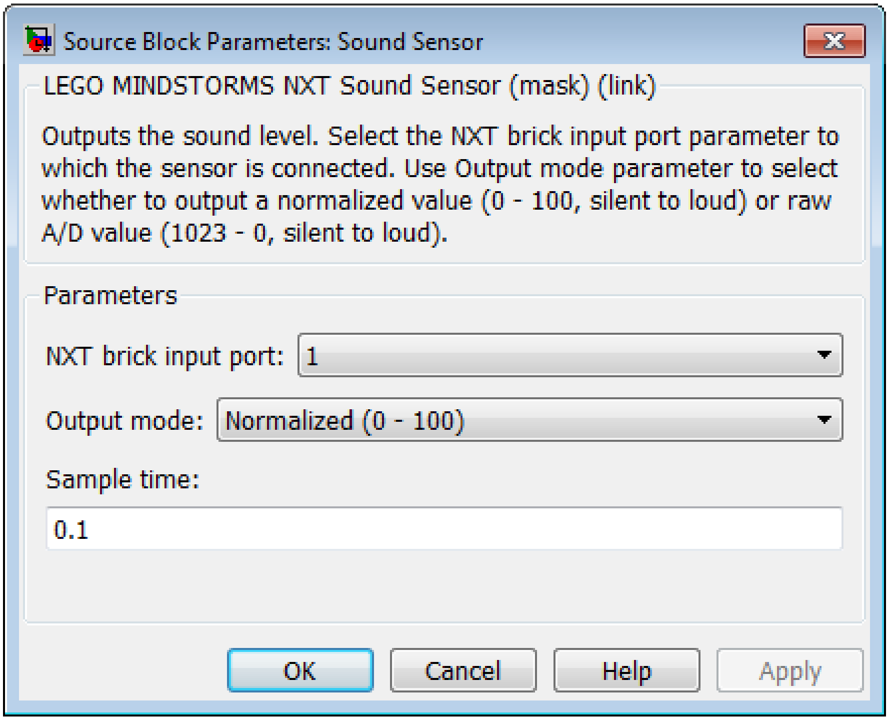
If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See "Block Produces Zeros in Simulation".

Use the Sound Sensor



1. Open the `legonxtlib` block library and copy the blocks shown to a new model.
2. Connect the Sound block to the LCD block, as shown.
3. Prepare and run the model on the NXT brick.
4. While making sounds close to the sensor, and observe the sound levels shown on the LCD display.

Dialog Box



LEGO MINDSTORMS NXT Sound Sensor (mask) (link)

Outputs the sound level. Select the NXT brick input port parameter to which the sensor is connected. Use Output mode parameter to select whether to output a normalized value (0 - 100, silent to loud) or raw A/D value (1023 - 0, silent to loud).

Parameters

NXT brick input port: 1

Output mode: Normalized (0 - 100)

Sample time:

0.1

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are `1`, `2`, `3`, or `4`.

Output mode

Select the type of data to output:

`Sound level (0 - 100)` outputs the intensity of the sound on a scale from 0 to 100. 0 is the weakest intensity. 100 is the strongest intensity.

`Raw A/D (1023 - 0)` outputs the intensity of the sound on a scale from 1023 to 0. 1023 is the weakest intensity. 0 is the strongest intensity.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

"Install Support for LEGO MINDSTORMS NXT Hardware" **|** "Detect and Fix Task Overruns on NXT Brick" **|**

External Web Sites

- http://mindstorms.lego.com

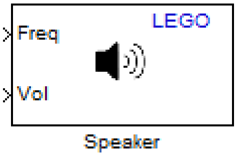
LEGO MINDSTORMS NXT Speaker

Purpose

Play tones on speaker in NXT brick

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

Use this block to play tones from the speaker in the NXT brick.

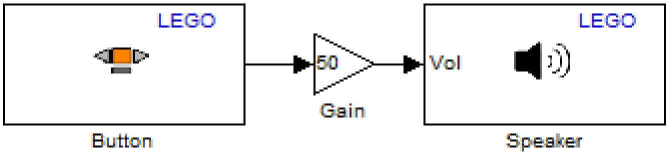
To play a tone, send an audible value to the block input for frequency (**Freq**) and/or volume (**Vol**). To stop playing a tone, send an inaudible value, such as zero, to either block input. The range of audible frequencies that the speaker can produce is approximately 32 to 13000 Hz. The speaker volume ranges from `0` to `100` (softest to loudest).

To set either the frequency or volume to a static value, disable one of the block inputs using the **Input mode** parameter, and then enter the static value for the equivalent parameter on the block mask. For example, if the speaker is only going to play a single frequency, set **Input mode** to `Volume only`, and then set the **Frequency (Hz)** parameter to `440`.

This block automatically converts the data type of the data it receives. You do not need to perform data-type conversion on the input signals.

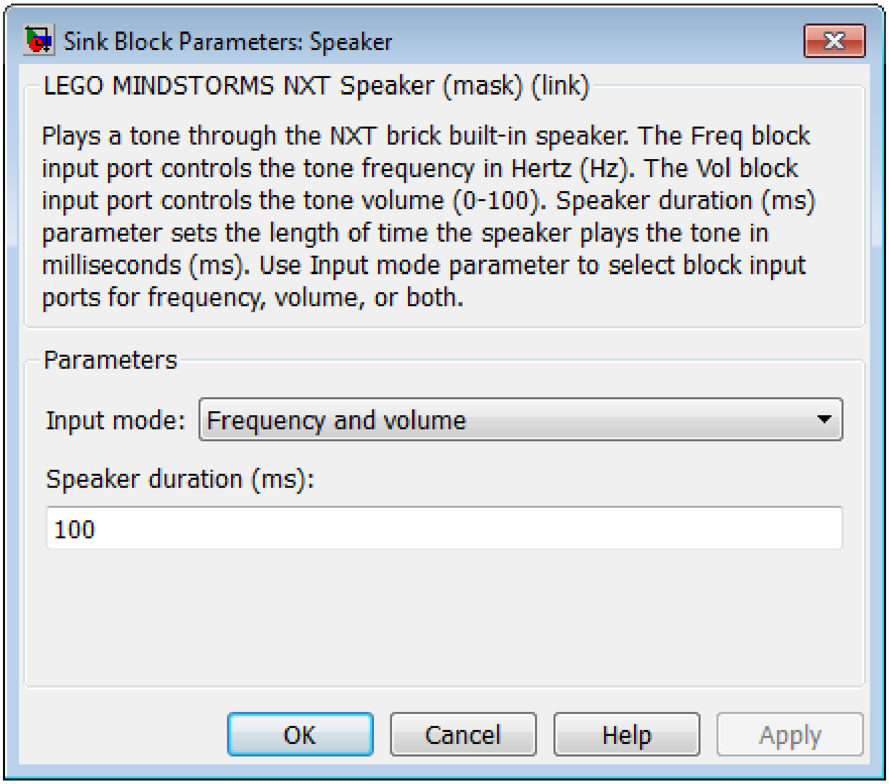
If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block does nothing. See "Block Produces Zeros in Simulation".

Use the Speaker



1. Open the `legonxtlib` block library and copy the blocks shown to a new model.
2. Double click the Speaker block. In the block mask that opens, change the **Input mode** parameter to `Volume only`, and enter a value of 440 for the **Frequency (Hz)** parameter. Then click **OK**.
3. Connect the Button block to the Gain block, and the Gain block to the Speaker block, as shown.
4. Double click the Gain block. In the block mask that opens, set the **Gain** parameter to `50`. When you press the orange button on the NXT brick, the Gain block will send a value of 50 to the volume input on the Speaker block.
5. Prepare and run the model on the NXT brick.
6. When you press the orange button, the speaker emits a 440 Hz tone.

Dialog Box



Sink Block Parameters: Speaker

LEGO MINDSTORMS NXT Speaker (mask) (link)

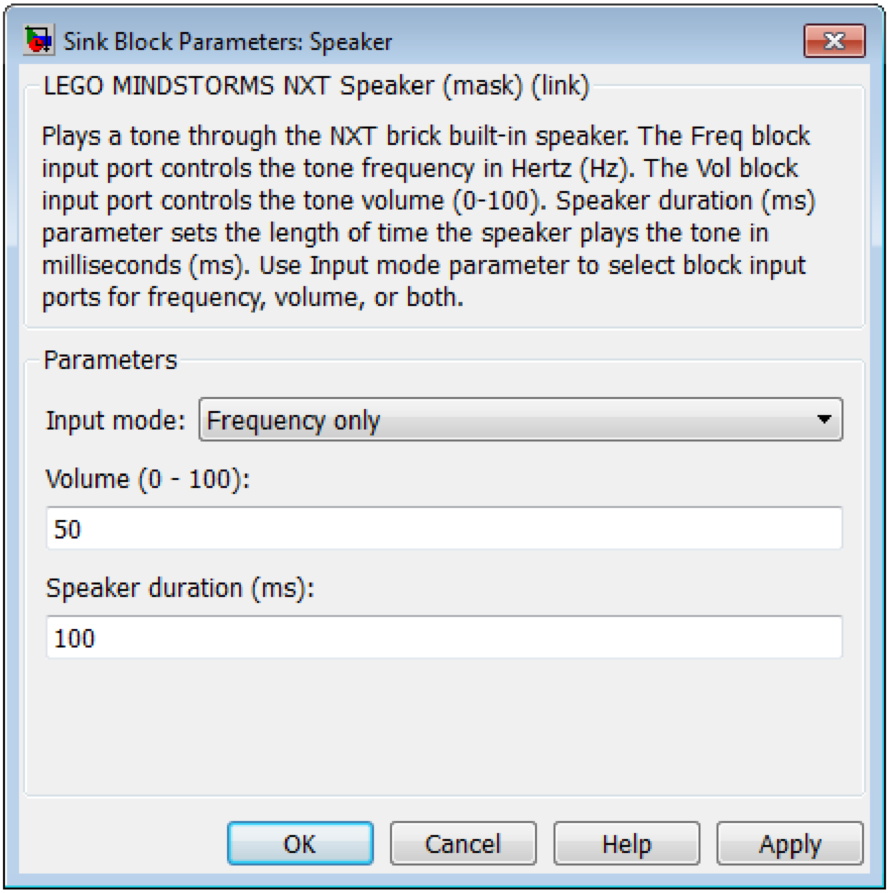
Plays a tone through the NXT brick built-in speaker. The Freq block input port controls the tone frequency in Hertz (Hz). The Vol block input port controls the tone volume (0-100). Speaker duration (ms) parameter sets the length of time the speaker plays the tone in milliseconds (ms). Use Input mode parameter to select block input ports for frequency, volume, or both.

Parameters

Input mode: Frequency and volume

Speaker duration (ms):

100



LEGO MINDSTORMS NXT Speaker (mask) (link)

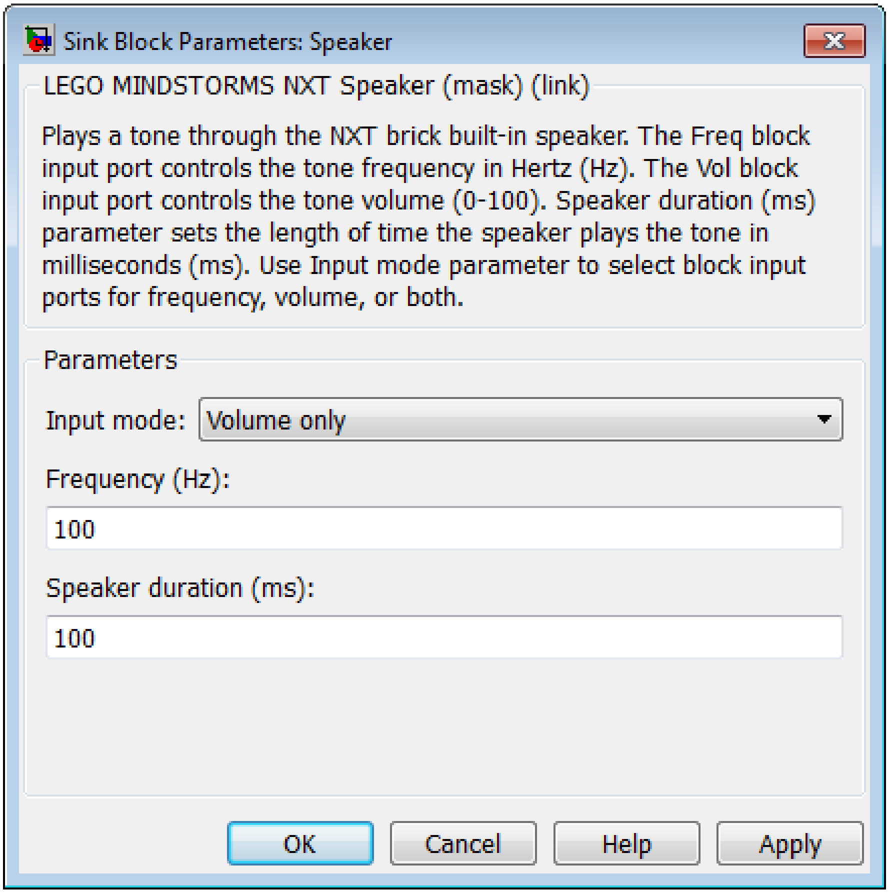
Plays a tone through the NXT brick built-in speaker. The Freq block input port controls the tone frequency in Hertz (Hz). The Vol block input port controls the tone volume (0-100). Speaker duration (ms) parameter sets the length of time the speaker plays the tone in milliseconds (ms). Use Input mode parameter to select block input ports for frequency, volume, or both.

Parameters

Input mode: Frequency only

Volume (0 - 100): 50

Speaker duration (ms): 100



LEGO MINDSTORMS NXT Speaker (mask) (link)

Plays a tone through the NXT brick built-in speaker. The Freq block input port controls the tone frequency in Hertz (Hz). The Vol block input port controls the tone volume (0-100). Speaker duration (ms) parameter sets the length of time the speaker plays the tone in milliseconds (ms). Use Input mode parameter to select block input ports for frequency, volume, or both.

Parameters

Input mode: Volume only

Frequency (Hz): 100

Speaker duration (ms): 100

Input mode

Enable or disable the block inputs:

- `Frequency and volume` — Enables both **Freq** and **Vol**.
- `Volume only` — Enables the **Vol** block input, disables the **Freq** block input, and enables the **Frequency (Hz)** parameter on the block mask.

- `Frequency only` — Enables the **Freq** block input, disables the **Vol** block input, and enables the **Volume (0 - 100)** parameter on the block mask.

Frequency (Hz)

Enter the frequency of the tone the speaker emits. When the **Input mode** parameter is configured to enable a **Freq** input, this parameter is not available.

Volume (0 - 100)

Set the volume of the tone the speaker emits. When the **Input mode** parameter is configured to enable a **Vol** input, this parameter is not available.

Speaker duration (ms)

Play a continuous tone or an intermittent tone (beeps). If this value is equal to or greater than the sample time of the source block, the speaker emits a continuous tone. If this value is shorter than the sample time of the source block, the speaker emits an intermittent tone at each sample period. For example, in the preceding ex_battery_voltage_alarm model, if the sample time for the battery is 0.1 seconds, and the speaker duration is 100 milliseconds (0.1 seconds), the speaker plays a continuous tone. If you change the speaker duration to 50 milliseconds (half the battery sample time), the speaker emits a beep every 0.1 seconds.

See Also

LEGO MINDSTORMS NXT Battery | LEGO MINDSTORMS NXT Button | LEGO MINDSTORMS NXT LCD | LEGO MINDSTORMS NXT Speaker | LEGO MINDSTORMS NXT Timer | “Install Support for LEGO MINDSTORMS NXT Hardware” |

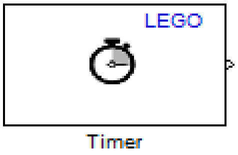
External Web Sites

- http://mindstorms.lego.com

LEGO MINDSTORMS NXT Timer

Purpose Measure elapsed time from timer in NXT brick

Library Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Description

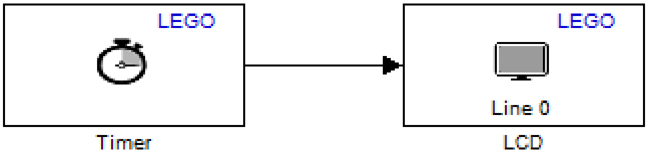
Measure the elapsed time, in milliseconds, from the time the NXT brick starts running the model.

With External mode disabled, the default setting, the NXT brick starts running the model when you press the right-arrow button on the NXT brick. With External mode enabled, the NXT brick automatically starts running the model when Simulink finishes downloading the model to the NXT brick.

You can see when the NXT brick starts running a model by observing the LCD on the NXT brick. For models that do not use an LCD block, the LCD on the NXT brick displays “I am running ...” followed by the model name. For models that use an LCD block, the model displays whatever output is sent to it.

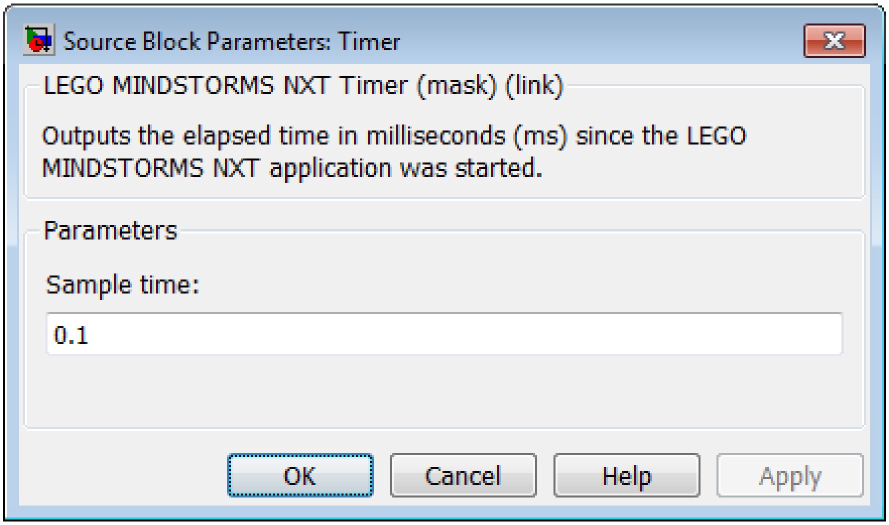
If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

Use the Timer



1. Open the `legonxtlib` block library and copy the blocks shown to a new model.
2. Connect the Timer block to the LCD block, as shown.
3. Prepare and run the model on the NXT brick.
4. Observe the elapsed time, in milliseconds, on the LCD display.

Dialog Box


Source Block Parameters: Timer

LEGO MINDSTORMS NXT Timer (mask) (link)

Outputs the elapsed time in milliseconds (ms) since the LEGO MINDSTORMS NXT application was started.

Parameters

Sample time: 0.1

Sample time

Specify how often the block reads timer values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

LEGO MINDSTORMS NXT Battery | LEGO MINDSTORMS NXT Button | LEGO MINDSTORMS NXT LCD | LEGO MINDSTORMS NXT Speaker | LEGO MINDSTORMS NXT Timer | “Install Support for LEGO MINDSTORMS NXT Hardware” | “Detect and Fix Task Overruns on NXT Brick” |

External Web Sites

- http://mindstorms.lego.com

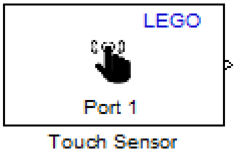
LEGO MINDSTORMS NXT Touch Sensor

Purpose

Output state of touch sensor

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware

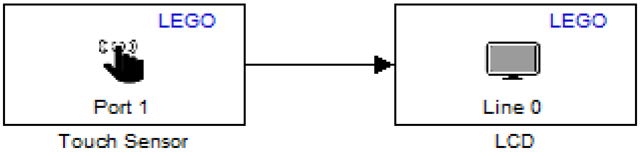


Description

This block outputs the state of the LEGO MINDSTORMS 9843 Touch Sensor: `1` while pressed, and `0` while released.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See “Block Produces Zeros in Simulation”.

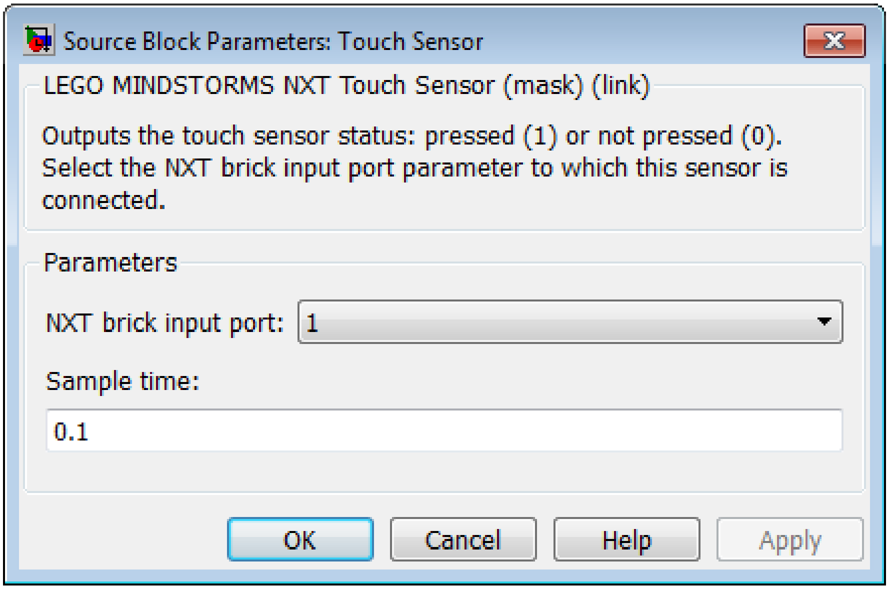
Use the Touch Sensor



1. Open the `legonxtlib` block library and copy the blocks shown to a new model.
2. Connect the Touch block to the LCD block, as shown.
3. Connect the touch sensor to Port 1 on the NXT brick.
4. Prepare and run the model on the NXT brick.
5. Observe the state of the touch sensor on the LCD display while you press and release the touch sensor.

LEGO MINDSTORMS NXT Touch Sensor

Dialog Box


Source Block Parameters: Touch Sensor
LEGO MINDSTORMS NXT Touch Sensor (mask) (link)

Outputs the touch sensor status: pressed (1) or not pressed (0). Select the NXT brick input port parameter to which this sensor is connected.

Parameters

NXT brick input port: 1

Sample time: 0.1

OK
Cancel
Help
Apply

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are `1`, `2`, `3`, or `4`.

Sample time

Specify how often the block reads sensor values.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

"Install Support for LEGO MINDSTORMS NXT Hardware" | "Detect and Fix Task Overruns on NXT Brick" |

External Web Sites

- http://mindstorms.lego.com

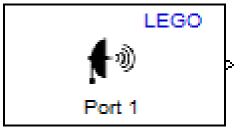
LEGO MINDSTORMS NXT Ultrasonic Sensor

Purpose

Measure distance from object in centimeters

Library

Simulink Support Package for LEGO MINDSTORMS NXT Hardware



Ultrasonic Sensor

Description

Measure the distance between the LEGO MINDSTORMS 9846 Ultrasonic Sensor and the nearest object in front of the sensor.

The sensor can detect objects from approximately 5 to 255 centimeters away. When the nearest object is beyond the maximum range of the Ultrasonic Sensor, the sensor outputs the maximum value, `255`.

The measured distances are approximate. For greater precision, calibrate the sensor output values against physical measurements.

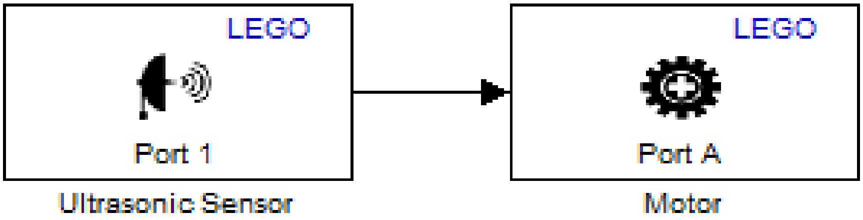
The distance from which the sensor first detects an approaching object depends on:

- The ultrasonic reflectance of the object, which is a function of the object's size and composition. The sensor detects large hard objects from a greater distance than small soft ones. For example, the sensor might detect a pane of glass at 255 cm and a hand puppet at 150 cm.
- The angle of incidence of the object relative to the sensor. The sensor detects objects directly in front of it at greater distances than objects off to the sides.

If you run a simulation of a model that contains this block, and the simulation does not run the model on the target hardware (e.g., External mode), this block outputs zeroes. See "Block Produces Zeros in Simulation".

Use the Ultrasonic Sensor

With the following blocks, the output from ultrasonic sensor drives the motor forward. As the ultrasonic sensor approaches an obstacle, the output values decrease, slowing the motor.



1. Open the `legonxtlib` block library and copy the blocks shown to a new model.

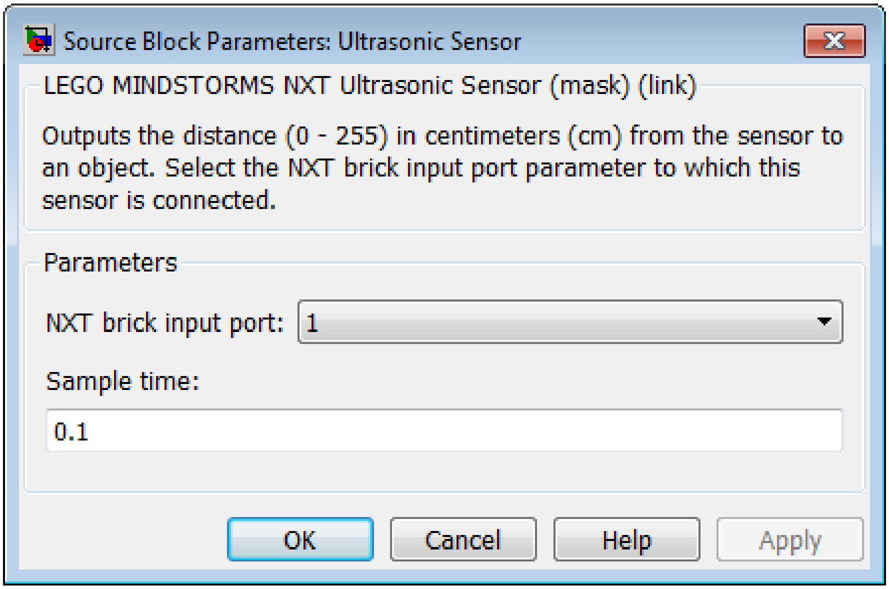
2. Connect the Ultrasonic Sensor block to the Motor block, as shown.

3. Connect the ultrasonic sensor to Port 1 on the NXT brick, and connect the servo motor to Port A on the NXT brick.

4. Prepare and run the model on the NXT brick.

5. Observe the speed of the motor decrease as you hold the ultrasonic sensor closer to an object.

Dialog Box



LEGO MINDSTORMS NXT Ultrasonic Sensor (mask) (link)

Outputs the distance (0 - 255) in centimeters (cm) from the sensor to an object. Select the NXT brick input port parameter to which this sensor is connected.

Parameters

NXT brick input port: 1

Sample time:

0.1

NXT brick input port

Select the NXT sensor port to which the sensor is connected. Avoid assigning multiple devices to the same port. The options are `1`, `2`, `3`, or `4`.

Sample time

Specify how often the block reads sensor values. This value defaults to `0.1`. Shorter sample times, such as `0.01`, may produce unreliable measurements.

Smaller values require the processor to complete the same number of instructions in less time, which can cause task overruns.

See Also

“Install Support for LEGO MINDSTORMS NXT Hardware” **|** “Detect and Fix Task Overruns on NXT Brick” **|**

External Web Sites

- http://mindstorms.lego.com

2

Functions — Alphabetical List

add_block

Purpose

Add block to model

Syntax

```
add_block('src', 'dest')
block = add_block('src', 'dest')
add_block('src', 'dest', 'param1', value1, ...)
add_block('src', 'dest', 'MakeNameUnique', 'on')
add_block('src_inport', 'dest_inport', 'CopyOption',
   'duplicate')
```

Description

`add_block('src', 'dest')` copies the block with the full path name that you specify with `'src'` to a new block with a full path name that you specify with `'dest'`. The block parameters of the new block are identical to those of the original. If the `'src'` block is a Subsystem block, then `add_block` copies all the blocks in the subsystem.

`block = add_block('src', 'dest')` returns the handle of the newly created block.

`add_block('src', 'dest', 'param1', value1, ...)` creates a copy of the `'src'` block, with the named parameters having the specified values. Additional arguments must occur in parameter/value pairs.

`add_block('src', 'dest', 'MakeNameUnique', 'on')` creates a unique name for the new block, based on the name of the `'dest'` block. The `add_block` function creates a unique name only if the `'dest'` block name exists in the model into which you add the new block. By default, `MakeNameUnique` is `off`.

`add_block('src_inport', 'dest_inport', 'CopyOption', 'duplicate')` applies only to Inport blocks. It creates a copy with the same port number as the `'src_inport'` block.

Before you add a block from a source model, open the source model. For example, use `load_system` (model opens invisibly) or `open_system` (model opens visibly). Also open the destination model if it is different from the source model.

Calling `add_block` triggers the `CopyFnc` and `PreCopyFcn` block callback functions.

You can use `'built-in/`*`blocktype`*`'` as a source block path name for Simulink built-in blocks, where *`blocktype`* is the built-in block type (that is, the value of its `BlockType` parameter (see "Common Block Properties" on page 6-96). However, using `'built-in/`*`blocktype`*`'` causes some default parameter values of some blocks to be different from the defaults that you get if you added those blocks interactively using Simulink.

Examples

Copy the Scope block from the Sinks subsystem of the `simulink` system to a block named `Scope` in the `Controller` subsystem of the `f14` system.

```
simulink;
open_system('f14');
add_block('simulink/Sinks/Scope', 'f14/Controller/Scope')
```

Create a subsystem named `Controller2` in the `f14` system. You do not have to open the Library Browser.

```
open_system('f14');
add_block('built-in/SubSystem', 'f14/Controller2')
```

Copy the built-in Gain block to a block named `Speed` in the `f14` system and assign the `Gain` parameter a value of 4. You do not have to open the Library Browser.

```
open_system('f14');
add_block('built-in/Gain', 'F14/Speed', 'Gain', '4')
```

Copy the block named `Mu` in `vdp` and create a copy. Because the model already contains a `Mu` block, the command names the new block `Mu1`. Open the `vdp` model, which is both the source and destination model, and get the handle of the added block.

```
open_system('vdp');
block = add_block('vdp/Mu', 'vdp/Mu', 'MakeNameUnique', 'on')
```

See Also

`delete_block` | `replace_block` | `set_param`

Tutorials

- “Create Annotations Programmatically”

How To

- “Add Blocks to a Model”
- “Create Model Callbacks”

add_exec_event_listener

Purpose

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.

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 string 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.

Purpose

Add line to Simulink system

Syntax

```
h = add_line('sys','oport','iport')
h = add_line('sys','oport','iport', 'autorouting','on')
h = add_line('sys', points)
```

Description

The `add_line` command adds a line to the specified system and returns a handle to the new line. You can define the line in two ways:

- By naming the block ports that are to be connected by the line
- By specifying the location of the points that define the line segments

`add_line('sys', 'oport', 'iport')` adds a straight line to a system from the specified block output port `'oport'` to the specified block input port `'iport'`. `'oport'` and `'iport'` are strings consisting of a block name and a port identifier in the form `'block/port'`. Most block ports are identified by numbering the ports from top to bottom or from left to right, such as `'Gain/1'` or `'Sum/2'`. Enable, Trigger, State, and Action ports are identified by name, such as `'subsystem_name/Enable'`, `'subsystem_name/Trigger'`, `'Integrator/State'`, or `if_action_subsystem_name/Ifaction'`.

`add_line('sys','oport','iport', 'autorouting','on')` works like `add_line('sys','oport','iport')` except that it routes the line around intervening blocks. The default value for autorouting is `'off'`.

`add_line(system, points)` adds a segmented line to a system. Each row of the `points` array specifies the x and y coordinates of a point on a line segment. The origin is the top-left corner of the window. The signal flows from the point defined in the first row to the point defined in the last row. If the start of the new line is close to the output of an existing block or line, a connection is made. Likewise, if the end of the line is close to an existing input, a connection is made.

Examples

This command adds a line to the `mymodel` system connecting the output of the Sine Wave block to the first input of the Mux block.

```
add_line('mymodel','Sine Wave/1','Mux/1')
```

This command adds a line to the `mymodel` system extending from (`20,55`) to (`40,10`) to (`60,60`).

```
add_line('mymodel',[20 55; 40 10; 60 60])
```

See Also

`delete_line`

Purpose

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 strings are case sensitive. The value of the parameter must be a string. 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 string values `'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`

addFile

Purpose Add file to Simulink Project

Syntax

```
addFile(proj,file)
```

Description `addFile(proj,file)` adds a file to the project `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.

file - Path of file
string

Path of the file to add relative to the project root folder, including the file extension, specified as a string. The file must be within the root folder.

Example: 'models/myModelName.slx'

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.

```
addFile(proj,'mymodel.slx');
```

See Also `simulinkprojectremoveFile`

addterms

Purpose 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`

Purpose

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`

How To

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

Purpose

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`

How To

- “Manage a Configuration Set”
- “Manage a Configuration Reference”

Purpose

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`.

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 within the root folder.

categoryName - Name of category for label

string

Name of the category for the label, specified as a string.

labelName - Name of label

string | label definition object

Name of the label to attach, specified as a string 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

string | numeric

Data to attach to the label, specified as a string or numeric. Data type depends on the label definition. Get a label to examine its `DataType` property using `file.Label` or `findLabel`.

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: [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')
```

Index into the Labels property to get the second label attached to this file.

```
reviewlabel = myfile.Labels(2)
```

```
reviewlabel =

  Label with properties:

            File: 'C:\Temp\project1\airframe\models\AnalogControl.mdl
            Data: []
        DataType: 'char'
            Name: 'To Review'
    CategoryName: 'Review'
```

Detach the new label from the file.

```
removeLabel(myfile,reviewlabel)
```

Attach a Label to a Subset of Files

Attach the label ’To Review’ in the ’Review’ category to all files in the project with 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 the file has the extension `.m`, attach the label `To Review`.

```
for fileIndex = 1:numel(files)
```

```
   file = files(fileIndex);
   [~, ~, fileExtension] = fileparts(file.Path);
   if strcmp(fileExtension,'.m')
       addLabel(file,'Review','To Review');
   end
end
```

In the Simulink Project Tool, the label `To Review` appears in the **Review** column next to the files in the project `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;
```

Get a particular file by name.

```
myfile = findFile(proj,'models/AnalogControl.mdl')
```

```
myfile =

  ProjectFile with properties:

      Path: [1x86 char]
    Labels: [1x1 slproject.Label]
```

Attach the label `'To Review'` and a string 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:\Temp\project1\airframe\models\AnalogControl.mdl
            Data: 'Whole team design review'
        DataType: 'char'
            Name: 'To Review'
    CategoryName: 'Review'
```

In the Simulink Project Tool, the label `To Review` and the label data appear in the **Review** column next to the file `AnalogControl.mdl`.

Alternatively, you can set or change label data using the data property.

```
mylabel = myfile.Labels(2);
mylabel.Data = 'Final review';
```

See Also

`simulinkprojectfindFilefindLabelcreateLabelremoveLabel`

Purpose

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`

Purpose

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 string or a cell array of strings. All strings must be valid block diagram names (which are the same as valid MATLAB variable 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

`find_system`

bdroot

Purpose Return name of top-level Simulink system

Syntax

```
bdroot
bdroot('obj')
```

Description `bdroot` with no arguments returns the top-level system name.

`bdroot('`*`obj`*`')`, where `'obj'` is a system or block pathname, returns the name of the top-level system containing the specified object name.

Examples This command returns the name of the top-level system that contains the current block.

```
bdroot(gcb)
```

See Also `find_system` | `gcb`

Purpose

Extract discrete-time linear state-space model around operating point

Syntax

```
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 string 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`

Purpose

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.

`'sys'` can be a string (which can be a system, a subsystem, or a full block pathname), a cell array of strings, 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` | `new_system` | `open_system` | `save_system`

closeDialog

Purpose

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`

How To

- “Manage a Configuration Set”
- “Manage a Configuration Reference”

Purpose Close Simulink Project

Syntax `close(proj)`

Description `close(proj)` closes the project `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 `simulinkproject`

coder.allowpcode

Purpose

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

`pcode` | `codegen`

Purpose Call external C/C++ function

Syntax

```
coder.ceval('cfun_name')
coder.ceval('cfun_name', cfun_arguments)
cfun_return = coder.ceval('cfun_name')
cfun_return = coder.ceval('cfun_name', cfun_arguments)
coder.ceval('-global','cfun_name',cfun_arguments)
cfun_return=coder.ceval('-global','cfun_name',cfun_arguments)
```

Description

`coder.ceval('`*`cfun_name`*`')` executes the external C/C++ function specified by the quoted string *`cfun_name`*. Define *`cfun_name`* in an external C/C++ source file or library.

`coder.ceval('`*`cfun_name`*`', `*`cfun_arguments`*`)` executes *`cfun_name`* with arguments *`cfun_arguments`*. *`cfun_arguments`* is a comma-separated list of input arguments in the order that *`cfun_name`* requires.

`cfun_return = coder.ceval('`*`cfun_name`*`')` 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.

`cfun_return = coder.ceval('`*`cfun_name`*`', `*`cfun_arguments`*`)` executes *`cfun_name`* with arguments *`cfun_arguments`* and returns *`cfun_return`*.

`coder.ceval('-global','`*`cfun_name`*`',`*`cfun_arguments`*`)`

`cfun_return=coder.ceval('-global','`*`cfun_name`*`',`*`cfun_arguments`*`)`

For code generation, you must specify the type, size, and complexity data type of return values and output arguments before calling `coder.ceval`.

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. In this case, if you do not use the `coder.ref`, `coder.rref`, and `coder.wref` constructs, a copy of the argument might appear in the generated code to enforce MATLAB semantics for arrays.

If you pass a global variable by reference using `coder.ref`, `coder.rref` or `coder.wref`, and the custom C code saves the address of this global variable, use the `-global` flag to synchronize for the variables passed to the custom C code. Synchronization occurs before and after calls to the custom code. If you do not synchronize global variables under these circumstances and the custom C code saves the address and accesses it again later, the value of the variable might be out of date.

Note The `-global` flag does not apply for MATLAB Function blocks.

You cannot use `coder.ceval` on functions that you declare extrinsic with `coder.extrinsic`.

Use `coder.ceval` only in MATLAB for code generation. `coder.ceval` generates an error in uncompiled MATLAB code. Use `coder.target` to determine if the MATLAB function is executing in MATLAB. If it is, do not use `coder.ceval` to call the C/C++ function. Instead, call the MATLAB version of the C/C++ function.

Examples

Call a C function `foo(u)` from a MATLAB function from which you intend to generate C code:

1. Create a C header file `foo.h` for a function `foo` that takes two input parameters of type `real_T` and returns a value of type `int32_T`.

   ```
   #include <tmwtypes.h>

   int32_T foo(real_T in1, real_T in2);
   ```

2. Write the C function `foo.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

int32_T foo(real_T in1, real_T in2)
{
  return in1 + in2;
}
```

3 Write a function `callfoo` that calls `foo` using `coder.ceval`.

```
function y = callfoo   %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
y = int32(0); % Constrain the return type to int32_T
if isempty(coder.target)
  % Executing in MATLAB, call MATLAB equivalent of
  % C function foo
  y = 10 + 20;
else
  % Executing in generated code, call C function foo
  y = coder.ceval('foo', 10, 20);
end
```

4 Generate C library code for function `callfoo`, passing `foo.c` and `foo.h` as parameters to include this custom C function in the generated code.

```
codegen -config:lib callfoo foo.c foo.h
```

`codegen` generates C code in the `codegen\lib\callfoo` subfolder.

```
int32_T callfoo(void)
{
    return foo(10.0, 20.0);
}
```

In this case, you have not specified the type of the input arguments, that is, the type of the constants `10` and `20`. Therefore, the arguments are implicitly of double-precision, floating-point type by default, because the default type for constants in MATLAB is `double`.

5 Cast the input arguments to specify their type explicitly.

```
y = coder.ceval('foo', int32(10), int32(20));
```

The generated code is:

```
int32_T callfoo(void)
{
   return foo(10, 20);
}
```

Call a C library function from MATLAB code:

1 Write a MATLAB function `absval`.

```
function y = absval(u)   %#codegen
y = abs(u);
```

2 Generate the C library for `absval.m`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib absval -args {0.0}
```

`codegen` creates the library `absval.lib` and header file `absval.h` in the folder `/codegen/lib/absval`. It also generates the functions `absval_initialize` and `absval_terminate` in the same folder.

3 Write a MATLAB function to call the generated C library functions using `coder.ceval`.

```
function y = callabsval  %#codegen
y = -2.75;
```

```
% Check the target. Do not use coder.ceval if callabsval is
% executing in MATLAB
if isempty(coder.target)
  % Executing in MATLAB, call function absval
  y = absval(y);
else
  % Executing in the generated code.
  % Call the initialize function before calling the
  % C function for the first time
  coder.ceval('absval_initialize');

  % Call the generated C library function absval
  y = coder.ceval('absval',y);

  % Call the terminate function after
  % calling the C function for the last time
  coder.ceval('absval_terminate');
end
```

4. Convert the code in `callabsval.m` to a MEX function so you can call the C library function `absval` directly from MATLAB.

```
codegen -config:mex callabsval codegen/lib/absval/absval.lib...
     codegen/lib/absval/absval.h
```

5. Call the C library by running the MEX function from MATLAB.

```
callabsval
```

See Also

`coder.ref` | `coder.rref` | `coder.wref` | `coder.target` | `codegen` | `coder.extrinsic`

Tutorials

- "Return Multiple Values from C Functions"

How To

- "Compilation Directive %#codegen"
- "Custom C/C++ Code Integration"

- “Data Definition Basics”

coder.cinclude

Purpose

Include header file in generated code

Syntax

```
coder.cinclude(AppHeaderFile)
coder.cinclude(SysHeaderFile)
```

Description

`coder.cinclude(AppHeaderFile)` includes an application header file in generated code using this format:

```
#include "HeaderFile"
```

`coder.cinclude(SysHeaderFile)` includes a system header file in generated code using this format:

```
#include <HeaderFile>
```

Limitations

- Do not call `coder.cinclude` inside run-time conditional constructs such as `if` statements, `switch` statements, `while`-loops, and `for`-loops. However, you can call `coder.cinclude` inside compile-time conditional statements, such as `coder.target`. For example:

  ```
  ...
  if ~isempty(coder.target)
     coder.cinclude('foo.h');
     coder.ceval('foo');
  end
  ...
  ```

Input Arguments

AppHeaderFile - Name of application header file
string

Name of an application header file, specified as a string. The header file must be located in the include path that you specify with the `-I` option when generating code using `codegen`.

Example: `coder.cinclude('myheader.h')`

Data Types
`char`

SysHeaderFile - Name of system header file

string

Name of a system header file, specified as a string enclosed in angle brackets `< >`. The header file must come from a standard list of system directories or from the include path that you specify with the `-I` option when generating code using `codegen`.

Example: `coder.cinclude('<stdio.h>')`

Data Types

`char`

Examples

Include Header File Conditionally in Generated Code

Generate code from a MATLAB function that calls an external C function to double its input. The MATLAB function uses `coder.cinclude` to include an application header file in generated C code running on a target machine, but not when the function runs in the MATLAB environment.

Write a C function `coder_cinclude_impl.c` that doubles its input.

```
#include "coder_cinclude_impl.h"
double coder_cinclude_impl(double u)
{
    return 2 * u;
}
```

The C function uses the application header file `coder_cinclude_impl.h`.

```
#if !defined(CODER_CINCLUDE_IMPL)
#define CODER_CINCLUDE_IMPL
extern double coder_cinclude_impl(double);
#endif
```

Write a MATLAB function that conditionally includes the application header file and calls the external C function.

```
function y = coder_cinclude_ex
```

```
%#codegen
    y = 21;
    if ~isempty(coder.target)
    % Running in generated code
        coder.cinclude('coder_cinclude_impl.h');
        y = coder.ceval('coder_cinclude_impl', y);
    else
    % Running in MATLAB
        y = y * 2;
    end
end
```

Compile the MATLAB function. Use the `-I` option to specify the path to the external header and C files. For example, the following command generates a C library that includes the external C and header file located in a subfolder called `cfiles`.

```
codegen -config:lib coder_cinclude -I cfiles
```

Here is the generated C code:

```
/*
 * coder_cinclude.c
 *
 * Code generation for function 'coder_cinclude'
 *
 * C source code generated on: Mon Dec 17 16:07:17 2012
 *
 */

/* Include files */
#include "rt_nonfinite.h"
#include "coder_cinclude.h"
#include "coder_cinclude_impl.h"

/* Function Definitions */
real_T coder_cinclude(void)
{
  return coder_cinclude_impl(21.0);
}

/* End of code generation (coder_cinclude.c) */
```

Besides the files that `coder.cinclude` specifies, `codegen` automatically includes the following files:

- Header file that defines the prototype for your entry-point function (in this case, `coder_cinclude.h`)
- `rt_nonfinite.h` (if you do not specify `SupportNonFinite=false` using `coder.config` when you compile the entry-point function).

Tips

- Call `coder.cinclude` before calling an external C/C++ function using `coder.ceval` to include in the generated code the header files required for the external function.
- Localize use of `coder.cinclude` at the call sites where you want to include each header file. Do not place all of your `coder.cinclude` calls in the top-level (entry-point) function unless you want to include the specified header files in every build.

See Also

`coder.ceval` | `coder.target`

coder.cstructname

Purpose Specify structure name in generated code

Syntax

```
coder.cstructname(structVar,'structName')
coder.cstructname(structVar,'structName','extern')
coder.cstructname(structVar,'structName','extern',Name,Value)
coder.cstructname(structType,'structName')
coder.cstructname(structType,'structName','extern')
coder.cstructname(structType,'structName','extern',Name,
   Value)
```

Description

`coder.cstructname(structVar,'structName')` allows you to specify the name of a structure in generated code. *`structVar`* is the structure variable. *`structName`* specifies the name to use for the structure variable `structVar` in the generated code. Use `coder.cstructname(structVar, 'structName')` in a function that is compiled using . You must call `coder.cstructname` before the first use of the structure variable in your function.

`coder.cstructname(structVar,'structName','extern')` declares an externally defined structure. It does not generate the definition of the structure type; provide it in a custom include file.

`coder.cstructname(structVar,'structName','extern',Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`coder.cstructname(structType,'structName')` returns a `coder.StructType` with the name `structName`. When the first argument is `structType`, `coder.cstructname` is a MATLAB function. You cannot use `coder.cstructname(structType, 'structName')` in a function that is compiled using . Use the returned type with the `-args` option.

`coder.cstructname(structType,'structName','extern')` returns a `coder.StructType` that uses an externally defined structure. Provide the structure definition in a custom include file.

`coder.cstructname(structType,'structName','extern',Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Tips

- `coder.cstructname(structVar, 'structName')` is ignored outside of code generation. Using `coder.cstructname` at the MATLAB command line and then calling `codegen` does not assign a name to a structure in the generated code. For example, if function `foo` does not use `coder.cstructname` to assign a name to structure `S`, the following commands do not assign the name `myStruct` to the structure variable `S` in generated code.

  ```
  coder.cstructname(S,'myStruct');
  codegen foo -args {S}
  ```

- To use `coder.cstructname` on arrays, use single indexing. For example, you cannot do `coder.cstructname(x(1,2))`. Instead, use single indexing, for example `coder.cstructname(x(n))`.
- Use of `coder.cstructname` with global variables is not supported.
- If you use `coder.cstructname` on an array, it sets the name of the base type of the array not the name of the array. Therefore, you cannot use `coder.cstructname` on the base element and then on the array. For example, the following code does not work because the second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previous `coder.cstructname`, `myStructName`.

  ```
  % Define scalar structure with field a
  myStruct = struct('a', 0);
  coder.cstructname(mystruct,'myStructName');
  % Define array of structure with field a
  myStructArray = repmat(myStruct,k,n);
  coder.cstructname(myStructArray,'myStructArrayName');
  ```

- If you are using custom structure types, specify the name of the header file that includes the external definition of the structure using the `HeaderFile` input argument.
- If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. To take advantage of target-specific function implementations that require data to be aligned, use the `Alignment` input argument.

Input Arguments

structName

The name to use for the structure in the generated code.

structType

`coder.StructType` object.

structVar

Structure variable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Alignment'

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary so it will not be matched by CRL functions that require alignment.

`Alignment` must be either `-1` or a power of `2` that is not greater than `128`.

Default: -1

'HeaderFile'

Name of the header file that contains the external definition of the structure, for example, `"mystruct.h"`.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` option, MATLAB Coder includes that header file exactly at the point where it is required.

Must be a non-empty string.

Examples

Apply `coder.cstructname` to Top-Level Inputs

Generate code for a MATLAB function that takes structure inputs.

1. Write a MATLAB function `topfun` that assigns the name `MyStruct` to its input parameter.

   ```
   function y = topfun(x)   %#codegen
   % Assign the name 'MyStruct' to the input variable in
   % the generated code
     coder.cstructname(x, 'MyStruct');
     y = x;
   end
   ```

2. Declare a structure `s` in MATLAB. `s` is the structure definition for the input variable `x`.

   ```
   s = struct('a',42,'b',4711);
   ```

3. Generate a MEX function for `topfun`, using the `-args` option to specify that the input parameter is a structure.

   ```
   codegen topfun.m -args { s }
   ```

`codegen` generates a MEX function in the default folder `codegen\mex\topfun`. The structure definition is in `topfun_types.h` in this folder.

```
typedef struct
{
    real_T a;
    real_T b;
} MyStruct;
```

Assign a Name to a Structure and Pass it to a Function

Assign the name `MyStruct` to the structure `structVar` and pass the structure to a C function `use_struct`.

1. Create a C header file `use_struct.h` for a function `use_struct` that takes a parameter of type `MyStruct`. Define a structure of type `MyStruct` in the header file.

```
#include <tmwtypes.h>

typedef struct MyStruct
{
    real_T s1;
    real_T s2;
} MyStruct;

void use_struct(struct MyStruct *my_struct);
```

2. Write the C function `use_struct.c`.

```
#include <stdio.h>
#include <stdlib.h>

#include "use_struct.h"

void use_struct(struct MyStruct *my_struct)
{
  real_T x = my_struct->s1;
  real_T y = my_struct->s2;
}
```

3 Write a MATLAB compliant function `m_use_struct` that declares a structure, assigns the name `MyStruct` to it, and then calls the C function `use_struct` using `coder.ceval`.

```
function m_use_struct   %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Declare a MATLAB structure
structVar.s1 = 1;
structVar.s2 = 2;

% Assign the name MyStruct to the structure variable.
% extern indicates this is an externally defined
% structure.
coder.cstructname(structVar, 'MyStruct', 'extern');

% Call the C function use_struct. The type of structVar
% matches the signature of use_struct.
% Use coder.rref to pass the the variable structVar by
% reference as a read-only input to the external C
% function use_struct
coder.ceval('use_struct', coder.rref(structVar));
```

4 Generate C library code for function `m_use_struct`, passing `use_struct.h` to include the structure definition.

```
codegen -config:lib m_use_struct use_struct.c use_struct.h
```

`codegen` generates C code in the default folder `codegen\lib\m_use_struct`. The generated header file `m_use_struct_types.h` in this folder does not contain a definition of the structure `MyStruct` because `MyStruct` is an external type.

Create a `coder.StructType` Object

Create a `coder.StructType` object specifying that it uses an externally-defined structure type.

```
S.a = coder.typeof(double(0));
S.b = coder.typeof(single(0));
T = coder.typeof(S);
T = coder.cstructname(T,'mytype','extern','HeaderFile','myheader.h');

T =

coder.StructType
   1x1 extern mytype (myheader.h) struct
      a: 1x1 double
      b: 1x1 single
```

| | | | |

Purpose

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`*.

By default, this option enables synchronization of global data between MATLAB and MEX functions at MEX function entry and exit, and before and after extrinsic calls. Use this option for maximum consistency between MATLAB and the MEX functions. If most extrinsic calls do not modify global data, but a few do, you can turn off synchronization before and after extrinsic calls. To do so, change the global synchronization mode to `At MEX-function entry and exit`.

Use the *–sync:on* option to turn on synchronization for 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 generation software generates code for the call to an extrinsic function, but does not generate 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.

Tips

- The code generation software 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 should declare extrinsic. This function opens the code generations readiness tool that detects code generation issues in your MATLAB code.

  During 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, a compiler error is issued from MATLAB.

Examples

The following code declares the MATLAB functions `patch` and `axis` 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 and axis as extrinsic

coder.extrinsic('patch', 'axis');

x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');
```

By declaring these functions extrinsic, you instruct the software not to compile or generate code for `patch` and `axis`. Instead it dispatches these functions to MATLAB for execution.
`coder.ceval | | coder.screener |`

-
- “Resolution of Function Calls in MATLAB Generated Code”
-
-
- “Restrictions on Extrinsic Functions for Code Generation”
-

Purpose Control inlining in generated code

Syntax

```
coder.inline('always')
coder.inline('never')
coder.inline('default')
```

Description

`coder.inline('always')` forces inlining of the current function in generated code.

`coder.inline('never')` prevents inlining of the current function in generated code. For example, you may want to prevent inlining to simplify the mapping between the MATLAB source code and the generated code.

`coder.inline('default')` uses internal heuristics to determine whether or not to inline the current function.

In most cases, the heuristics used produce highly optimized code. Use `coder.inline` only when you need to fine-tune these optimizations.

Place the `coder.inline` directive inside the function to which it applies. MATLAB Coder does not inline entry-point functions.

`coder.inline('always')` does not inline functions called from `parfor`-loops. MATLAB Coder does not inline functions into `parfor`-loops.

Examples

- "Preventing Function Inlining" on page 2-58
- "Using coder.inline In Control Flow Statements" on page 2-59

Preventing Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
  coder.inline('never');
  y = x;
end
```

Using 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 you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, 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('Can not divide by 0');
end

y = dividend / divisor;
```

coder.load

Purpose

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.

Input Arguments

filename - Name of file

string

Name of file, specified as a string 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'`

Data Types
`char`

var1,...,varN - Names of variables to load

string

Names of variables, specified as string constants. Use the `*` wildcard to match patterns.

Example: `load('myFile.mat','A*')` loads all variables in the file whose names start with `A`.

Data Types
`char`

expr1,...,exprN - Regular expressions indicating which variables to load

string

Regular expressions indicating which variables to load, specified as string constants.

Example: `load('myFile.mat', '^A', '^B')` loads only variables whose names begin with `A` or `B`.

Data Types
`char`

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.

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.

Limitations

- `coder.load` does not support loading objects.
- Arguments to `coder.load` must be compile-time constant strings.
- 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 generation software 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 generation software 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`

Concepts

- "Regular Expressions"

Purpose

Declare uninitialized variables

Syntax

`X = coder.nullcopy(A)`

Description

`X = coder.nullcopy(A)` copies type, size, and complexity of `A` to `X`, but does not copy element values. Preallocates memory for `X` without incurring the overhead of initializing memory.

`coder.nullcopy` does not support MATLAB classes as inputs.

Use With Caution

Use this function with caution. See "How to Eliminate Redundant Copies by Defining Uninitialized Variables".

Examples

The following example shows how to declare variable `X` as a 1-by-5 vector of real doubles without performing an unnecessary initialization:

```
function X = foo

N = 5;
X = coder.nullcopy(zeros(1,N));
for i = 1:N
   if mod(i,2) == 0
      X(i) = i;
   else
      X(i) = 0;
   end
end
```

Using `coder.nullcopy` with `zeros` lets you specify the size of vector `X` without initializing each element to zero.

How To

- "Eliminate Redundant Copies of Variables in Generated Code"
- "Eliminate Redundant Copies of Variables in Generated Code"

coder.opaque

Purpose Declare variable in generated code

Syntax

```
y = coder.opaque(type)
y = coder.opaque(type,value)
y = coder.opaque(type,'HeaderFile',HeaderFile)
y = coder.opaque(type,value,'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)` declares a variable `y` and specifies the initial value of `y` in the generated code.

`y = coder.opaque(type,'HeaderFile',HeaderFile)` declares a variable `y` and specifies the header file that contains the definition of `type`. The code generation software generates the `#include` statement for the header file where required in the generated code.

`y = coder.opaque(type,value,'HeaderFile',HeaderFile)` declares a variable `y` with the specified type, initial value, and header file in the generated code.

Input Arguments

type - Type of variable

string

Type of variable in generated code specified as a string 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 *'`

Data Types
`char`

value - Initial value of variable

string

Initial value of variable in generated code specified as a string 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 prior to 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 variable's address to an external function using `coder.wref`.

Example: `'NULL'`

Data Types
`char`

HeaderFile - Name of header file

string

Name of header file, specified as a string constant, that contains the definition of `type`.

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 generation software generates double quotes.

Example: `'foo.h'` generates `#include "foo.h"`

Specify the include path in the build configuration parameters.

Example: `cfg.CustomInclude = 'c:\myincludes'`

Data Types
`char`

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 intial 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
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `valtest`.

```
codegen -report -config cfg valtest
```

`codegen` generates C code in the `codegen\lib\valtest` folder.

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 intial 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 string
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;
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `filetest`.

```
codegen -report -config cfg filetest
```

`codegen` generates C code in the `codegen\lib\filetest` folder.

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
```

Tips

- `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 the empty string.
- 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 2-71
- 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
  ```

See Also

`coder.ceval` | `coder.ref` | `coder.rref` | `coder.wref`

Purpose

Pass argument by reference as read input or write output

Syntax

`[y =] coder.ceval('`*`function_name`*`',` **`coder.ref(`*`arg`*`)`**`, ... `u_n`)`

Arguments

`arg`

Variable passed by reference as an input or an output to the external C/C++ function called in `coder.ceval`. *`arg`* must be a scalar variable, a matrix variable, or an element of a matrix variable.

Description

`[y =] coder.ceval('`*`function_name`*`',` **`coder.ref(`*`arg`*`)`**`, ... `u_n`)` passes the variable *`arg`* by reference as an input or an output to the external C/C++ function called in `coder.ceval`. You add `coder.ref` inside `coder.ceval` as an argument to *`function_name`*. The argument list can contain multiple `coder.ref` constructs. Add a separate `coder.ref` construct for each argument that you want to pass by reference to *`function_name`*.

Only use `coder.ref` in MATLAB code that you have compiled with `codegen`. `coder.ref` generates an error in uncompiled MATLAB code.

Examples

In the following example, a MATLAB function `fcn` has a single input `u` and a single output `y`. `fcn` calls a C function `my_fcn`, passing `u` by reference as an input. The value of output `y` is passed to `fcn` by the C function through its `return` statement.

Here is the MATLAB function code:

```
function y = fcn(u) %#codegen

y = 0; %Constrain return type to double
y = coder.ceval('my_fcn', coder.ref(u));
```

The corresponding C function prototype looks like this:

```
real_T my_fcn(real_T *a)
```

In this example, the generated code infers the type of the input `u` from its definition in the parent model.

The C function prototype defines the input as a pointer because it is passed by reference.

The generated code cannot infer the type of the output `y`, so you must set it explicitly—in this case to a constant value 0 whose type defaults to `double`, matching the C type `real_T`. For a list of type mappings, see "Mapping MATLAB Types to C/C++".

See Also

`coder.ceval` | `coder.rref` | `coder.wref`

Purpose

Pass argument by reference as read-only input

Syntax

`[y =] coder.ceval('`*`function_name`*`',` **`coder.rref(`*`argI`*`)`**`, ...` u_n`)`

Arguments

`argI`

Variable passed by reference as a *read-only* input to the external C/C++ function called in `coder.ceval`.

Description

`[y =] coder.ceval('`*`function_name`*`',` **`coder.rref(`*`argI`*`)`**`, ...` u_n`)` passes the variable *`argI`* by reference as a *read-only* input to the external C/C++ function called in `coder.ceval`. You add `coder.rref` inside `coder.ceval` as an argument to *`function_name`*. The argument list can contain multiple `coder.rref` constructs. Add a separate `coder.rref` construct for each read-only argument that you want to pass by reference to *`function_name`*.

Caution

The generated code assumes that a variable passed by `coder.rref` is *read-only* and is optimized accordingly. Consequently, the C/C++ function must not write to the variable or results can be unpredictable.

Only use `coder.rref` in MATLAB code that you have compiled with `codegen`. `coder.rref` generates an error in uncompiled MATLAB code.

Examples

In the following example, a MATLAB function `fcn` has a single input `u` and a single output `y`. `fcn` calls a C function `foo`, passing `u` by reference as a read-only input. The value of output `y` is passed to `fcn` by the C function through its `return` statement.

Here is the MATLAB function code:

```
function y = fcn(u) %#codegen

y = 0; % Constrain return type to double
y = coder.ceval('foo', coder.rref(u));
```

The corresponding C function prototype looks like this:

```
real_T foo(real_T *a)
```

In this example, the generated code infers the type of the input `u` from its definition in the parent model.

The C function prototype defines the input as a pointer because it is passed by reference.

The generated code cannot infer the type of the output `y`, so you must set it explicitly—in this case to a constant value 0 whose type defaults to `double`, matching the C type `real_T`. For a list of type mappings, see “Mapping MATLAB Types to C/C++”.

See Also

`coder.ceval` | | `coder.opaque` | | `coder.ref` | | `coder.wref`

Purpose

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, such as recursion, cell arrays, nested functions, and function handles as code generation compliance issues and displays them in a report. If `fcn` calls other functions directly or indirectly that are not MathWorks® functions, `coder.screener` analyzes these functions too. It does not analyze MathWorks functions. `coder.screener` might not detect all code generation issues. Under certain circumstances, it might report false errors.

`coder.screener(fcn_1,...,fcn_n )` analyzes entry-point functions (`fcn_1,...,fcn_n`).

Tips

- Before using `coder.screener`, fix issues identified by the code analyzer.
- Before generating code, use `coder.screener` to check that a function is suitable for code generation. Fix all the issues that it detects.

Input Arguments

fcn

Name of entry-point MATLAB function that you want to analyze.

fcn_1,...,fcn_n

Comma-separated list of names of entry-point MATLAB functions that you want to analyze.

Examples

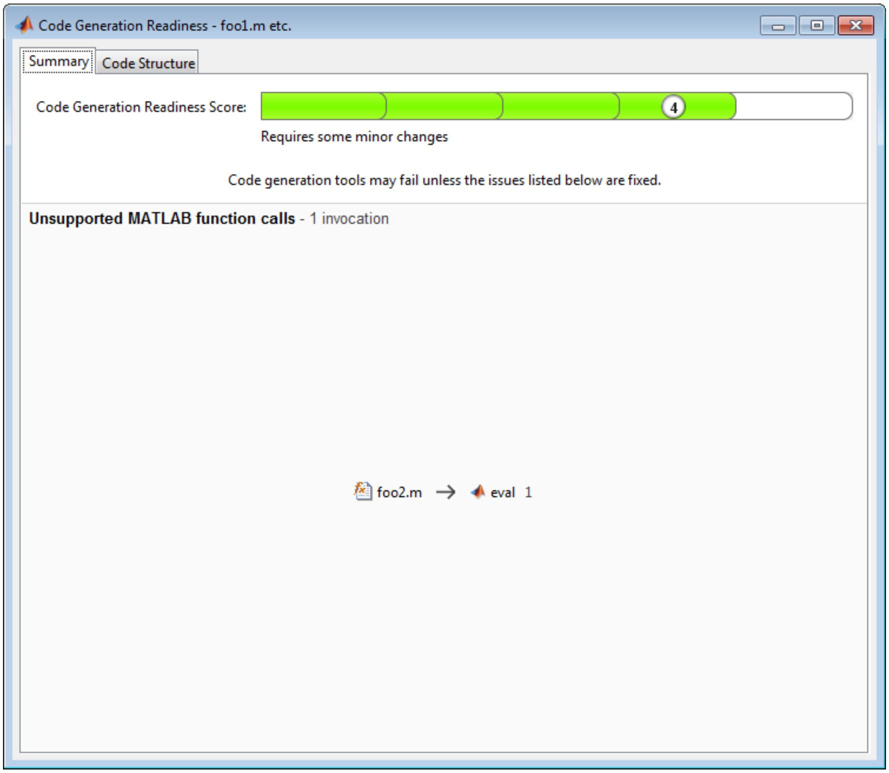
Identify Unsupported Functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks both the entry-point function, `foo1`, and the function `foo2` that `foo1` calls.

Analyze the MATLAB function `foo1` that calls `foo2`.

```
function out = foo1(in)
  out = foo2(in);
  disp(out);
end

function out = foo2(in)
  out = eval(in);
end

coder.screener('foo1')
```

The code generation readiness report opens. It provides a summary of the unsupported MATLAB function calls. The function `foo2` calls one unsupported MATLAB function.


Code Generation Readiness - foo1.m etc.
Summary
Code Structure
Code Generation Readiness Score:
4
Requires some minor changes

Code generation tools may fail unless the issues listed below are fixed.

Unsupported MATLAB function calls - 1 invocation

foo2.m → eval 1

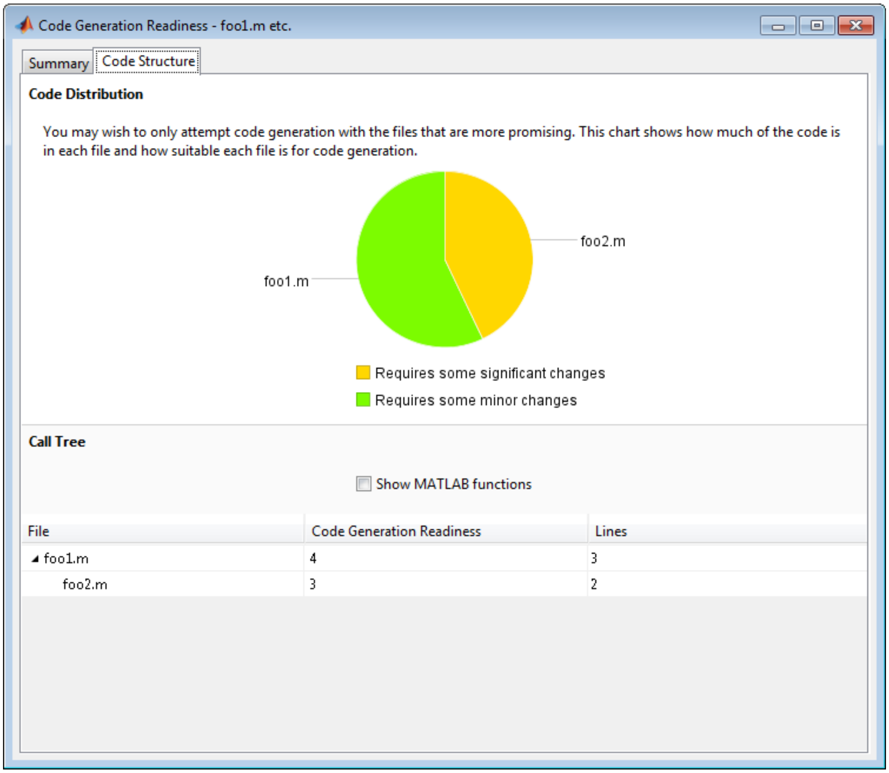
In the report, click the **Code Structure** tab and select **Show MATLAB functions**.

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 some significant changes are required.
- 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 to 5. 1 indicates that

significant changes are required; 5 indicates that the code generation readiness tool cannot 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
☐ Show MATLAB functions

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

File	Code Generation Readiness	Lines
◢ foo1.m	4	3
foo2.m	3	2

The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function which is not supported for code generation. 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 the `eval` function is not supported for code generation. When you generate a MEX function for `foo1`, the code generation software automatically calls out to MATLAB for `eval`. For standalone code generation, it does not generate code for it.

Identify Unsupported Data Types

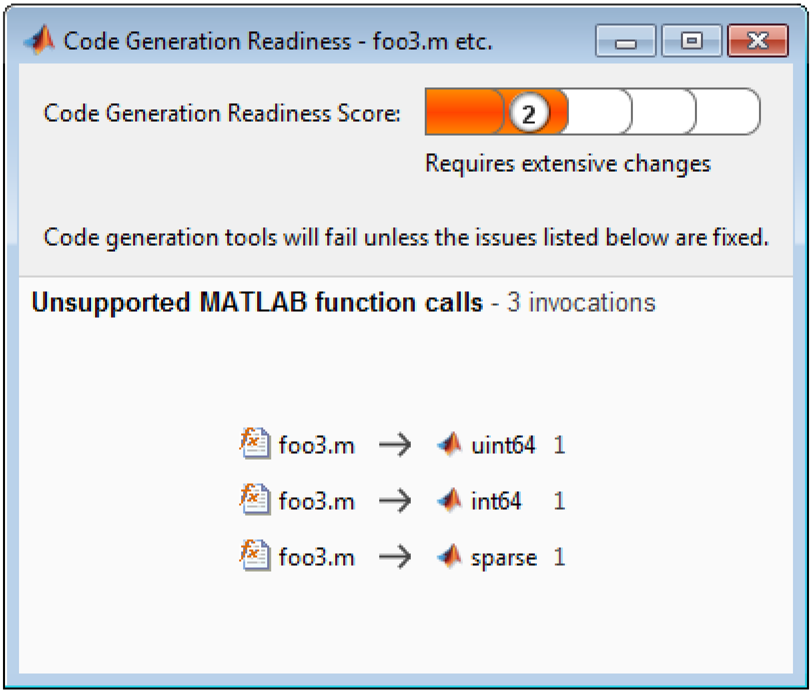
The `coder.screener` function identifies data types that are not supported for code generation.

Analyze the MATLAB function `foo3` that uses unsupported data types.

```
function [outInt64,outUint64,outSparse,outCasts] = foo3(inVal)
  outInt64 = int64(inVal);
  outUint64 = uint64(inVal);
  outSparse = sparse(inVal);
  outCasts = sparse(uint64(int64(inVal)));
end
```

```
coder.screener('foo3')
```

The code generation readiness report opens. It provides a summary of the unsupported data types.



Code Generation Readiness - foo3.m etc.

Code Generation Readiness Score: 2

Requires extensive changes

Code generation tools will fail unless the issues listed below are fixed.

Unsupported MATLAB function calls - 3 invocations

foo3.m → uint64 1

foo3.m → int64 1

foo3.m → sparse 1

The report assigns the code a code readiness score of 2, indicating that the code requires extensive changes.

Before generating code, you must fix the reported issues.

Determine code generation readiness for multiple entry-point functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks the entry-point functions `foo4` and `foo5`.

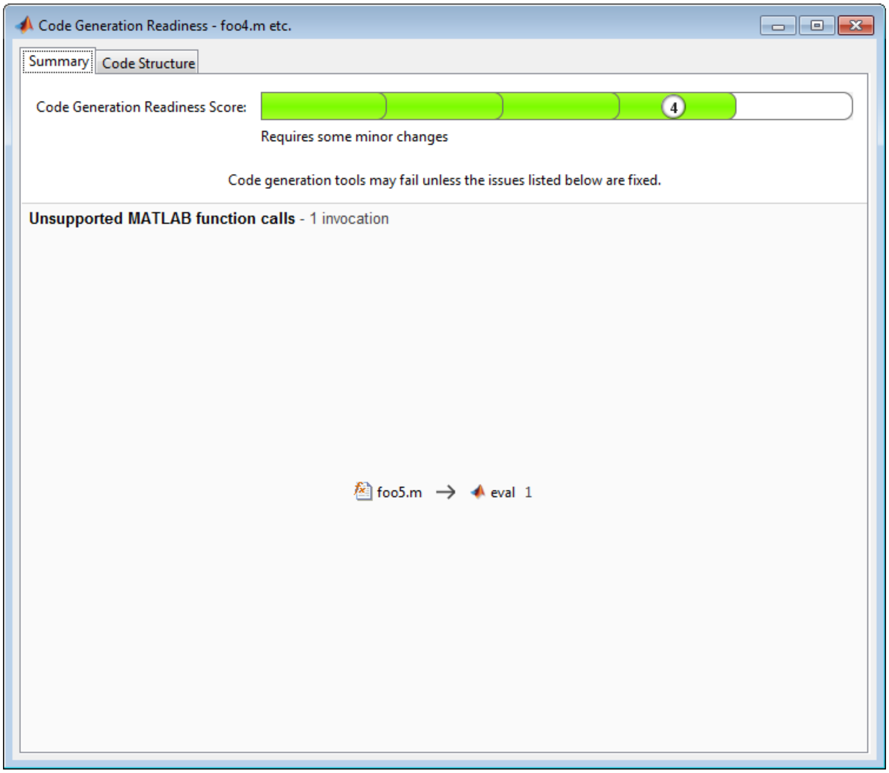
Analyze the MATLAB functions `foo4` and `foo5`.

```
function out = foo4(in)
  out = in;
  disp(out);
```

```
end

function out = foo5(in)
  out = eval(in);
end
```

```
coder.screener('foo4', 'foo5')
```

The code generation readiness report opens. It provides a summary of the unsupported MATLAB function calls. The function `foo5` calls one unsupported MATLAB function.


Code Generation Readiness - foo4.m etc.
Summary
Code Structure
4

Code Generation Readiness Score: 4

Requires some minor changes

Code generation tools may fail unless the issues listed below are fixed.

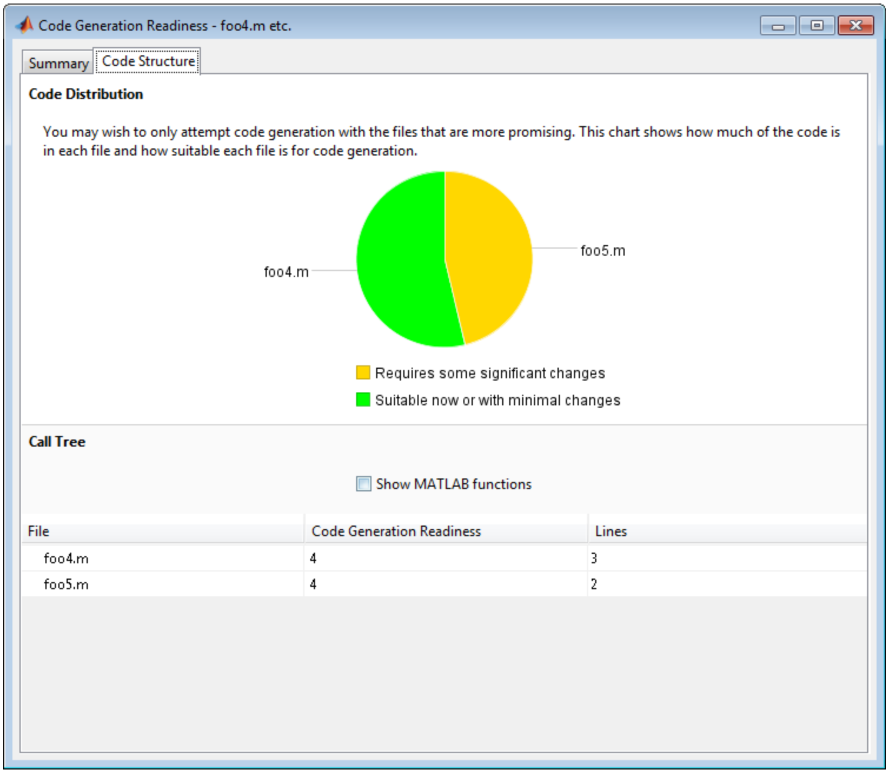
Unsupported MATLAB function calls - 1 invocation

foo5.m → eval 1

In the report, click the **Code Structure** tab and select **Show MATLAB functions**.

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 some significant changes are required.
- 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 to 5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool cannot detect issues.
- Displays a call tree.



Code Generation Readiness - foo4.m etc.

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.

🟨 Requires some significant changes

🟩 Suitable now or with minimal changes

Call Tree

☐ Show MATLAB functions

File	Code Generation Readiness	Lines
foo4.m	4	3
foo5.m	4	2

Alternatives

- “Run the Code Generation Readiness Tool From the Current Folder Browser”

Concepts

- “Functions Supported for Code Generation — Alphabetical List”
- “Functions Supported for Code Generation — Categorical List”
- “Code Generation Readiness Tool”

coder.target

Purpose Determine code generation target

Syntax `[y =] coder.target`

Description `[y =] coder.target` returns a string representing the code generation target.

String	Description
`''`	Function is executing in MATLAB
`'rtw'`	MATLAB Coder target
`'sfun'`	S-function target (Simulation target)
`'mex'`	MEX-function target
`'hdl'`	Stateflow HDL Coder target

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` returns `''`.

Examples Use `coder.target` to parameterize MATLAB functions that use custom C/C++ code so that they work in MATLAB or generated code.

```
if isempty(coder.target)
    % running in MATLAB
else
    % running in the generated code
end
```

.

Purpose

Convert a MATLAB value into its canonical type

Syntax

```
t=coder.typeof(v)
t=coder.typeof(v, sz, variable_dims)
t=coder.typeof(t)
```

Description

`t=coder.typeof(v)` creates a `coder.Type` object denoting the smallest non-constant type that contains `v`. `v` must be a MATLAB numeric, logical, char, enumeration, fixed-point array or a struct constructed from the preceding types. Use `coder.typeof` only to specify input parameter types.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `t`=coder.typeof(`v`) with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is [], the (upper bound) sizes of `v` remain unchanged. If the `variable_dims` input parameter is not specified, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it is applied to bounded dimensions or dimensions that are `1` or `0`, which are fixed.

`t=coder.typeof(t)`, where `t` is a `coder.Type` object, returns `t` itself.

Input Arguments

sz

Size vector specifying each dimension of type object

t

`coder.Type` object

v

MATLAB expression that describes the set of values represented by this type

`v` must be a MATLAB numeric, logical, char, enumeration, fixed-point array or a struct constructed from the preceding types.

variable_dims

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

Default: false(size(sz)) | sz==Inf

Output Arguments

t

`coder.Type` object

Examples

Create a type for a simple fixed-sized `5x6` matrix of doubles

```
coder.typeof(ones(5, 6))
 % returns 5x6 double
coder.typeof(0, [5 6])
 % also returns 5x6 double
```

Create a type for a variable-sized matrix of doubles

```
coder.typeof(ones(3,3), [], 1)
% returns :3 x :3 double
% ':' indicates variable-size dimensions
```

Create a type for a structure with a variable-size field

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
% Returns
% coder.StructType
%    1x1 struct
%      a:  :3x:5 double
%      b:  3x3  double
% ':' indicates variable-size dimensions
```

Create a type for a matrix with fixed-size and variable-size dimensions.

```
coder.typeof(0, [2,3,4], [1 0 1]);
% Returns :2x3x:4 double
% ':' indicates variable-size dimensions
```

```
coder.typeof(10, [1 5], 1)
% returns double 1 x  :5
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed size

```
coder.typeof(10,[inf,3])
% returns double:inf x 3
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable size with an upper bound of `3`

```
coder.typeof(10, [inf,3],[0 1])
% returns double :inf x :3
% ':' indicates variable-size dimensions
```

Convert a fixed-sized matrix to a variable-sized matrix

```
 coder.typeof(ones(5,5), [], 1)
% returns double :5x:5
% ':' indicates variable-size dimensions
```

```
| |
```

coder.unroll

Purpose

Copy body of `for`-loop in generated code for each iteration

Syntax

```
for i = coder.unroll(range)
for i = coder.unroll(range,flag)
```

Description

`for i = coder.unroll(range)` copies the body of a `for`-loop (unrolls a `for`-loop) in generated code for each iteration specified by the bounds in *`range`*. *`i`* is the loop counter variable.

`for i = coder.unroll(range,flag)` unrolls a `for`-loop as specified in *`range`* if *`flag`* is `true`.

You must use `coder.unroll` in a `for`-loop header. `coder.unroll` modifies the generated code, but does not change the computed results.

`coder.unroll` must be able to evaluate the bounds of the `for`-loop at compile time. The number of iterations cannot exceed 1024; unrolling large loops can increase compile time significantly and generate inefficient code

This function is ignored outside of code generation.

Input Arguments

flag

Boolean expression that indicates whether to unroll the `for`-loop:

`true`	Unroll the `for`-loop
`false`	Do not unroll the `for`-loop

range

Specifies the bounds of the `for`-loop iteration:

`init_val : end_val`	Iterate from *`init_val`* to *`end_val`*, using an increment of 1
`init_val : step_val : end_val`	Iterate from *`init_val`* to *`end_val`*, using *`step_val`* as an increment if positive or as a decrement if negative
Matrix variable	Iterate for a number of times equal to the number of columns in the matrix

Examples

To limit the number of times to copy the body of a `for`-loop in generated code:

1. Write a MATLAB function `getrand(n)` that uses a `for`-loop to generate a vector of length `n` and assign random numbers to specific elements. Add a test function `test_unroll`. This function calls `getrand(n)` with `n` equal to values both less than and greater than the threshold for copying the `for`-loop in generated code.

```
function [y1, y2] = test_unroll() %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
  % Calling getrand 8 times triggers unroll
  y1 = getrand(8);
  % Calling getrand 50 times does not trigger unroll
  y2 = getrand(50);

function y = getrand(n)
  % Turn off inlining to make
  % generated code easier to read
  coder.inline('never');

  % Set flag variable dounroll to repeat loop body
  % only for fewer than 10 iterations
  dounroll = n < 10;
```

```
% Declare size, class, and complexity
% of variable y by assignment
y = zeros(n, 1);
% Loop body begins
for i = coder.unroll(1:2:n, dounroll)
    if (i > 2) && (i < n-2)
        y(i) = rand();
    end;
end;
% Loop body ends
```

2 In the default output folder, `codegen/lib/test_unroll`, generate C static library code for `test_unroll` :

```
codegen -config:lib test_unroll
```

In `test_unroll.c`, the generated C code for `getrand(8)` repeats the body of the `for`-loop (unrolls the loop) because the number of iterations is less than 10:

```
static void m_getrand(real_T y[8])
{
  int32_T i0;
  for(i0 = 0; i0 < 8; i0++) {
    y[i0] = 0.0;
  }
  /*  Loop body begins */
  y[2] = m_rand();
  y[4] = m_rand();
  /*  Loop body ends */
}
```

The generated C code for `getrand(50)` does not unroll the `for`-loop because the number of iterations is greater than 10:

```
static void m_b_getrand(real_T y[50])
{
  int32_T i;
```

```
    for(i = 0; i < 50; i++) {
      y[i] = 0.0;
    }
    /*  Loop body begins */
    for(i = 0; i < 50; i += 2) {
      if((i + 1 > 2) && (i + 1 < 48)) {
        y[i] = m_rand();
      }
    }
    /*  Loop body ends */
  }
```

| | | `for`

How To

- “Using Logicals in Array Indexing”
-

coder.varsize

Purpose

Declare variable-size data

Syntax

`coder.varsize('`var_1`', '`var_2`', ...)`
`coder.varsize('`var_1`', '`var_2`', ..., `*`ubound`*`)`
`coder.varsize('`var_1`', '`var_2`', ..., `*`ubound`*`, `*`dims`*`)`
`coder.varsize('`var_1`', '`var_2`', ..., [], `*`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`'` must be a quoted string that represents a variable or structure field. If the structure field is a structure array, use colon (`:`) as the index expression, indicating that elements of the array are variable sized. 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-sized 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-sized 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 may add the declaration before the first assignment to each 'var_n'.

`coder.varsize` cannot be applied to global variables.

`coder.varsize` is not supported for MATLAB class properties.

`coder.varsize` is ignored outside of code generation.

Examples

Develop a simple stack that varies in size up to 32 elements as you push and pop data at run time.

1. Write primary function `test_stack` to issue commands for pushing data on and popping data from a stack. Write local function `stack` to execute the push and pop commands.

```
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

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

2 Generate a MEX function for `test_stack`:

generates a MEX function in the current folder.

3 Run `test_stack` 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.

1. 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-sized the field `values` inside each element of the matrix `data`.

2. Generate a MEX function for `struct_example`:

3. Run `struct_example`.

   Each time you run `struct_example` you get a different answer because the function loads the array with random numbers.

Alternatives

You can use the `assert` function to constrain an upper bound within a range of values, such as when growing a variable in a loop.

See Also

`assert` | | | `size` | `varargin`

-
-
-
-
-
- "Incompatibilities with MATLAB in Variable-Size Support for Code Generation"

Purpose

Pass argument by reference as write-only output

Syntax

`[y =] coder.ceval('`*`function_name`*`',`
`coder.wref(`*`argO`*`)``, ... `*u_n*`);`

Arguments

`argO`

Variable passed by reference as a *write-only* output to the external C/C++ function called in `coder.ceval`.

Description

`[y =] coder.ceval('`*`function_name`*`',` **`coder.wref(`*`argO`*`)`**`, ...` *u_n*`);` passes the variable *`argO`* by reference as a *write-only* output to the external C/C++ function called in `coder.ceval`. You add `coder.wref` inside `coder.ceval` as an argument to *`function_name`*. The argument list can contain multiple `coder.wref` constructs. Add a separate `coder.wref` construct for each write-only argument that you want to pass by reference to *`function_name`*.

Caution

The generated code assumes that a variable passed by `coder.wref` is *write-only* and optimizes the code accordingly. Consequently, the C/C++ function must write to the variable. If the variable is a vector or matrix, the C/C++ function must write to *every* element of the variable. Otherwise, results are unpredictable.

Only use `coder.wref` in MATLAB code that you have compiled with `codegen`. `coder.wref` generates an error in uncompiled MATLAB code.

Examples

In the following example, a MATLAB function `fcn` has a single input `u` and a single output `y`, a 5-by-10 matrix. `fcn` calls a C function `init` to initialize the matrix, passing `y` by reference as a write-only output. Here is the MATLAB function code:

```
function y = fcn(u)
%#codegen
```

```
y = zeros(5,10,'int8'); %Constrain output to an int8 matrix
coder.ceval('init', coder.wref(y));
```

The corresponding C function prototype looks like this:

```
void init(int8_T *x);
```

In this example:

- Although the C function is void, `coder.wref` allows it to access, modify, and return a matrix to the MATLAB function.
- The C function prototype defines the output as a pointer because it is passed by reference.
- For C/C++ code generation, you must set the type of the output `y` explcitly—in this case to an `int8` matrix, matching the C type `int8_T`. For a list of type mappings, see “Mapping MATLAB Types to C/C++”.
- The generated code collapses matrices to a single dimension.

See Also

`coder.ceval` | `coder.ref` | `coder.rref`

Purpose

Create category of Simulink Project labels

Syntax

```
createCategory(proj,categoryName)
createCategory(proj,categoryName,dataType)
```

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.

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

string

Name of the category of labels to create, specified as a string.

dataType - Class of data to store in labels

string

The class of data to store in labels in the new category, specified as a string.

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 string 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 = 'Maintainance 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
```

Tips

After you create a new category, you can create labels in the new category. See `createLabel`.

See Also

`simulinkprojectcreateLabel`

createLabel

Purpose

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.

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

string

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: 'Review'
            DataType: 'char'
    LabelDefinitions: [1x4 slproject.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat,'review me');
```

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 string 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 = 'Maintainance responsibility';
disp(label)
```

```
Label with properties:

            File: [1x80 char]
            Data: 'Maintainance responsibility'
        DataType: 'char'
            Name: 'Tom'
    CategoryName: 'Engineers'
```

See Also `addLabelcreateCategory`

Purpose

Delete block from Simulink system

Syntax

```
delete_block('blk')
```

Description

`delete_block('blk')`, where *`blk`* is a full block pathname, deletes the specified block from a system.

Examples

This command removes the Out1 block from the `vdp` system.

```
delete_block('vdp/Out1')
```

See Also

`add_block` | `replace_block`

delete_line

Purpose

Delete line from Simulink system

Syntax

```
delete_line('model', 'outPort', 'inPort')
delete_line('model', [x y])
delete_line(lineHandle)
```

Description

`delete_line('sys', 'oport', 'iport')` deletes the line extending from the specified block output port *`oport`* to the specified block input port *`iport`*. *`oport`* and *`iport`* are strings consisting of a block name and a port identifier in the form *`block/port`*. Most block ports are identified by numbering the ports from top to bottom or from left to right, such as `Gain/1` or `Sum/2`. Enable, Trigger, and State ports are identified by name, such as *`subsystem_name/Enable`*, *`subsystem_name/Trigger`*, *`Integrator/State`*, or *`if_action_subsystem_name/Ifaction`*.

`delete_line('sys', [x y])` deletes one of the lines in the system that contains the specified point (x,y), if any such line exists.

`delete_line(LineHandle)` deletes the line specified by the handle.

Examples

Remove Line Using Block Port Names

For the model `vdp`, remove the line connecting the Product block with the Gain block.

```
open_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.

```
open_system('vdp')
delete_line('vdp', 'Product/1', 'Mu/1')
LineHandle = add_line('vdp', 'Product/1', 'Mu/1')
delete_line(LineHandle)
```

See Also `add_line`

delete_param

Purpose 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`

dependencies.fileDependencyAnalysis

Purpose

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('model
   name', 'manifestfile')
```

Description

`files = dependencies.fileDependencyAnalysis('modelname')` returns `files`, a cell array of strings 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`.

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.

Input Arguments

modelname

String specifying the name of the model to analyze for dependencies.

manifestfile

(Optional) String 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 strings containing the full-paths of all existing files referenced by the model `modelname`.

Default: [ ]

missing

A cell array of strings containing the names of any files that are referenced by the model `modelname`, but cannot be found.

Default: [ ]

depfile

String 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

String 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.

Alternatives

You can interactively run dependency analysis from the Simulink project. See “Choose Files and 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`

Concepts

- “What Are Model Dependencies?”

dependencies.toolboxDependencyAnalysis

Purpose

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.

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 2-115.

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.

Input Arguments

files_in

Cell array of strings containing `.m` or `.mdl` 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
```

Alternatives

You can interactively run dependency analysis from the Simulink project. See “Choose Files and Run Dependency Analysis”.

To create a report to identify where dependencies arise, and for more control over dependency analysis options, you can interactively generate a manifest and report. See “Analyze Model Dependencies”.

To programmatically check which *files* are required, see `dependencies.fileDependencyAnalysis`.

See Also

`dependencies.fileDependencyAnalysis`

Concepts

- “What Are Model Dependencies?”

detachConfigSet

Purpose

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`

How To

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

removeLabel(was detachLabelFromFile)

Purpose

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`.

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`.

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'
```

See Also `simulinkprojectfindFilefindLabelcreateLabeladdLabel`

Purpose

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. 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 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`

How To

- "Data Validity Diagnostics Overview"
- "Symbol Resolution"

Purpose

Get or set editor invoked by Simulink DocBlock

Syntax

```
docblock('setEditorHTML', editCmd)
docblock('setEditorDOC', editCmd)
docblock('setEditorTXT', editCmd)
editCmd = docblock('getEditorHTML')
editCmd = docblock('getEditorDOC')
editCmd = docblock('getEditorTXT')
```

Description

`docblock('setEditorHTML', editCmd)` sets the HTML editor invoked by a DocBlock. The *`editCmd`* string specifies a command, executed at the MATLAB prompt, which launches a custom HTML editor. By default, a DocBlock invokes Microsoft Word (if available) as the HTML editor; otherwise, it opens HTML documents using the editor you specified on the **Editor/Debugger Preferences** pane of the MATLAB Preferences dialog box.

Use the `"%<FileName>"` token in the *`editCmd`* string to represent the full pathname to the document. Use the empty string `''` as the *`editCmd`* to reset the DocBlock to its default editor for a particular document type.

`docblock('setEditorDOC', editCmd)` sets the Rich Text Format (RTF) editor invoked by a DocBlock. The *`editCmd`* string specifies a command, executed at the MATLAB prompt, which launches a custom RTF editor. By default, a DocBlock invokes Microsoft Word (if available) as the RTF editor. Otherwise, it opens RTF documents using the editor you specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.

`docblock('setEditorTXT', editCmd)` sets the text editor invoked by a DocBlock. The *`editCmd`* string specifies a command, executed at the MATLAB prompt, which launches a custom text editor. By default, a DocBlock invokes the editor you specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.

`editCmd = docblock('getEditorHTML')` returns the value of the current command used to invoke an HTML editor when double-clicking a DocBlock.

editCmd = `docblock('getEditorDOC')` returns the value of the current command used to invoke a RTF editor when double-clicking a DocBlock.

editCmd = `docblock('getEditorTXT')` returns the value of the current command used to invoke a text editor when double-clicking a DocBlock.

Examples

Specify Microsoft Notepad as the DocBlock editor for RTF documents:

```
docblock('setEditorRTF','system(''notepad "%<FileName>"'');')
```

Reset the DocBlock to use its default editor for RTF documents:

```
docblock('setEditorRTF','')
```

Specify Mozilla Composer as the HTML editor for the DocBlock:

```
docblock('setEditorHTML','system(''/usr/local/bin/mozilla ...
  -edit "%<FileName>" &'');')
```

See Also

DocBlock

How To

- “Use a Simulink DocBlock to Add a Comment”

Purpose

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 selecting **New > Simulink Project > From Archive** in the Simulink Project Tool.

`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.

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

string

Zip file name or path, specified as a string 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'`

definitionType - Definition file type, specified as `slproject.DefinitionFiles.SingleFile` or `slproject.DefinitionFiles.MultiFile`.

slproject.DefinitionFiles.SingleFile | slproject.DefinitionFiles.MultiFile

Definition file type. 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.

Example:
`export(proj,'proj.zip',slproject.DefinitionFiles.SingleFile)`

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')
```

Related Examples

- "Archive Projects in Zip Files"

Purpose

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`.

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

string

Name of the category to get, specified as a string.

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.

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,'Review')
```

```
category =
  Category with properties:

                Name: 'Review'
            DataType: 'char'
    LabelDefinitions: [1x5 slproject.LabelDefinition]
```

Alternatively, you can examine categories by index. Get the first category.

```
proj.Categories(1)
```

```
ans =

  Category with properties:

                Name: 'Review'
            DataType: 'char'
    LabelDefinitions: [1x4 slproject.LabelDefinition]
```

Find out what you can do with the category.

```
methods(category)
```

```
Methods for class slproject.Category:

createLabel          findLabel  removeLabel
```

See Also `simulinkprojectcreateLabelremoveLabel`

Purpose

Get Simulink Project file by name

Syntax

`file = findFile(proj,fileName)`

Description

`file = findFile(proj,fileName)` returns a specific project file by name. You need to get a file before you can query labels, or use `addLabel` or `removeLabel`.

Input Arguments

proj - Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

fileName - Path of file

string

Path of the file to find, including any subfolders under the project root, specified as a string.

Output Arguments

file - Project file

file object

Project file, returned as a file object that you can query or modify.

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]
```

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:

ProjectFile          addLabel    removeLabel  findLabel
```

Alternatively, you can find project files by their labels using `findLabel`.

See Also `simulinkproject` `findCategory` `findLabel` `addLabel` `removeLabel`

Purpose

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`. 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.

Input Arguments

file - File to search labels of

file object

File to search the labels of, specified as a file object. 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 within the root folder.

categoryName - Name of category

string

Name of the parent category for the label, specified as a string.

labelName - Name of label

string

Name of the label to get, specified as a string.

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.

Examples

Find Files with the Label To Review

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 `To Review`.

```
for fileIndex = 1:numel(files)
   file = files(fileIndex);
   [~, ~, fileExtension] = fileparts(file.Path);
   if strcmp(fileExtension,'.m')
       addLabel(file,'Review','To Review');
   end
end
```

Find all the files with the label `To Review` and add them to a list returned in `files_to_review`.

```
files_to_review = {};
for jj=1:numel(files)
    this_file = files(jj);

       label = findLabel(this_file,'Review','To Review');

       if ( ~isempty(label))
        % This is a file labeled 'To Review'. Add to the
        % list of files to review.
        files_to_review = [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,'Review','To Review');
```

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 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);
```

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 = findLabel(category,'To Review')
```

```
ld = 

  LabelDefinition with properties:

             Name: 'To Review'
```

```
                 CategoryName: 'Review'
```

See Also `simulinkprojectfindFilecreateLabeladdLabel`

findLabelDefinition(renamed to findLabel)

Purpose

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.

Input Arguments

labelName - Name of label

string

Name of the label to get, specified as a string.

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.

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');
```

See Also `simulinkprojectfindCategorycreateLabeladdLabel`

Purpose

Find Model blocks and referenced models at all levels or at top level only

Syntax

```
[refMdls, mdlBlks] = find_mdlrefs('modelName')
[refMdls, mdlBlks] = find_mdlrefs('modelName', allLevels)
[refMdls, mdlBlks] = find_mdlrefs('modelName', 'Param1',
   Val1, ... ,'ParamN', ValN)
```

Description

`[refMdls, mdlBlks] = find_mdlrefs('modelName')` finds all Model blocks and referenced models contained by the model hierarchy of which *`modelName`* is the top model.

`[refMdls, mdlBlks] = find_mdlrefs('modelName', allLevels)` If *`allLevels`* is `true`, this is equivalent to the preceding syntax. If *`allLevels`* is `false`, the function searches only the top level of *`modelName`*, ignoring any subordinate hierarchy.

`[refMdls, mdlBlks] = find_mdlrefs('modelName', 'Param1', Val1, ... ,'ParamN', ValN)` searches the model as specified by the optional name/value pairs *`'Param1', Val1, ... , 'ParamN', ValN`*.

Input Arguments

modelName

The model in which to search for Model blocks and referenced models.

allLevels

Boolean specifying whether to search the complete hierarchy (`true`) or only the top model *`modelName`* (`false`). Default: `true`.

'Param1', Val1, ... , 'ParamN', ValN

One or more name/value pairs that control the actions of the function. The possible names and values, and their defaults, are:

`'AllLevels'`	Boolean specifying whether to search the complete hierarchy (`true`) or only the top level (`false`). The default if you omit `'AllLevels'` is `true`: the function searches the complete hierarchy of which *`modelName`* is the top model.
`'IncludeProtectedModels'`	Boolean that specifies whether the output *`refMdls`* includes the names of protected models (`true`) or excludes them(`false`). The default if you omit `'IncludeProtectedModels'` is `false`: the function excludes protected models from *`refMdls`*.
`'Variants'`	String that specifies whether the output *`refMdls`* includes the names of variant models. The possible values are: • `'ActiveVariants'` — Include the active variant of each Model Variants block included in the search. • `'ActivePlusCodeVariants'` — Include all variants for all Model Variants blocks included in the search that have **Generate preprocessor conditionals** option selected. • `'AllVariants'` — Include all variants for all Model Variants blocks included in the search.

The default if you
omit `'Variants'` is
`'ActivePlusCodeVariants'`.

The `find_mdlrefs` function provides two different ways to search all levels of the model *`modelName`*. Both techniques give the same results, but only the name/value technique allows you to control inclusion of protected and variant models in *`refMdls`*.

Output Arguments

mdlBlks

Depending on the value of `'AllLevels'`, a list containing the names of all Model blocks:

- In the hierarchy of which *`modelName`* is the top model
- In *`modelName`* alone

refMdls

Depending on the value of `'AllLevels'`, a list containing the names of all models referenced by a Model block:

- In the hierarchy of which *`modelName`* is the top model
- In *`modelName`* alone

Optional inputs to `find_mdlrefs` control inclusion of protected and variant models. The last element of the `refMdls` is *`modelName`* itself.

Examples

Return in `MyMdlBlks` the names all Model blocks at the top level of `MyModel`. Return in `MyRefMdls` the names of all models referenced by Model blocks at the top level, including the active variant of every variant Model block:

```
[MyRefMdls, MyMdlBlks] = find_mdlrefs(MyModel, 'AllLevels',...
false, 'Variants', 'ActiveVariants')
```

See Also

Model | `view_mdlrefs`

How To

- “Model Reference”
- “Set Up Model Variants”
- “Protected Model”

Purpose

Find systems, blocks, lines, ports, and annotations

Syntax

```
find_system(sys, 'c1', cv1, 'c2', cv2,...'p1',
v1, 'p2', v2,...)
```

Description

`find_system(`*`sys`*`, '`*`c1`*`', `*`cv1`*`, '`*`c2`*`', `*`cv2`*`,...'`*`p1`*`', `*`v1`*`, '`*`p2`*`', `*`v2`*`,...)` searches the loaded systems or subsystems specified by *`sys`*, using the constraints specified by *`c1`*, *`c2`*, etc., and returns handles or paths to the objects whose parameters, *`p1`*, *`p2`*, etc., have the values, *`v1`*, *`v2`*, etc. *`sys`* can be a pathname (or cell array of pathnames), a handle (or vector of handles), or omitted. To search block dialog parameter values, specify `'BlockDialogParams'` as the parameter name.

Note All the search constraints must precede all the parameter-value pairs in the argument list.

If `sys` is a pathname or cell array of pathnames, `find_system` returns a cell array of pathnames of the objects it finds. If `sys` is a handle or a vector of handles, `find_system` returns a vector of handles to the objects that it finds. If `sys` is omitted, `find_system` searches all loaded systems and returns a cell array of pathnames.

Case is ignored for parameter names. Value strings are case sensitive by default (see the ’CaseSensitive’ search constraint for more information). Any parameters that correspond to dialog box entries have string values. See “Model Parameters” on page 6-2 and “Block-Specific Parameters” on page 6-109 for a list of model and block parameters.

You can specify any of the following search constraints.

Name	Value Type	Description
`’SearchDepth’`	`scalar`	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.
`'LookUnderMasks'`	`'none'`	Search skips masked subsystems.
	`{'graphical'}`	Search includes masked subsystems that have no workspaces and no dialogs. This is the default.
	`'functional'`	Search includes masked subsystems that do not have dialogs.
	`'all'`	Search includes all masked subsystems.
`'FollowLinks'`	`'on'| {'off'}`	If `'on'`, search follows links into library blocks. The default is `'off'`.
`'FindAll'`	`'on'| {'off'}`	If `'on'`, search extends to lines, ports, and annotations within systems. The default is `'off'`. Note that `find_system` returns a vector of handles when this option is `'on'`, regardless of the array type of `sys`.
`'CaseSensitive'`	`{'on'}| 'off'`	If `'on'`, search considers case when matching search strings. The default is `'on'`.

Name	Value Type	Description
`'RegExp'`	`'on'| {'off'}`	If `'on'`, search treats search expressions as regular expressions. Use to search for partial matches. The default is`'off'`. See “Searching with Regular Expressions” on page 2-147.
`'LoadFullyIfNeeded'`	`{'on'}| 'off'`	If `'on'`, attempts to load any partially loaded models. The default is`'on'`. If `'off'`, disables the model loading, for example to prevent load warnings.
`'Variants'`	`{'ActiveVariants'}`	(Default) 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.

The table indicates default constraint values in brackets. If you omit a `'constraint'`, `find_system` uses the default constraint value.

Examples

This command returns a cell array containing the names of all loaded systems and blocks.

```
find_system
```

This command returns the names of all loaded block diagrams.

```
open_bd = find_system('type', 'block_diagram')
```

This command returns the names of all Goto blocks that are children of the `Unlocked` subsystem in the `sldemo_clutch` system.

```
sldemo_clutch
find_system('sldemo_clutch/
Unlocked','SearchDepth',1,'BlockType','Goto')
```

These commands return the names of all Gain blocks in the `vdp` system having a `Gain` parameter value of 1.

```
vdp
gb = find_system('vdp', 'BlockType', 'Gain')
find_system(gb, 'Gain', '1')
```

The preceding two commands are equivalent to this command:

```
vdp
find_system('vdp', 'BlockType', 'Gain', 'Gain', '1')
```

These commands obtain the handles of all lines and annotations in the `vdp` system.

```
vdp
sys = get_param('vdp', 'Handle');
l = find_system(sys, 'FindAll', 'on', 'type', 'line');
a = find_system(sys, 'FindAll', 'on', 'type',
'annotation');
```

This command turns off the `LoadFullyIfNeeded` search constraint.

```
find_system(gcs,'LoadFullyIfNeeded','off','ParameterName','ParameterValue')
```

This command searches for partial matches for `X` within any block dialog parameter value:

```
find_system('Regexp', 'on', 'BlockDialogParams','X')
```

Searching Block Dialog Parameter Values

Use the parameter `'BlockDialogParams'` to search block dialog parameter values. For example, to find any block which has a dialog parameter whose value is the string `'X+1'`, enter:

```
find_system('BlockDialogParams','X+1')
```

To find any partial matches within parameter values, for any `find_system` command, first specify the option `'Regexp', 'on'`. For the example above, to find `X` within the parameter value `'X+1'`, enter:

```
find_system('Regexp', 'on', 'BlockDialogParams','X')
```

Searching with Regular Expressions

If you specify the `'RegExp'` constraint as `'on'`, `find_system` treats search value strings as regular expressions. A regular expression is a string of characters in which some characters have special pattern-matching significance. For example, a period (.) in a regular expression matches not only itself but any other character.

Regular expressions greatly expand the types of searches you can perform with `find_system`. For example, regular expressions allow you to do partial-word searches. You can search for all objects that have a specified parameter that contains or begins or ends with a specified string of characters.

To use regular expressions effectively, you need to learn the meanings of the special characters that regular expressions can contain. The following table lists the special characters supported by `find_system` and explains their usage.

Expression	Usage
`.`	Matches any character. For example, the string `'a.'` matches `'aa'`, `'ab'`, `'ac'`, etc.
`*`	Matches zero or more of preceding character. For example, `'ab*'` matches `'a'`, `'ab'`, `'abb'`, etc. The expression `'.*'` matches any string, including the empty string.

Expression	Usage
`+`	Matches one or more of preceding character. For example, `'ab+'` matches `'ab'`, `'abb'`, etc.
`^`	Matches start of string. For example, `'^a.*'` matches any string that starts with `'a'`.
`$`	Matches end of string. For example, `'.*a$'` matches any string that ends with `'a'`.
`\`	Causes the next character to be treated as an ordinary character. This escape character lets regular expressions match expressions that contain special characters. For example, the search string `'\\'` matches any string containing a `\` character.
`[]`	Matches any one of a specified set of characters. For example, `'f[oa]r'` matches `'for'` and `'far'`. Some characters have special meaning within brackets. A hyphen (`-`) indicates a range of characters to match. For example, `'[a-zA-Z1-9]'` matches any alphanumeric character. A circumflex (`^`) indicates characters that should not produce a match. For example, `'f[^i]r'` matches `'far'` and `'for'` but not `'fir'`.
`\w`	Matches a word character. (This is a shorthand expression for `[a-z_A-Z0-9]`.) For example, `'^\w'` matches `'mu'` but not `'&mu'`.
`\d`	Matches any digit (shorthand for `[0-9]`). For example, `'\d+'` matches any integer.
`\D`	Matches any nondigit (shorthand for `[^0-9]`).
`\s`	Matches a white space (shorthand for `[ \t\r\n\f]`).
`\S`	Matches a non white-space (shorthand for `[^ \t\r\n\f]`).
`\<WORD\>`	Matches `WORD` exactly, where `WORD` is a string of characters separated by white space from other words. For example, `'\<to\>'` matches `'to'` but not `'today'`.

To use regular expressions to search Simulink systems, specify the `'regexp'` search constraint as `'on'` in a `find_system` command and use a regular expression anywhere you would use an ordinary search value string.

For example, the following command finds all the inport and outport blocks in the `sldemo_clutch` model that is provided with Simulink software.

```
sldemo_clutch
find_system('sldemo_clutch', 'regexp', 'on', 'blocktype', 'port')
```

See Also

`get_param` | `set_param`

Purpose

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

`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.

`fixdt(Signed, WordLength, FractionLength)` returns a `Simulink.NumericType` object describing a fixed-point data type with binary point scaling.

`fixdt(Signed, WordLength, TotalSlope, Bias)` or `fixdt(Signed, WordLength, SlopeAdjustmentFactor, FixedExponent, Bias)` returns a `Simulink.NumericType` object describing a fixed-point data type with slope and bias scaling.

`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” in the Fixed-Point Designer documentation.

`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`

fixpt_evenspace_cleanup

Purpose

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`

Tutorials

- “Effects of Spacing on Speed, Error, and Memory Usage”
- “Create Lookup Tables for a Sine Function”

Purpose

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`

Tutorials

- “Producing Lookup Table Data”

fixpt_look1_func_approx

Purpose

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`

Tutorials

- “Producing Lookup Table Data”

How To

- “Use Lookup Table Approximation Functions”

fixpt_look1_func_plot

Purpose

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.

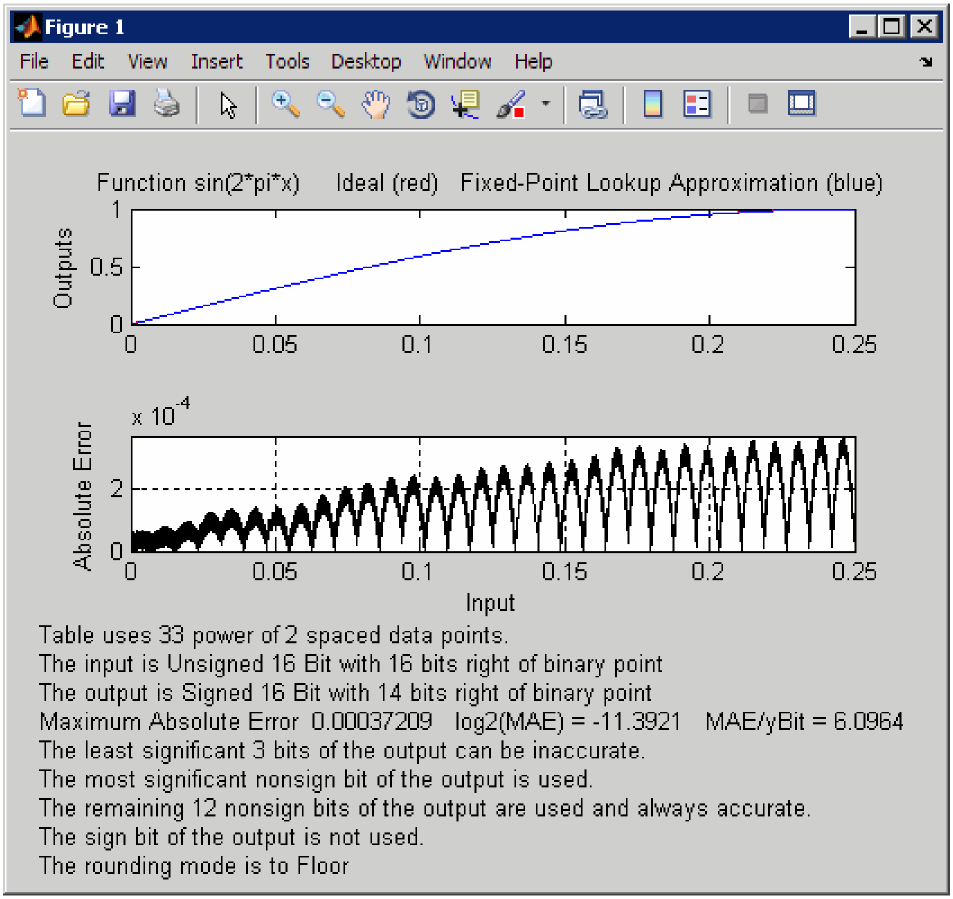


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
Function sin(2*pi*x)   Ideal (red)   Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0
0.05
0.1
0.15
0.2
0.25
x 10^-4
Absolute Error
2
0
0
0.05
0.1
0.15
0.2
0.25
Input

```
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`

Tutorials

- "Producing Lookup Table Data"

How To

- "Use Lookup Table Approximation Functions"

Purpose

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')
```

fixptbestexp

Purpose

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`

Purpose

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`

float

Purpose

Create MATLAB structure describing floating-point data type

Syntax

```
a = float('single')
a = float('double')
```

Description

`float('single')` returns a MATLAB structure that describes the data type of an IEEE single (32 total bits, 8 exponent bits).

`float('double')` returns a MATLAB structure that describes the data type of an IEEE double (64 total bits, 11 exponent bits).

`float` is automatically called when a floating-point number is specified in a block dialog box.

Examples

Define an IEEE single data type:

```
>> a = float('single')

a =

    Class: 'SINGLE'
```

See Also

`fixdt` | `sfix` | `sfrac` | `sint` | `ufix` | `ufrac` | `uint`

Purpose

Edit print frames for Simulink and Stateflow block diagrams

Syntax

```
frameedit
frameedit filename
```

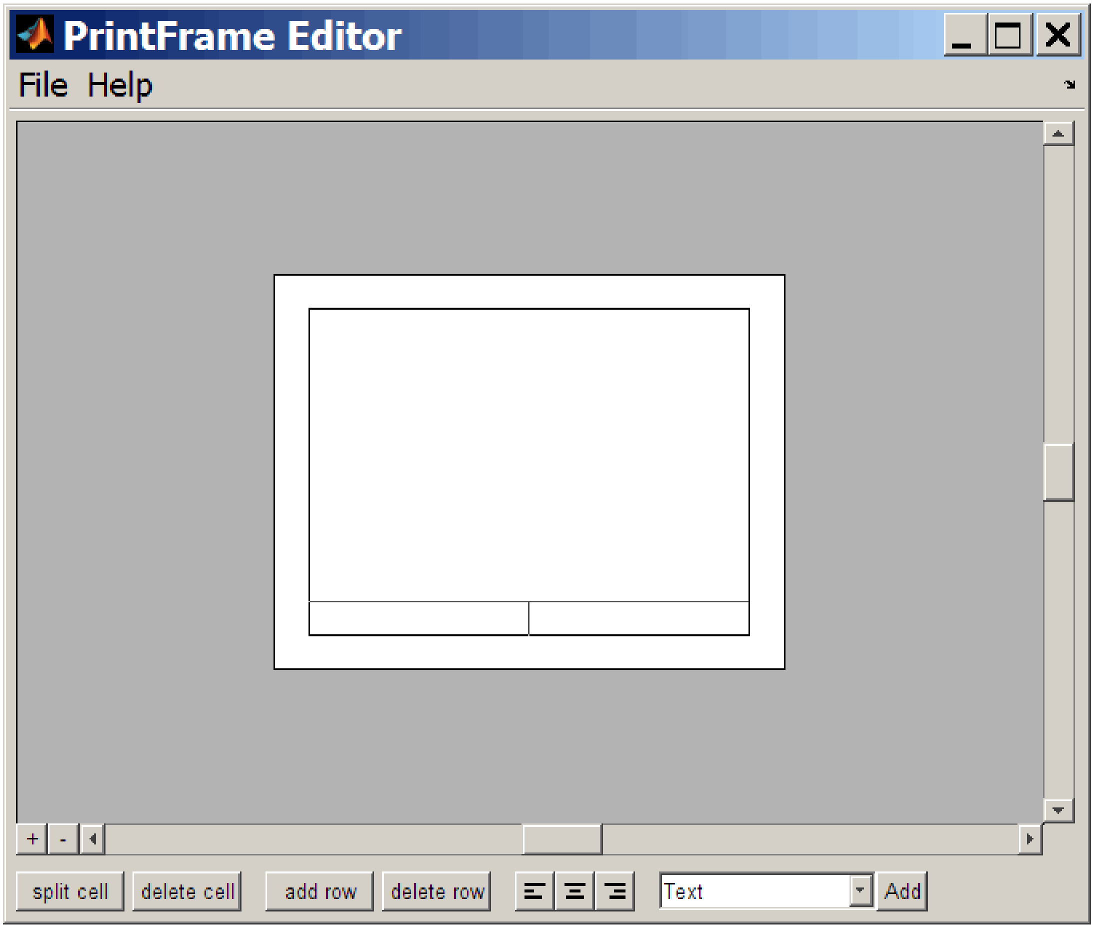
Description

`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.


PrintFrame Editor
File
Help
+
-
split cell
delete cell
add row
delete row
Text
Add

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.

fxptdlg

Purpose 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 **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, namely, `MinMaxOverflowArchiveMode`, `MinMaxOverflowLogging`, `DataTypeOverride`, and `DataTypeOverrideAppliesTo` (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 **Model Hierarchy** pane. If Fixed-Point Designer software is installed, the Fixed-Point Tool **Contents** pane 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. You can display an object's dialog box by right-clicking the appropriate entry in the **Contents** pane and selecting **Properties**.

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. In the **Model Hierarchy** pane, select the root model.

2. From the Simulink model **Analysis** menu, select **Fixed Point Tool**.

   The Fixed-Point Tool opens.

3. In the **Current system settings** panel:

   - Set the **Fixed-point instrumentation mode** parameter to `Force off`.

- Set the **Data type override** parameter to `Double` or `Single`.
- Set the **Data type override applies to** parameter to `All numeric types`.

4 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');
```

Parameters and Dialog Box

The Fixed-Point Tool includes the following components:

- **Main** toolbar (see “Main Toolbar” on page 2-180)
- **Model Hierarchy** pane (see “Model Hierarchy Pane” on page 2-185)
- **Contents** pane (see “Contents Pane” on page 2-186)
- **Dialog** pane (see “Dialog Pane” on page 2-193)

Main Toolbar

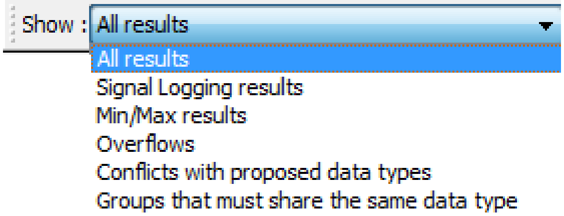
The Fixed-Point Tool’s main toolbar appears near the top of the Fixed-Point Tool window under the Fixed-Point Tool’s menu.

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.

Button	**Usage**
⏸	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.
ℹ	Show details for selected result.
∿	Compare selected runs.
⇹	Create a difference plot for the selected signals.
∿	Plot the selected signal.
📊	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. For more information, see "Plot Interface" on page 2-235.
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 dialog box to review these issues. ✖ The proposed data type will introduce errors if applied to this object. Open the Result Details dialog box 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

Show Option	Result
	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. Objects that control the **Fixed-point instrumentation mode** parameter display a red flag on their icons, while those that control the **Data type override** parameter display a green flag. 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`
`fo`	`Off`

See "Dialog Pane" on page 2-193 for more information about these parameters.

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 2-190.

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".
DerivedMax	Maximum value the Fixed-Point tool derives for this signal from design ranges specified for blocks.

Column Label	Description
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.

Column Label	Description
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.
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”.

Column Label	Description
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"
- "Changing Column Order and Width"
- "Sorting Rows by Column"

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.
`Automatic Data Typing View`	**Name**, **Run**, **CompiledDT**, **CompiledDesignMax**, **CompiledDesignMin**,**Accept**, **ProposedDT**, **SpecifiedDT**, **DesignMin**, **DesignMax**, **DerivedMin**,	After proposing data types if proposal is based on simulation, derived, and design min/max.

View Name	Columns Provided	When Does the Fixed-Point Tool Display this View?
	DerivedMax, **SimMin**, **SimMax**, **OverflowWraps**, **Saturations**, **ProposedMin**, **ProposedMax**	
`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.

View Name	**Columns Provided**	**When Does the Fixed-Point Tool Display this View?**
`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.

For more information on controlling views, see "Control Model Explorer Contents Using 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.

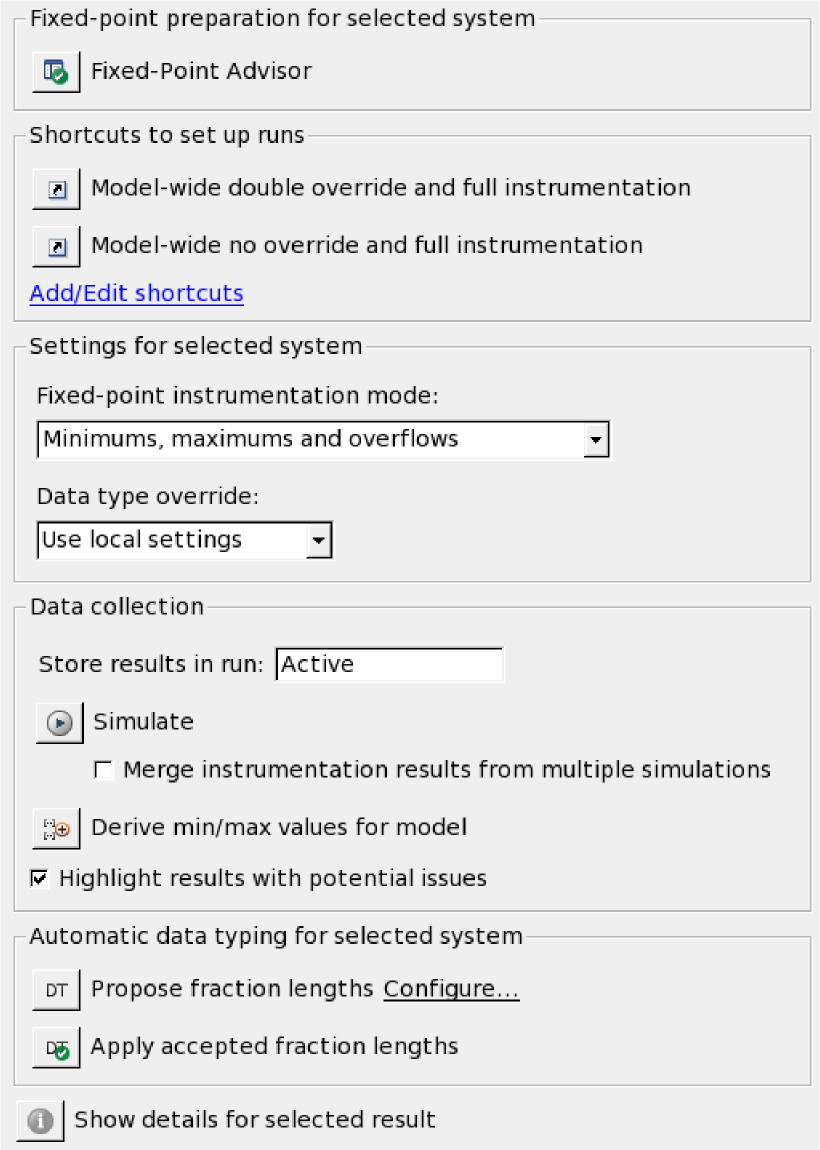
Sorting by Columns

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 object selected in the **Model Hierarchy** pane.



Fixed-point preparation for selected system

Fixed-Point Advisor

Shortcuts to set up runs

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

Store results in run: Active

Simulate

☐ Merge instrumentation results from multiple simulations

Derive min/max values for model

☑ Highlight results with potential issues

Automatic data typing for selected system

DT Propose fraction lengths Configure...

Apply accepted fraction lengths

Show details for selected result

The Dialog pane includes the following components:

Component	Description
Fixed-point preparation for selected system	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 2-197.
Shortcuts to set up runs	Contains shortcuts that allow you to set up run parameters, such as the run name and data type override settings, by clicking a button. The Fixed-Point Tool provides several factory default shortcuts for common configurations and allows you to set up your own shortcuts. For more information, see "Shortcuts to set up runs" on page 2-198.
Settings for selected system	Contains controls to set the fixed-point instrumentation mode and data type override settings. For more information, see "Fixed-point instrumentation mode" on page 2-201, "Data type override" on page 2-203, and "Data type override applies to" on page 2-206.
Data collection	Contains controls to collect simulation or derived minimum and maximum data for your model.

Component	Description
Automatic data typing for selected system	Contains controls to propose and, optionally, accept data type proposals.
Show details for selected result	Use this button to view data type information about the object selected in the **Contents** pane. For more information, see "Show details for selected result" on page 2-225.

Tips

From the Fixed-Point Tool **View** menu, you can customize the layout of the **Dialog** pane. Select:

- **Show Dialog View** to show/hide the **Dialog** pane. By default, the Fixed-Point Tool displays this pane. Hide it to make more room for displaying results.
- **Settings for selected system** to show/hide the **Settings for selected system** pane. By default, the Fixed-Point Tool displays this pane.
- **Show Fixed-Point Advisor** to show/hide the **Prepare conversion in the Fixed-Point Advisor** button. By default, the Fixed-Point Tool displays this button.

Fixed-Point Advisor

Open the Fixed-Point Advisor

Action

Opens the Fixed-Point Advisor which guides you through the tasks to prepare a floating-point model for conversion to fixed point.

Use the Fixed-Point Advisor if:

- You have a floating-point model
- You have inheritance rules in your model
- Your model contains blocks that do not support fixed-point data types

Shortcuts to set up runs

Use shortcuts to configure model-wide data type override and instrumentation settings prior to simulation. The Fixed-Point Tool provides:

- Frequently-used factory default shortcuts
- The ability to add and edit custom shortcuts

Note The factory default shortcuts apply to the whole model. You cannot use these shortcuts to configure subsystems.

Factory Defaults

Factory Default Shortcut	Description
Model-wide double override and full instrumentation	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 shortcut appears in the **Shortcuts to set up runs** pane.
Model-wide no override with full instrumentation	Sets: • **Run name** to `FixedPoint` • **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`

Factory Default Shortcut	Description
	• **Data type override** to `Use local settings` By default, a button for this shortcut appears in the **Shortcuts to set up runs** pane.
Model-wide single override and full instrumentation	Sets: • **Run name** to `SingleOverride` • **Fixed-point instrumentation mode** to `Minimums, maximums and overflows` • **Data type override** to `Single` • **Data type override applies to** to `All numeric types`

Factory Default Shortcut	Description
Model-wide instrumentation off	Sets **Fixed-point instrumentation mode** to `Force Off`. Does not affect run name or data type override settings.

Add/Edit shortcuts

Use **Add/Edit shortcuts** to open the "Shortcut Editor" on page 2-226, which you use to add new shortcuts or edit existing user-defined shortcuts. You cannot edit the factory default shortcuts.

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

- The Fixed-Point Tool marks the system controlling the **Fixed-point instrumentation mode** with a red flag 🚩.
- 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.
- The Fixed-Point Tool marks the system controlling the **Data type override** mode with a green flag ⚑.
- 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 setting`, 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

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

- The following Simulink blocks allow you to set data types in their block masks, but ignore the **Data type override** setting:
  - Probe
  - Trigger
  - Width
- Stateflow does not support the **Data type override** setting `Scaled double`.

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
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'`

Store results in run

Specifies the run name

If you use a shortcut to set up a run, by default, the Fixed-Point Tool uses the run name associated with this shortcut. 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.
- You can edit the run name in the Contents pane **Run** column.

Simulate

Simulates model and stores results.

Action

Simulates the model and stores the results with the run name specified in **Store results in run**. The Fixed-Point Tool displays the run name in the **Run** column of the **Contents** pane.

Tips

Using the **Simulate** button is the same as simulating the model using the **Run** button in the Simulink model.

Command-Line Alternative

Command: `sim`

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.

Derive min/max values 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 information based on the 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.

- **Name**
- **Run**
- **CompiledDesignMax**
- **CompiledDesignMin**
- **DerivedMax**
- **DerivedMin**
- **DesignMax**
- **DesignMin**

If **Highlight results with potential issues** is selected, the tool alerts you to potential issues for each object in the list by highlighting the object in the Contents pane.

Command-Line Alternative

No command line alternative available.

Dependencies

Range analysis:

- Requires a Fixed-Point Designer license.

- Does not run on Mac platforms.

Highlight results with potential issues

Highlight results that have saturations, overflows or insufficient design min/max information to derive results.

Settings

Default: On

☑ On

Highlight results with potential issues.

After simulating the model, the Fixed-Point Tool highlights results that have either saturations or overflows.

After analyzing the model to derive minimum and maximum values, the tool highlights both the results that have issues and the results that are likely to be the cause of the issue. For example, if the tool cannot derive ranges for an object because an upstream object does not have sufficient design minimum and maximum information, it highlights both results.

The Fixed-Point Tool uses different colors to highlight the results that have issues and the results that might be the cause of issues.

☐ Off

Do not highlight results. Remove existing highlighting.

Command-Line Alternative

No command line alternative available.

Tip

Select a highlighted result and click **Show details for selected result** to view more information about the issue.

Propose fraction lengths

Use range information and propose fraction lengths for model objects.

Action

Use this button to perform the first phase of the automatic data typing procedure, in which the Fixed-Point Tool uses range information for model objects—from design minimum and maximum values the objects specify explicitly, or from logged minimum and maximum values that occur during simulation, or from derived minimum and maximum values. Based on these values, the tool proposes fraction lengths for blocks whose:

- **Lock output data type setting against changes by the fixed-point tools** or **Lock data type settings against changes by the fixed-point tools** check box is cleared.
- Data type specifies a generalized fixed-point number.

The Fixed-Point Tool lists its scaling proposals in the **Contents** pane. The tool alerts you to potential scaling issues for each object in the list by displaying a green, yellow, or red icon, as shown here:

✓	The proposed scaling poses no issues for this object.
⚠	The proposed scaling poses potential issues for this object. Open the Autoscale Information dialog box to review these issues.
✗	The proposed scaling will introduce data type errors if applied to this object. Open the Autoscale Information dialog box for details about how to resolve these issues.

Command-Line Alternative

No command line alternative available.

Propose word lengths

Use range information and propose word lengths for model objects.

Action

Use this button to perform the first phase of the automatic data typing procedure, in which the Fixed-Point Tool uses range information for model objects—from design minimum and maximum values the objects specify explicitly, or from logged minimum and maximum values that occur during simulation, or from derived minimum and maximum values. Based on these values, the tool proposes word lengths for blocks whose:

- **Lock output data type setting against changes by the fixed-point tools** or **Lock data type settings against changes by the fixed-point tools** check box is cleared.
- Data type specifies a generalized fixed-point number.

The Fixed-Point Tool lists its data type proposals in the **Contents** pane. The tool alerts you to potential issues for each object in the list by displaying a green, yellow, or red icon, as shown here:

✓	The proposed data type poses no issues for this object.
⚠	The proposed data type poses potential issues for this object. Open the Result Details dialog box to review these issues.
✗	The proposed data type will introduce errors if applied to this object. Open the Result Details dialog box for details about how to resolve these issues.

Command-Line Alternative

No command line alternative available.

Propose data types

Propose fraction lengths for specified word lengths

Select to propose fraction lengths for the word lengths specified in the model.

Note This parameter appears only if the **Automatic data typing for selected system Settings** pane is displayed.

Propose word lengths for specified fraction lengths

Select to propose word lengths for the fraction lengths specified in the model.

Note This parameter appears only if the **Automatic data typing for selected system Settings** pane is displayed.

Default data type for all floating-point signals

Specifies the default data type to use for all floating-point signals

Use to change the floating-point data types in the model to fixed point.

Settings

Default:

`Remain floating-point`

Does not change the data type of floating-point signals.

Select this option if you are using the Fixed-Point Tool to derive minimum and maximum values for the model. To propose data types for the model, you must set the default data type to be a built-in integer or fixed-point data type.

`Same as embedded hardware integer`

Sets the default data type for all floating-point signals to the integer type specified for the target hardware.

Select this option if your target hardware is a microprocessor.

`int8`

Sets the default data type for all floating-point signals to `int8`.

`int16`

Sets the default data type for all floating-point signals to `int16`.

`int32`

Sets the default data type for all floating-point signals to `int32`.

`fixdt(1,16,4)`

Sets the default data type for all floating-point signals to the fixed-point data type specified by `fixdt`. You can modify the parameters provided to `fixdt` to specify signedness, word length and fraction length.

Dependency

This parameter appears only if you click the **Configure** link on the **Automatic data typing for selected system Settings** pane.

Derived min/max

Propose fraction or word lengths based on design and derived minimum and maximum information.

Settings

Default: On

☑ On

Propose fraction or word lengths based on derived minimum and maximum values, but only for blocks that do not specify design minimum or maximum values using, for example, **Output minimum** and **Output maximum** parameters.

If **Simulation min/max** is also selected, the Fixed-Point Tool bases its proposal on both the derived and simulation min/max values and the proposed data types cover the union of the derived range and the simulation range.

☐ Off

Ignore derived minimum and maximum values when proposing fraction lengths.

Command-Line Alternative

No command line alternative available.

Dependency

This parameter appears only if you click the **Configure** link on the **Automatic data typing for selected system Settings** pane.

Simulation min/max

Propose fraction or word lengths based on design and simulation minimum and maximum values.

Settings

Default: On

☑ On

Proposes fraction lengths based on simulation minimum and maximum values, but only for blocks that do not specify design minimum or maximum values using, for example, **Output minimum** and **Output maximum** parameters.

If **Derived min/max** is also selected, the Fixed-Point Tool bases its proposal on both the derived and simulation min/max values and the proposed data types cover the union of the derived range and the simulation range.

☐ Off

Ignores simulation minimum and maximum values when proposing fraction lengths.

Command-Line Alternative

No command line alternative available.

Dependency

This parameter appears only if you click the **Configure** link on the **Automatic data typing for selected system Settings** pane.

Safety margin for design and derived min/max (%)

Specify safety factor for design and derived minimum and maximum values.

Settings

Default: 0

The design and derived minimum and maximum values are adjusted by the percentage designated by this parameter. 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.

Dependency

Before performing autoscaling, 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 data.

Command-Line Alternative

No command line alternative available.

Dependency

This parameter appears only if you click the **Configure** link on the **Automatic data typing for selected system Settings** pane.

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. 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.

Dependency

This parameter appears only if you click the **Configure** link on the **Automatic data typing for selected system Settings** pane.

Apply accepted fraction lengths

Apply scaling proposals

Action

Use this button to perform the second phase of the automatic data typing procedure, in which the Fixed-Point Tool applies the scaling proposals to the objects whose **Accept** check box in the **Contents** pane is selected.

Dependency

This button appears when **Propose fraction lengths for specified word lengths** is selected.

Command-Line Alternative

No command line alternative available.

Apply accepted word lengths

Apply data type proposals

Action

Use this button to perform the second phase of the automatic data typing procedure, in which the Fixed-Point Tool applies the data type proposals to the objects whose **Accept** check box in the **Contents** pane is selected.

Dependency

This button appears when **Propose word lengths for specified fraction lengths** is selected.

Command-Line Alternative

No command line alternative available.

Show details for selected result

Display details for object selected in the **Contents** pane

Action

Use this option to:

- Determine why a fraction length cannot be proposed, for example, fraction lengths cannot be proposed for Library Links or Mask Subsystems.
- Provide access to showing blocks with shared data types.
- Obtain more information on the proposal for the selected result.

Command-Line Alternative

No command line alternative available.

Shortcut Editor

Shortcut Editor Overview

Use the Shortcut Editor to add or edit user-defined shortcuts. You cannot modify the factory default shortcuts. If you add a new shortcut and want it to appear as a button on the Fixed-Point Tool **Shortcuts to set up runs** pane, use the controls in the **Manage shortcuts** panel.

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.

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.

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 defaults to 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.

See Also

"Data type override" on page 2-203

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.

See Also

“Data type override applies to” on page 2-206

Manage shortcuts

Control which shortcuts appear as buttons on the Fixed-Point Tool **Shortcuts to set up runs** panel and the order in which they appear.

Plot Interface

The Fixed-Point Tool uses the Simulation Data Inspector tool plotting capabilities that enable you to plot signals for graphical analysis. The tool can access signal data that resides in the MATLAB workspace, allowing you to plot simulation results associated with:

- Scope blocks whose **Save data to workspace** parameter is selected

- To Workspace blocks
- Root-level Outport blocks, when the **Output** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box is selected
- Logged signal data

Tip The **Contents** pane of the Fixed-Point Tool displays an antenna icon next to items that you can plot.

Using the Simulation Data Inspector tool, you can:

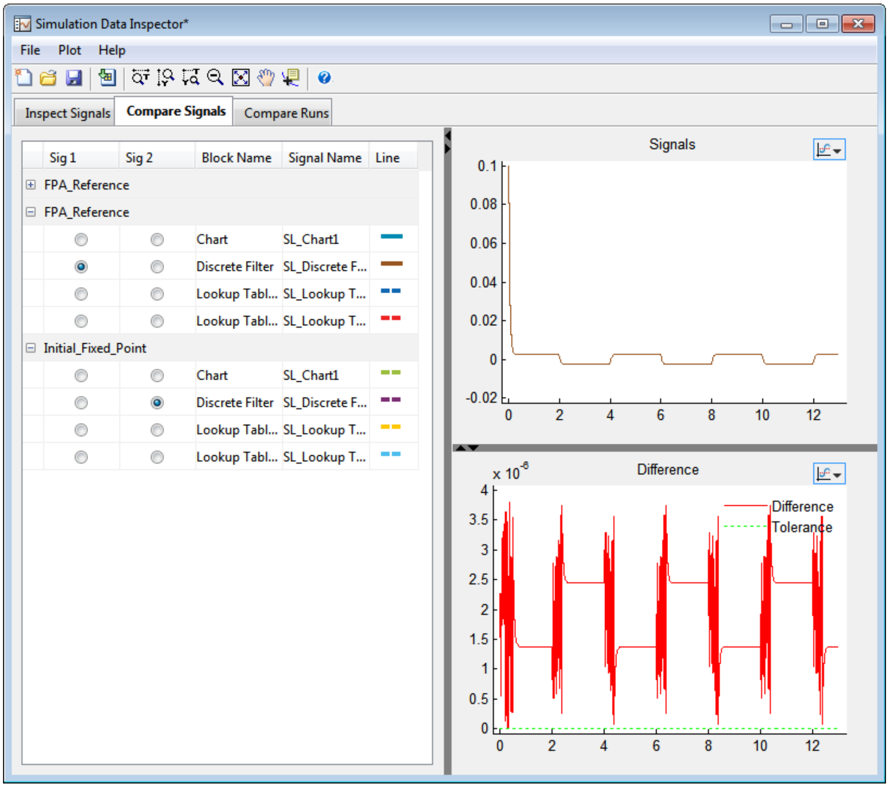
- Compare signals
- Inspect signals
- Compare runs

Compare Signals

To compare signals:

1. In one of the runs that you want to compare, select a logged signal.
2. From the Fixed-Point Tool menu, select **Results > Compare Signals** or click .
3. If there are more than two runs, in the **Compare Runs Selector** dialog box, select the run that you want to compare, and then click **OK**.

Plots the signal in both selected runs on the upper axes, and plots the difference between the selected runs on the lower axes. For more details, see “Propose Fraction Lengths Using Simulation Range Data”.


Simulation Data Inspector*
File
Plot
Help
Inspect Signals
Compare Signals
Compare Runs
Signals
0.1
0.08
0.06
0.04
0.02
0
-0.02
0
2
4
6
8
10
12
x 10^-6
Difference
4
3.5
3
2.5
2
1.5
1
0.5
0
0
2
4
6
8
10
12
Difference
Tolerance

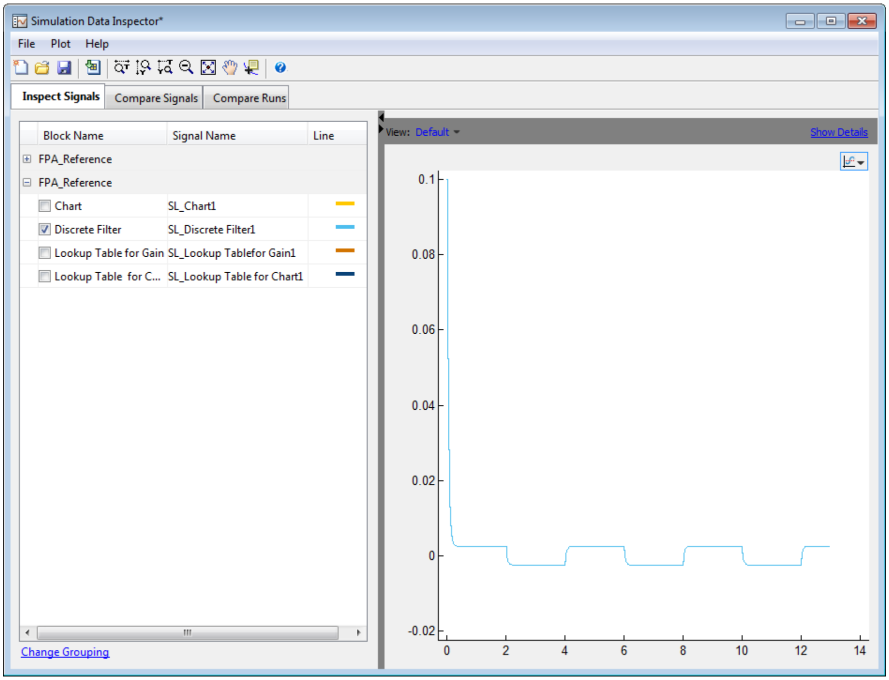
Sig 1	Sig 2	Block Name	Signal Name	Line
⊞ FPA_Reference				
⊟ FPA_Reference				
○	○	Chart	SL_Chart1	━
●	○	Discrete Filter	SL_Discrete F...	━
○	○	Lookup Tabl...	SL_Lookup T...	╍
○	○	Lookup Tabl...	SL_Lookup T...	╍
⊟ Initial_Fixed_Point				
○	○	Chart	SL_Chart1	╍
○	●	Discrete Filter	SL_Discrete F...	╍
○	○	Lookup Tabl...	SL_Lookup T...	╍
○	○	Lookup Tabl...	SL_Lookup T...	╍

Inspect Signal

To inspect a signal:

1. Select the logged signal that you want to inspect.
2. From the Fixed-Point Tool menu, select **Results > Inspect Signal** or click .

Plots data as a function of time.


Simulation Data Inspector*
File
Plot
Help
Inspect Signals
Compare Signals
Compare Runs
View: Default
Show Details
Change Grouping
0.1
0.08
0.06
0.04
0.02
0
-0.02
0
2
4
6
8
10
12
14

Block Name	Signal Name	Line
⊞ FPA_Reference		
⊟ FPA_Reference		
☐ Chart	SL_Chart1	🟨
☑ Discrete Filter	SL_Discrete Filter1	🟦
☐ Lookup Table for Gain	SL_Lookup Tablefor Gain1	🟧
☐ Lookup Table for C...	SL_Lookup Table for Chart1	🟦

Note that, unlike the Fixed-Point Tool, the Simulation Data Inspector allows multiple runs with the same name.

Compare Runs

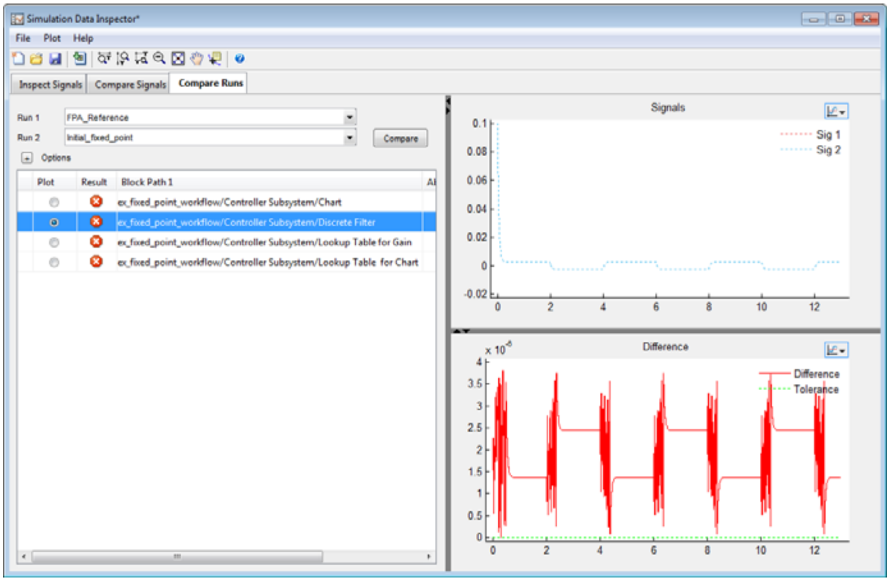
To compare runs:

1. In one of the runs that you want to compare, select a logged signal.

2. From the Fixed-Point Tool menu, select **Results > Compare Runs** or click .

3. If there are more than two runs, in the **Compare Runs Selector** dialog box, select the run that you want to compare, and then click **OK**.

Plots the signal in both selected runs on the upper axes, and plots the difference between the selected runs on the lower axes.


Simulation Data Inspector*
File
Plot
Help
Inspect Signals
Compare Signals
Compare Runs
Compare
Options
Signals
Sig 1
Sig 2
0.1
0.08
0.06
0.04
0.02
0
-0.02
0
2
4
6
8
10
12
x 10⁻⁶
Difference
Difference
Tolerance
4
3.5
3
2.5
2
1.5
1
0.5
0

Run 1: FPA_Reference

Run 2: Initial_fixed_point

Plot	Result	Block Path 1	Al
○	☒	ex_fixed_point_workflow/Controller Subsystem/Chart	
●	☒	ex_fixed_point_workflow/Controller Subsystem/Discrete Filter	
○	☒	ex_fixed_point_workflow/Controller Subsystem/Lookup Table for Gain	
○	☒	ex_fixed_point_workflow/Controller Subsystem/Lookup Table for Chart	

Histogram Plot

To view the histogram plot of a signal:

1. Select the logged signal that you want to plot.

2. From the Fixed-Point Tool menu, select **Results > Histogram Plot of Signal** or click 📊.

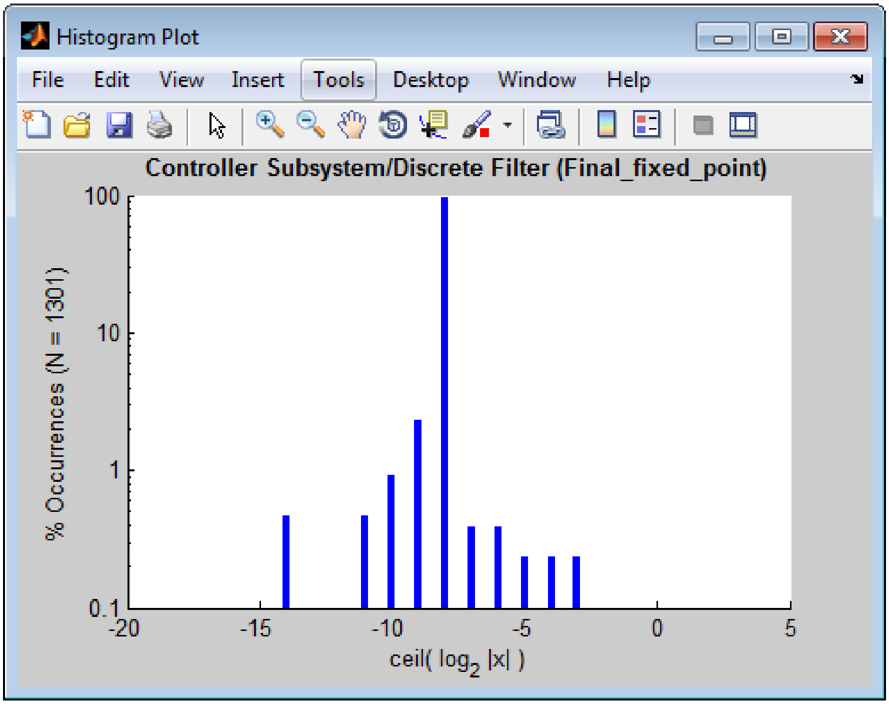
The histogram plot helps you visualize the dynamic range of a signal.

This plot provides information about the:

- Total number of samples (N)
- The maximum number of bits needed to prevent overflow

- Number of times each bit has been used to represent the data (as a percentage of the total number of samples)
- Number of times that exact zero occurred (without the effect of quantization). This does not include the number of zeroes that occurred due to rounding.

You can use this information to estimate the word size needed to represent the signal.


Histogram Plot
File Edit View Insert Tools Desktop Window Help
Controller Subsystem/Discrete Filter (Final_fixed_point)
% Occurrences (N = 1301)
100
10
1
0.1
-20
-15
-10
-5
0
5
ceil(log2 |x|)

Signal Logging Options

The Fixed-Point Tool provides options that allow you to control signal logging in a model (see "Signal Logging" in*Simulink User's Guide*). Using these options, you can enable or disable logging for multiple signals simultaneously, based on signal attributes such as:

- The location of signals in a model hierarchy

- Whether or not signals have names

The Fixed-Point Tool does not control signal logging for referenced models or subsystems with library links. It ignores these objects when enabling or disabling signal logging in a model hierarchy.

You can control the signal logging of a Simulink subsystem, that is placed inside of a Stateflow Chart, from the subsystem parent node.

How to Access the Signal Logging Options in the Fixed-Point Tool

1. In the **Model Hierarchy** pane, right-click a node that represents either a model or a subsystem.

   The Fixed-Point Tool displays a context menu for the selected node.

2. In the context menu, select either **Enable Signal Logging** or **Disable Signal Logging**.

   The Fixed-Point Tool displays a submenu that lists a variety of signal logging options.

Signal Logging Options

Choose from the following signal logging options:

Select...	To Enable or Disable Signal Logging for...
All Signals in this System	All signals in the selected system
All Signals from Here Down	All signals in the selected system and its subsystems
Outports in this System	All Outport blocks in the selected system
Named Signals in this System	All signals in the selected system, whose **Signal name** parameter specifies a value

Select...	To Enable or Disable Signal Logging for...
Named Signals from Here Down	All signals in the selected system and its subsystems, whose **Signal name** parameter specifies a value
Unnamed Signals in this System	All signals in the selected system, whose **Signal name** parameter is empty
Unnamed Signals from Here Down	All signals in the selected system and its subsystems, whose **Signal name** parameter is empty

Examples

To learn how to use the tool, see "Propose Fraction Lengths Using Simulation Range Data".

See Also

`autofixexp`

Purpose

Get pathname of current block

Syntax

```
gcb
gcb('sys')
```

Description

`gcb` returns the full block pathname of the current block in the current system.

`gcb('sys')` returns the full block pathname of the current block in the specified system.

The current block is one of these:

- During editing, the current block is the block most recently clicked.
- During simulation of a system that contains S-Function blocks, the current block is the S-Function block currently executing its corresponding MATLAB function.
- During callbacks, the current block is the block whose callback routine is being executed.
- During evaluation of the `MaskInitialization` string, the current block is the block whose mask is being evaluated.

Examples

This command returns the path of the most recently selected block.

```
gcb
ans =
   clutch/Locked/Inertia
```

This command gets the value of the `Gain` parameter of the current block.

```
get_param(gcb,'Gain')
ans =
   1/(Iv+Ie)
```

See Also

`gcbh` | `gcs`

gcbh

Purpose 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`

Purpose

Get pathname of current system

Syntax

`gcs`

Description

`gcs` returns the full pathname of the current system.

The current system is one of these:

- During editing, the current system is the system or subsystem most recently clicked.
- During simulation of a system that contains S-Function blocks, the current system is the system or subsystem containing the S-Function block that is currently being evaluated.
- During callbacks, the current system is the system containing any block whose callback routine is being executed.
- During evaluation of the `MaskInitialization` string, the current system is the system containing the block whose mask is being evaluated.

The current system is always the current model or a subsystem of the current model. Use `bdroot` to get the current model.

Examples

This example returns the path of the system that contains the most recently selected block.

```
gcs
ans =
   clutch/Locked
```

See Also

`bdroot` | `gcb`

get_param

Purpose Get system and block parameter values

Syntax

```
ParameterValue = get_param(Object, ParameterName)
ParameterValues = get_param(ObjectCellArray, ParameterName)
ParameterValue = get_param(ObjectHandle, ParameterName)
ParameterValue = get_param(0, ParameterName)

ParameterStruct = get_param(Object,'ObjectParameters')
ParameterCellArray = get_param(Object, 'DialogParameters')
```

Description

`ParameterValue = get_param(Object, ParameterName)` returns the value of the specified parameter. Case is ignored for parameter names. Value strings are case-sensitive.

`ParameterValues = get_param(ObjectCellArray, ParameterName)` accepts a cell array with full path specifiers, and returns the values of a parameter common to all objects specified in the cell array.

`ParameterValue = get_param(ObjectHandle, ParameterName)` returns the value of the specified parameter for the object whose handle is `ObjectHandle`.

`ParameterValue = get_param(0, ParameterName)` returns the current value of a Simulink session parameter or the default value of a model or block parameter.

`ParameterStruct = get_param(Object,'ObjectParameters')` returns a structure that describes the`Object` parameters.

`ParameterCellArray = get_param(Object, 'DialogParameters')` returns a cell array containing the names of the `Object` dialog parameters.

Input Arguments

Object - Model name, or path and name of block

Model name, or path and name of block.

ObjectCellArray - Cell array of full path specifiers for objects

Cell array with full path specifiers for objects from which `get_param` returns the values of a specified parameter.

ObjectHandle - Handle to a model object

Handle to a model object from which `get_param` returns a specified parameter value.

ParameterName - Name of a parameter

Name of a parameter for which `get_param` returns a value.

Output Arguments

ParameterValue - Value of a parameter

Value of a parameter specified by `ParameterName`.

ParameterValues - Values of a parameter

Values of a parameter specified by `ParameterName` in each object specified in `ObjectCellArray`.

ParameterStruct - Structure with parameter information

Structure with parameter information for `Object`. Each field of the structure corresponds to the parameter with the same name as the field. Each field contains three subordinate fields: `Type`, `Enum` (if applicable), and `Attributes`. These fields respectively specify the parameter's data type, enumerated values (if applicable) , and attributes.

ParameterCellArray - Cell array of dialog parameter names

Cell array of dialog parameter names for the dialog parameters of `Object`.

Examples

Get a Block Parameter Value and Attributes

List the block parameter names for the `Inertia` block in the `Requisite Friction` subsystem of the `sldemo_clutch` model.

```
block_parameters = get_param('sldemo_clutch/Friction Mode Logic/Requisite
```

```
ans =
                                          Gain: [1x1 struct]
                             Multiplication: [1x1 struct]
                                 ParamMin: [1x1 struct]
                                ParamMax: [1x1 struct]
                  ParamDataTypeStr: [1x1 struct]
                                     OutMin: [1x1 struct]
                                    OutMax: [1x1 struct]
                      OutDataTypeStr: [1x1 struct]
                                LockScale: [1x1 struct]
                                  RndMeth: [1x1 struct]
    SaturateOnIntegerOverflow: [1x1 struct]
                             SampleTime: [1x1 struct]
```

Return the value of the `Gain` parameter.

```
block_parameter_value = get_param('sldemo_clutch/Friction Mode Logic/Requ
```

```
ans =
          1/(Iv+Ie)
```

Return attributes for the `Gain` parameter.

```
block_parameter_attributes = block_parameters.Gain.Attributes
```

```
ans =
           'read-write'    'was-link-instance'
```

Display Block Types for all Blocks in a Model

Get list of blocks.

```
blks = find_system(gcs, 'Type', 'block')

blks =
    'sldemo_clutch/Friction Mode Logic/Requisite Friction'
    'sldemo_clutch/Friction Mode Logic/Requisite Friction/Tin'
    [1x67 char]
    [1x66 char]
    'sldemo_clutch/Friction Mode Logic/Requisite Friction/Sum'
    'sldemo_clutch/Friction Mode Logic/Requisite Friction/Sum1'
    [1x68 char]
    'sldemo_clutch/Friction Mode Logic/Requisite Friction/w1'
    'sldemo_clutch/Friction Mode Logic/Requisite Friction/Tf'
```

Get list of block types.

```
listblks = get_param(blks, 'BlockType')

listblks =
    'SubSystem'
    'Inport'
    'Gain'
    'Gain'
    'Sum'
    'Sum'
    'Gain'
    'From'
    'Outport'
```

Get Name of Block

Return the name of the currently selected block

```
get_param(gcb, 'Name')
```

See Also

`find_system` | `gcb` | `set_param`

Related Examples

- “Associating User Data with Blocks”
- “Model Parameters” on page 6-2
- “Block-Specific Parameters” on page 6-109
- “Common Block Properties” on page 6-96
- “Use MATLAB Commands to Change Workspace Data”

Purpose

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`

How To

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

Purpose Get information about annotation

Syntax `getCallbackAnnotation`

Description `getCallbackAnnotation` is intended to be invoked by annotation callback functions. If it is invoked from an annotation callback function, it returns an instance of `Simulink.Annotation` class that represents the annotation associated with the callback function. The callback function can then use the instance to get and set the annotation's properties, such as its text, font and color. If this function is not invoked from an annotation callback function, it returns nothing, i.e., `[]`.

Purpose Get one of model's configuration sets or configuration references

Syntax `myConfigObj = getConfigSet('model', 'configObjName')`

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObjName`

The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description `getConfigSet` returns the configuration set or configuration reference (configuration object) that is attached to *`model`* and is named *`configObjName`*. If no such object exists, an error occurs.

Examples The following example returns the configuration object that is named `DevConfig` and attached to the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
myConfigObj = getConfigSet(gcs, 'DevConfig');
```

See Also `attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

How To

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

getConfigSets

Purpose 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 strings 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`

How To

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

Purpose

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

`gcb` | `find_system`

getInputString

Purpose

Create comma-separated list of variables to map

Syntax

```
externalInputString = getInputString(inputmap,'base')

externalInputString = getInputString(inputmap,filename)
```

Description

`externalInputString = getInputString(inputmap,'base')` creates an input string using the supplied mapping `inputmap` and the variables loaded in the base workspace (`'base'`).

This function generates a comma-separated list of variables (input string) 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 string into the text field.

This function is most useful if you have created a custom mapping.

`externalInputString = getInputString(inputmap,filename)` creates an input string using the supplied mapping `inputmap` and the variables defined in `filename`.

Input Arguments

inputmap - Map object

string

Map object, as returned from the `getRootInportMap` function.

filename - Input variables

MAT-file name as string

Input variables, contained in a MAT-file. The file contains variables to map.

Example: `'data.mat'`

Data Types
`char`

Output Arguments

externalInputString - External input string

Comma-separated string

External input string, returned as a comma-separated string. The string contains root inport information that you can specify to the `sim` command or the **Configuration Parameters > Data Import/Export > Input** parameter.

Examples

Create an input string from the base workspace

Create an input string 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 string.

```
sim('slexAutotransRootInportsExample','ExternalInput',...
externalInputString);
```

Create an external input string from variables in a MAT-file

Create an external input string 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 string.

```
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);
```

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.

See Also

`getrootinportmap`

Concepts

- “Import and Map Data to Root-Level Inports”

getRootInportMap

Purpose

Creates 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.

`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.

Use the `getRootInportMap` function when creating a custom mapping mode to map data to root-level inports. See `BlockNameIgnorePrefix.m` for an example of a custom mapping algorithm.

Tips

Load the model before running this function.

Input Arguments

Empty

Create an empty map object.

Default: none

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'model'

Name of model to associate with the root inport map.

Default: None

'blockName'

Block names of root-level input ports. The tool assigns data to ports according to the name of the root-inport block. If the tool finds a data element whose name matches the name of a root-inport block, it maps the data to the corresponding port.

The value for this argument can be:

Block name of root-level input ports.
Cell array containing multiple block names of root-level input ports.

Default: None

'signalName'

Signal names to be mapped. The tool assigns data to ports according to the name of the signal on the port. If the tool finds a data element whose name matches the name of a signal at a port, it maps the data to the corresponding port.

The value for this argument can be:

Signal name to be mapped.
Cell array containing multiple signal names of signals to be mapped.

Default: None

'inputmap'

Name of mapping object to override.

Default: None

Output Arguments

map

Custom object that you can use to map data to root-level input port.

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, `matlabroot\help\toolbox\simulink\examples\ex_minportsOnlyModel`.

```
load_system ex_minportsOnlyModel;
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 `matlabroot\help\toolbox\simulink\examples\ex_minportsOnlyModel`.

Create a mapping object of vectors.

```
load_system ex_minportsOnlyModel;
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
load_system ex_minportsOnlyModel;
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]
```

See Also

`getInputString`

Related Examples

- "Create Custom Mapping File Function"

Purpose

Get value of variable from workspace

Syntax

```
variableValue = getVariable(workspaceHandle,variableName)
variableValue = workspaceHandle.getVariable(variableName)
```

Description

`variableValue = getVariable(workspaceHandle,variableName)` returns the value of the variable. If the variable does not exist in the workspace, an error occurs.

`variableValue = workspaceHandle.getVariable(variableName)` is an alternative syntax.

Input Arguments

workspaceHandle

Handle to the workspace containing the variable.

variableName

Name of the variable containing the value.

Output Arguments

variableValue

Value of the variable.

Examples

Get the value of the workspace variable `K`, which is defined in model `mdl`.

```
wksp = get_param(mdl,'ModelWorkspace')
value = wksp.getVariable('K')
```

```
value =
   5
```

See Also

`get_param`

hasVariable

Purpose

Determine if variable exists in workspace

Syntax

```
variableExists = hasVariable(workspaceHandle,variableName)
variableExists = workspaceHandle.hasVariable(variableName)
```

Description

`variableExists = hasVariable(workspaceHandle,variableName)` returns `1` if the variable exists in the workspace, and `0` if not.

`variableExists = workspaceHandle.hasVariable(variableName)` is an alternative syntax.

Input Arguments

workspaceHandle

Handle to the workspace.

variableName

Name of the variable.

Output Arguments

variableExists

Boolean value that indicates whether the variable exists in the workspace (`1` if true and `0` if false).

Examples

Determine if the variable `K` exists in the workspace for model `mdl`.

```
wksp = get_param(mdl,`ModelWorkspace')
exists = wksp.hasVariable('K')

exists =
   1
```

See Also

`get_param`

Purpose

Highlight Simulink object

Syntax

`hilite_system(`*`block_path`*`)`
`hilite_system(`*`block_path`*`, `*`hilite_scheme`*`)`

Description

`hilite_system(`*`block_path`*`)` highlights a model object using colors specified by the default highlighting scheme. `hilite_system(`*`block_path`*`, `*`hilite_scheme`*`)` highlights a model object using the foreground and background colors specified in the highlighting scheme.

Tips

- Calling `hilite_system` does not clear highlighted objects from previous `hilite_system` calls.
- Using a traceability tag for *`block_path`* requires a Simulink Coder license. If you call `hilite_system` with a traceability tag as input, do the following:
  - If you closed and reopened a model, you must update the model before calling `hilite_system`.
  - If you changed your model, such as adding a block to your diagram, before calling `hilite_system`, generate new code for the model. When the system hierarchy of the model changes, traceability tags change. If you use a traceability tag from previously generated code, `hilite_system` might highlight the wrong block.
  - If a block name contains a newline character (`\n`), it is replaced with a space for readability. When calling `hilite_system`, in the block path string, replace the newline character with a space.
- `hilite_system` might not work for a block, if the block name contains:
  - A single quote (`'`).
  - An asterisk (`*`), that causes name ambiguity relative to other names in the model. This name ambiguity occurs in a block name or at the end of a block name if an asterisk precedes or follows a slash (`/`).

  - The character (`char(255)`).

Input Arguments

block_path

A string in two possible formats:

- A full block path
- A traceability tag from the comments of Simulink Coder generated code. Using a traceability tag requires a Simulink Coder license. In this case, the format is `<`*`system`*`>/`*`block_name`*:
  - *`system`* is one of the following:
    - The string `Root`.
    - A unique system number assigned by the Simulink engine.
  - *`block_name`* is the name of the source block. If a block name contains a newline character (`\n`), in the block path string, replace the newline character with a space.

hilite_scheme

String identifying a highlighting scheme name. For more information, see “Highlighting Scheme” on page 2-268.

Definitions

Highlighting Scheme

A highlighting scheme specifies the foreground and background colors for a model object. Possible highlighting scheme names are:

`default`
`none` (clears the highlighting for an object)
`find`
`unique`
`different`
`user1`
`user2`
`user3`
`user4`

```
user5
```

You can alter the specification for a highlighting scheme by using the following command:

```
set_param(0, 'HiliteAncestorsData', HILITE_DATA)
```

`HILITE_DATA` is a MATLAB structure array with the following fields:

- `'HiliteType'`: string specifying a highlighting scheme.
- `'ForegroundColor'`: string specifying a foreground “Color” on page 2-269.
- `'BackgroundColor'`: string specifying a background “Color” on page 2-269.

Color

The supported color strings for foreground and background colors are:

```
black
white
gray
red
orange
yellow
green
darkGreen
blue
lightBlue
cyan
magenta
```

Examples

1. Open a Simulink model. For example, in the MATLAB Command Window, type

   ```
   slexAircraftExample
   ```

2 Use the block path to highlight the `Controller` block.

```
hilite_system('slexAircraftExample/Controller')
```

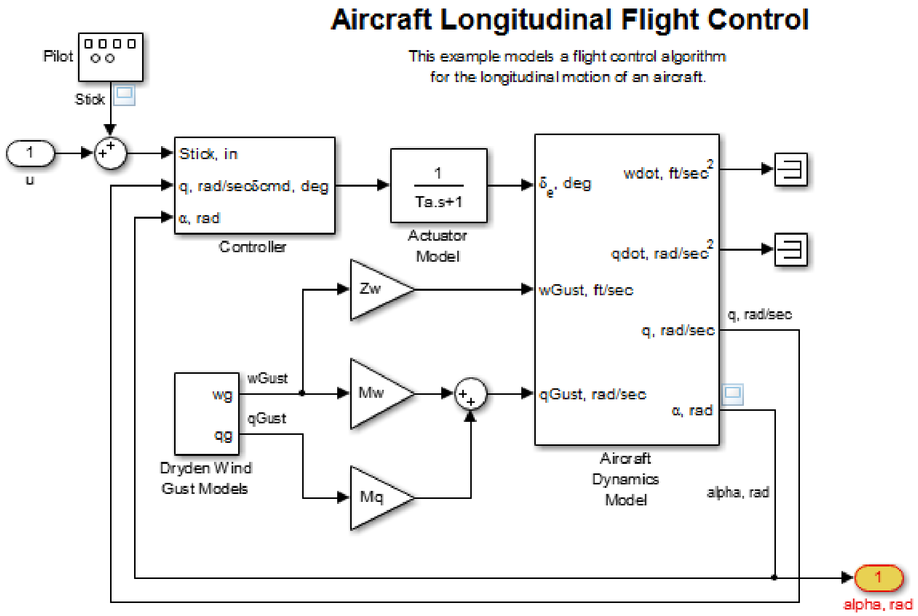
In the model diagram, the `Controller` block is highlighted yellow.

If you have a Simulink Coder license, you can trace generated code to the corresponding source system or block in a model.

1 Open the Model Configuration Parameters dialog box. In the **Solver** pane, set solver parameters as follows:
   - Set the solver **Type** to `Fixed-step`.
   - Set **Fixed step size** to `0.1`.
2 Generate code for the model. **Code > C/C++ Code > Build Model**.
3 In an editor or within an HTML code generation report, open a generated source or header file.
4 As you review lines of code, note traceability tags that correspond to code of interest. To highlight a block using a traceability tag, enter:

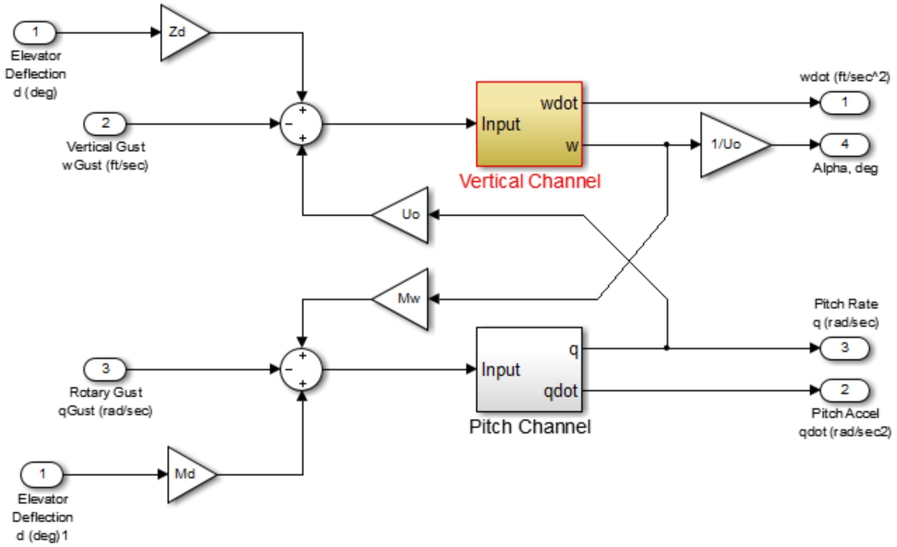
```
hilite_system('<Root>/alpha, rad')
```

The following figure shows block `alpha, rad` highlighted.


Aircraft Longitudinal Flight Control
This example models a flight control algorithm
for the longitudinal motion of an aircraft.
Pilot
Stick
1
u
Stick, in
q, rad/sec
δcmd, deg
α, rad
Controller
1
Ta.s+1
Actuator
Model
δe, deg
wdot, ft/sec2
qdot, rad/sec2
wGust, ft/sec
q, rad/sec
qGust, rad/sec
α, rad
Aircraft
Dynamics
Model
q, rad/sec
alpha, rad
Zw
Mw
Mq
wGust
qGust
wg
qg
Dryden Wind
Gust Models
1
alpha, rad

You can also use the `hilite_system` command to highlight a block within a subsystem. Specify the `Aircraft Dynamics Model` subsystem using its traceability tag.

```
hilite_system('<S1>/Vertical Channel')
```


1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
+
-
+
Input
wdot
w
Vertical Channel
wdot (ft/sec^2)
1
1/Uo
4
Alpha, deg
Uo
Mw
Pitch Rate q (rad/sec)
3
+
-
+
3
Rotary Gust qGust (rad/sec)
Input
q
qdot
Pitch Channel
2
Pitch Accel qdot (rad/sec2)
1
Elevator Deflection d (deg) 1
Md

See Also

Purpose

Returns intersection of two vectors of `Simulink.VariableUsage` objects

Syntax

```
VarsOut = intersect (VarsIn1, VarsIn2)
VarsOut = VarsIn1.intersect (VarsIn2)
```

Description

`VarsOut = intersect (VarsIn1, VarsIn2)` inputs two vectors of `Simulink.VariableUsage` objects and returns a vector containing a `Simulink.VariableUsage` object for each variable that is represented in both input vectors. The function compares input objects by comparing their `Name`, `Workspace`, and `WorkspaceType` values. If all three value are the same in both objects, the two objects match. Any returned object is a new object having the same field values as the matching input objects, except that its `UsedByBlocks` field contains the union of the values in the `UsedByBlocks` fields of the two matching objects.

`VarsOut = VarsIn1.intersect (VarsIn2)` is the equivalent method syntax (`Simulink.VariableUsage.intersect`) and gives identical results.

Input Arguments

VarsIn1

A vector of `Simulink.VariableUsage` objects

VarsIn2

Another vector of `Simulink.VariableUsage` objects

Output Arguments

VarsOut

A vector of `Simulink.VariableUsage` objects containing a copy of any object in *`VarsIn1`* that matches an object in *`VarsIn2`*. Two objects match if their `Name`, `Workspace`, and `WorkspaceType` values are the same. The `UsedByBlocks` field of each returned object is the union of the values of that field in the matching input objects.

Examples

Given two models, discover which variables are needed by the first model, the second model, or both models:

```
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
commonVars = intersect(model1Vars, model2Vars);
```

See Also

`Simulink.findVars` | `Simulink.VariableUsage` | `setdiff`

Purpose

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.

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`

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
```

See Also

`simulinkprojectreload`

legacy_code

Purpose 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('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 any compiler options that you might specify. The following examples show how to specify no options, one option, and multiple options:

```
legacy_code('compile', s);
legacy_code('compile', s, '-DCOMPILE_VALUE1=1');
legacy_code('compile', s,...
       {'-DCOMPILE_VALUE1=1', '-DCOMPILE_VALUE2=2',...
 '-DCOMPILE_VALUE3=3'});
```

`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 “Integrate External Code Using Legacy Code Tool”.

`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” and “Integrate External Code Using Legacy Code Tool” in the Simulink Coder documentation.

`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 string specifying a name for the S-function to be generated by the Legacy Code Tool.

Define Legacy Code Tool Function Specifications

- `InitializeConditionsFcnSpec` — A nonempty string 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 string 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 string 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 string 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 strings specifying the file names of header files required for compilation.
- `SourceFiles` — A cell array of strings specifying source files required for compilation. You can specify the source files using absolute or relative path names.
- `HostLibFiles` — A cell array of strings specifying library files required for host compilation. You can specify the library files using absolute or relative path names.
- `TargetLibFiles` — A cell array of strings 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 strings specifying directories containing header files. You can specify the directories using absolute or relative path names.
- `SrcPaths` — A cell array of strings specifying directories containing source files. You can specify the directories using absolute or relative path names.
- `LibPaths` — A cell array of strings 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 string 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`).

`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 string `'slblock_generate'`. If you omit this argument, the block appears in an empty model editor window.

How To

- "Integrate C Functions Using Legacy Code Tool"
- "Integrate External Code Using Legacy Code Tool"

Purpose

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`

How To

- "About Block Libraries and Linked Blocks"

Purpose

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 string 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

`linmod` | `dlinmod` | `linmod2` | `linmodv5`

linmod2

Purpose

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 string 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

`linmod` | `dlinmod` | `linmodv5`

linmodv5

Purpose

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 string 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` | `dlinmod` | `linmod2`

load_system

Purpose

Invisibly load Simulink model

Syntax

```
load_system('sys')
```

Description

`load_system('sys')` loads `sys`, where *`sys`* is the name of a Simulink model, into memory without making its model window visible.

You cannot use `load_system` to load MATLAB file models last saved in Simulink Version 1.3 (for example: `load_system mymodel.m`). If you have a MATLAB file model, you must upgrade to Simulink model file format as follows:

1. Execute the model as a function:

   ```
   mymodel
   ```

2. Save the model as a Simulink model file:

   ```
   save_system mymodel
   ```

Examples

The command

```
load_system('vdp')
```

loads the `vdp` sample model into memory.

See Also

`close_system` | `open_system`

Purpose

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 (`flag`) argument specifies the phase of the simulation to be executed. See "Simulating Dynamic Systems" for a description of the steps that Simulink software uses to simulate a model.

This command is intended to allow linear analysis and other MATLAB program-based tools to run a simulation step-by-step, gathering information about the states and outputs of a model, at each step. It is 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	Discrete states of the model at time `t`.
`t`	Time step
`x`	State vector
`u`	Inputs
`flag`	String that indicates 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.

Examples

This 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 You must always terminate simulation of the model by invoking the model command with the `'term'` command. Simulink software does not let you close the model until you have terminated the simulation.

See Also

`sim`

modeladvisor

Purpose

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.

Input Arguments

model - Model or subsystem name

string

Model or subsystem name or handle, specified as a string.

Data Types

`char`

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)
```

See Also “Consult the Model Advisor”

new_system

Purpose

Create empty Simulink system

Syntax

```
new_system(sys)
new_system(sys, model)
new_system(sys, model, subsystem_path)
new_system(sys, model, 'ErrorIfShadowed')
new_system(sys, 'Library')
h = new_system(sys)
```

Description

`new_system(sys)` or `new_system(sys, model)` creates an empty system where *`sys`* is the name of the new system. This command displays an error if *`sys`* is a MATLAB keyword, `'simulink'`, or more than 63 characters long.

`new_system(sys, model, subsystem_path)` creates a system from a subsystem where *`subsystem_path`* is the full path of the subsystem. The model that contains the subsystem must be open when this command is executed.

`new_system(sys, model, 'ErrorIfShadowed')` creates an empty system having the specified name. This command generates an error if another model, MATLAB file, or variable of the same name exists on the MATLAB path or workspace.

`new_system(sys, 'Library')` creates an empty library.

`h = new_system(sys)` returns the numeric handle of the system that has been created. You can pass `h` to any of the Simulink API functions, for example, `open_system(h)`.

Note The `new_system` command does not open the window of the system or library that it creates.

See "Model Parameters" on page 6-2 and "Block-Specific Parameters" on page 6-109 for a list of the default parameter values for the new system.

Examples

This command creates a new system named `'mysys'`.

```
new_system('mysys')
```

The command

```
new_system('mysys','Library')
```

creates, but does not open, a new library named `'sys'`.

The command

```
new_system('vdp','Model','ErrorIfShadowed')
```

returns an error because `'vdp'` is the name of a model on the MATLAB path.

The commands

```
load_system('f14')
new_system('mycontroller','Model','f14/Controller')
```

create a new model named `mycontroller` that has the same contents as does the subsystem named `Controller` in the `f14` model.

The commands

```
 h = new_system('mymodel')
h =
     3.0012

>> get_param(h,'Name')
ans =
   mymodel

 open_system(h)
```

return the numeric handle of the system that has been created, and use that handle to get parameters and open the model.

See Also `close_system` | `open_system` | `save_system`

Purpose

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`

open_system

Purpose

Open Simulink system window or block dialog box

Syntax

```
open_system('sys')
open_system('blk')
open_system('blk', 'force')
open_system('blk', 'parameter')
open_system('blk', 'mask')
open_system('blk', 'OpenFcn')
open_system('sys', 'window')
open_system('sys', 'tab')
```

Description

`open_system('sys')` opens the specified system or subsystem in a Simulink Editor model window, where `sys` is the name of a model on the MATLAB path, the fully qualified path name of a model, or the relative path name of a subsystem of an already opened system (for example, `engine/Combustion`). On UNIX systems, the fully qualified path name of a model can start with a tilde (~), signifying your home directory.

Note Use `load_system` to load a model before you use `open_system` to open a subsystem in that model. If you use the `open_system` command to open a subsystem without first opening or loading the model that contains the subsystem, Simulink gives an error.

`open_system('blk')`, where `'blk'` is a full block path name, opens the dialog box associated with the specified block. If the block `OpenFcn` callback parameter is defined, the routine is evaluated.

`open_system('blk', 'force')`, where `blk` is a full block path name or a masked system (looks under the mask of the specified system). The system opens in a new Simulink Editor tab. For a masked system, this command is equivalent to using the **Look Under Mask** menu item.

`open_system('blk', 'parameter')` opens the block parameter dialog box, or for masked subsystems, the Mask Parameters dialog box.

`open_system('sys', 'mask')` opens the block mask.

`open_system('blk', 'OpenFcn')` runs the block open function.

`open_system('sys', 'window')` opens a new Simulink Editor instance to display the contents of the subsystem `sys` opened by this command.

`open_system('sys', 'tab')` opens a new Simulink Editor tab to display the contents of the subsystem `sys` opened by this command.

Note To insert carriage return or line feed characters into paths passed to the `open_system` command, use the MATLAB `sprintf` command . For example, the path to the `Aircraft Dynamics Model` subsystem of the `f14` model contains line feeds. To open the subsystem, enter the following command at the MATLAB command line:

```
open_system(['f14/Aircraft' sprintf('\n') 'Dynamics' sprintf('\n') 'Model'])
```

Examples

Open a Model

Open the `f14` model.

```
open_system('f14')
bdclose all
```

Open Subsystem

Open the `Controller` subsystem of the `f14` model.

```
load_system('f14')
open_system('f14/Controller')
bdclose all
```

Open Subsystem in Existing Tab

Open the `Aircraft Dynamics Model` subsystem in the same tab that the `Controller` subsystem had used.

```
load_system('f14')
```

```
open_system('f14')
open_system('f14/Controller')
open_system('f14/Aircraft Dynamics Model')
bdclose all
```

Open Subsystem in Separate Window

Open a subsystem in its own Simulink Editor window.

```
load_system('f14')
open_system('f14')
open_system('f14/Controller','window')
bdclose all
```

Open a Referenced Model

Open the model sldemo_mdlref_counter, which is referenced by the CounterA Model block in sldemo_mdlref_basic. Note that a separate Simulink Editor opens.

```
open_system('sldemo_mdlref_basic')
open_system('sldemo_mdlref_basic/CounterA')
bdclose all
```

Open Block Dialog Box

Open the block parameters dialog box for the first Gain block that is in the `Controller` subsystem.

```
load_system('f14')
open_system('f14/Controller/Gain')
bdclose_all
```

Run Block Open Callback Function

This example defines an `OpenFcn` callback for a block and executes that callback when you open that block.

```
load_system('f14')
set_param('f14/Controller/Gain','OpenFcn','open_system(''sf_car'')')
open_system('f14/Controller/Gain','OpenFnc')
bdclose_all
```

Open Masked Subsystem

This example shows a model that contains a masked subsystem, and a block that contains an `OpenFcn` callback that causes both the parameter dialog box for the block and the mask dialog box for the masked subsystem to appear.

```
load_system('sf_car')
set_param('sf_car/transmission','OpenFcn',['open_system(''sf_car/trans
'''parameter'');open_system(''sf_car/Vehicle'',''mask'')'])
open_system('sf_car')
open_system('sf_car/transmission')
bdclose all
```

Open Multiple Systems with One Command

This example shows how you can use a vector of systems to open multiple systems with one `open_system` command.

```
vector = {'f14', 'vdp'}
open_system(vector)
bdclose_all
```

See Also

`close_system` | `load_system` | `new_system` | `save_system`

Purpose

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`

How To

- “Manage a Configuration Set”
- “Manage a Configuration Reference”

Purpose

Open Performance Advisor

Syntax

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 string specifying the name or handle to the model or subsystem.

Examples

Open Performance Advisor

Open Performance Advisor on the `vdp` example model.

```
performancedvisor('vdp')
```

Performance Advisor opens the `vdp` model and opens Performance Advisor on the model.

Alternatives

Related Examples

- “Prepare a Model for Performance Advisor”
- “Run Performance Advisor Checks”

Concepts

- “Performance Advisor Window”
-

reload

Purpose

Reload Simulink Project

Syntax

`reload(proj)`

Description

`reload(proj)` reloads the project. Use `reload` when you want to run the project startup shortcuts.

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.

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)
```

See Also

`simulinkprojectisLoaded`

Purpose

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`.

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

string

Name of the category to remove, which exists in the project, specified as a string.

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.

See Also

`simulinkprojectfindCategorycreateCategory`

removeFile

Purpose Remove file from Simulink Project

Syntax `removeFile(proj,file)`

Description `removeFile(proj,file)` removes a file from the project `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.

file - Path of file
string | file object

Path of the file to remove relative to the project root folder, including the file extension, specified as a string or a file object returned by `findFile`. The file must be within the root folder.

Example: `'models/myModelName.slx'`

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')
```

See Also `simulinkprojectaddFilefindFile`

Purpose

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`.

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

string

Name of the label to remove, specified as a string.

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

string

Name of the category that contains the label to remove, specified as a string.

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`.

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: 'Review'
            DataType: 'char'
    LabelDefinitions: [1x4 slproject.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat,'review me');
```

Remove the new label.

```
removeLabel(cat,'review me');
```

Alternatively, remove the label from the project by specifying the category by name.

```
removeLabel(proj,'Review','review me');
```

See Also

`simulinkprojectfindLabelfindCategorycreateLabeladdLabel`

replace_block

Purpose

Replace blocks in Simulink model

Syntax

```
replace_block('sys', 'old_blk', 'new_blk')
replace_block('sys', 'parameter', 'value', ..., 'blk')
```

Description

`replace_block('`*`sys`*`', '`*`old_blk`*`', '`*`new_blk`*`')` replaces all blocks in *`sys`* having the block or mask type *`old_blk`* with *`new_blk`*.

- If *`new_blk`* is a Simulink built-in block, only the block name is necessary.
- If *`old_blk`* or *`new_blk`* is in another system, its full block pathname is required.
- If *`noprompt`* is omitted, Simulink software displays a dialog box that asks you to select matching blocks before making the replacement. Specifying the *`noprompt`* argument suppresses the dialog box from being displayed.
- If a return variable is specified, the paths of the replaced blocks are stored in that variable.

`replace_block('`*`sys`*`', '`*`parameter`*`', '`*`value`*`', ..., '`*`blk`*`')` replaces all blocks in *`sys`* having the specified values for the specified parameters with *`blk`*. You can specify any number of parameter name/value pairs. You can also specify `find_system` parameter/value pairs followed by any number of block parameter/value pairs. For example, to replace blocks inside links, specify `'FollowLinks', 'on'` to follow links into library blocks. For information on block parameters, see "Block-Specific Parameters" on page 6-109.

Note Because it may be difficult to undo the changes this command makes, it is a good idea to save your Simulink model first.

Examples

This command replaces all Gain blocks in the `f14` system with Integrator blocks and stores the paths of the replaced blocks in

`RepNames`. Simulink software lists the matching blocks in a dialog box before making the replacement.

```
RepNames = replace_block('f14','Gain','Integrator')
```

This command replaces all blocks in the `Unlocked` subsystem in the `sldemo_clutch` system having a `Gain` of `'bv'` with the Integrator block. Simulink software displays a dialog box listing the matching blocks before making the replacement.

```
replace_block('sldemo_clutch/Locked','Gain','bv','Integrator')
```

This command replaces the Gain blocks in the `f14` system with Integrator blocks but does not display the dialog box.

```
replace_block('f14','Gain','Integrator','noprompt')
```

This command replaces the `Lockup Detection` subsystem in the `sldemo_clutch` model with a Gain block.

```
replace_block('sldemo_clutch','Name','Lockup Detection','built-in/Gain')
```

This command from the mask initialization of a linked block replaces blocks inside itself:

```
replace_block(gcb, 'FollowLinks', 'on', 'BlockType', 'Gain', 'Integrator', 'noprompt')
```

See Also

`find_system` | `set_param`

save_system

Purpose Save Simulink system

Syntax

```
save_system
save_system(sys)
save_system(sys, newsysname)
save_system(sys, newsysname.slx)
save_system(sys, newsysname, Name,Value)
save_system(sys, 'exported_file_name.xml', 'ExportToXML',
   true)
filename = save_system(sys)
```

Description

`save_system` saves the current top-level system. If the system has not previously been saved, `save_system` creates a new file in the current folder.

`save_system(sys)` saves the top-level system that you specify in `sys` to a file using the current system name. `sys` must be a system name with no file extension. The system must be loaded. `sys` can be a string, a cell array of strings, a numeric handle, or an array of numeric handles. If you specify any options they apply to all the systems that you save.

`save_system(sys, newsysname)` saves the top-level system that you specify to a file using the new system name `newsysname`. The system must be loaded. `newsysname` can be a system name, or a filename with file extension and optional path, or empty. If you do not specify a file extension (`.slx` or `.mdl`) then `save_system` uses the file format specified in your Simulink preferences.

`save_system(sys, newsysname.slx)` saves the top-level system `sys` to a new file `newsysname` in the SLX file format.

`save_system(sys, newsysname, Name,Value)` saves the system with additional options specified by one or more `Name,Value` pair arguments.

`save_system(sys, '`*`exported_file_name`*`.xml', 'ExportToXML', true)` exports the system to a file in a simple XML format. Do not use `ExportToXML` with any other `save_system` options.

`filename = save_system(sys)` returns the fully-qualified file name of the file you saved.

`save_system` can save only entire systems. To save a subsystem, use the `Simulink.SubSystem.copyContentsToBlockDiagram` function to copy the subsystem contents to a new block diagram and then save it using `save_system`. See `Simulink.SubSystem.copyContentsToBlockDiagram`.

If you set the `UpdateHistory` property of the model to `UpdateHistoryWhenSave`, you see the following behavior:

- When you save interactively, you see a dialog prompting for a comment to include in the model history.
- When you save using `save_system`, you do not see a prompt for a comment. `save_system` reuses the previous comment, unless you set `'ModifiedComment'` before saving:

```
set_param(mymodel,'ModifiedComment',mycomment)
```

Input Arguments

sys

Top-level system to save. `'`*`sys`*`'` must be a system name, not a file name, i.e., without a file extension.

The system must be open. `'`*`sys`*`'` can be a string, a cell array of strings, a numeric handle, or an array of numeric handles.

newsysname

New system name.

`'`*`newsysname`*`'` can be a system name, or a filename with file extension and optional path. If you do not specify a file extension (`.slx` or `.mdl`) then `save_system` uses the file format specified in your Simulink preferences.

`'`*`newsysname`*`'` can be empty (`[]`), in which case the current name is used. You must specify a *`newsysname`* argument before any name-value pair arguments even when you don't want to name a new sys, and in that case you leave it empty.

If `'sys'` refers to more than one block diagram, `'newsysname'` must be a cell array of new names.

This command displays an error if you enter any of the following as the new system name:

- A MATLAB keyword
- `'simulink'`
- More than 63 characters

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `save_system(sys, newsysname, 'SaveModelWorkspace', true, 'BreakUserLinks', true, 'OverwriteIfChangedOnDisk', true)`

'BreakAllLinks'

Logical value that indicates whether the function replaces links to library blocks with copies of the library blocks in the saved file. The `'BreakAllLinks'` option affects any linked block, including user-defined and Simulink library blocks. Also accepts `on` or `off`.

`true`
`false` (default)

Note

The `'BreakAllLinks'` option can result in compatibility issues when upgrading to newer versions of Simulink software. For example:

- Any masks on top of library links to Simulink S-functions will not upgrade to the new version of the S-function.
- Any library links to masked subsystems in a Simulink library will not upgrade to the new subsystem behavior.
- Any broken links prevent the automatic library forwarding mechanism from upgrading the link.

If you have saved a model with broken links, use the `Check model, local libraries, and referenced models for known upgrade issues` option in the Model Advisor to scan the model for out-of-date blocks. You can then use the `slupdate` command to upgrade the Simulink blocks to their current versions. Subsequently running the Model Advisor lists any remaining third-party library and optional Simulink blockset blocks that are still out of date and need manual upgrading.

'BreakUserLinks'

Logical value that indicates whether the function replaces links to user-defined library blocks with copies of the library blocks in the saved file. Also accepts `on` or `off`.

`true`
`false` (default)

Default: false

'ErrorIfShadowed'

Logical value that indicates whether the function generates an error if the new name already exists on the MATLAB path or workspace. Also accepts `on` or `off`.

`true`
`false` (default)

'ExportToXML'

Logical value that indicates whether the function exports the specified block diagram to a file in a simple XML format. Specify the full name of the file, including an extension. The block diagram in memory does not change and no callbacks execute. Do not use this option with any other `save_system` options. Also accepts `on` or `off`.

`true`
`false` (default)

'ExportToVersion'

MATLAB release name, which specifies a previous Simulink version. `save_system` exports the system to a format that the specified previous Simulink version can load. You cannot export to your current version.

If the system contains functionality not supported by the specified Simulink software version, the command removes the functionality and replaces any unsupported blocks with empty masked subsystem blocks colored yellow. As a result, the converted system may generate different results.

To export to Release 2012a and later, you can specify model file format as SLX or MDL. If you do not specify a format, you export your default model file format.

`SaveAsVersion` is a legacy option for this argument that is also supported.

These version names are not case sensitive:

`'R14'`
`'R14SP1'`
`'R14SP2'`
`'R14SP3'`
`'R2006A'`

`'R2006B'`
`'R2007A'`
`'R2007B'`
`'R2008A'`
`'R2008B'`
`'R2009A'`
`'R2009B'`
`'R2010A'`
`'R2010B'`
`'R2011A'`
`'R2011B'`
`'R2012A_MDL'`
`'R2012A_SLX'`
`'R2012B_MDL'`
`'R2012B_SLX'`

'OverwriteIfChangedOnDisk'

Logical value that indicates whether to overwrite the file on disk (`true`) even if it has been externally modified since the system was loaded. To save the model regardless of whether the file has been changed on disk, supply the `OverwriteIfChangedOnDisk` option with value `true`.

If the file has changed on disk since the model was loaded, `save_system` displays an error to prevent the changes on disk from being overwritten, unless you use the `OverwriteIfChangedOnDisk` option set to `true`. Also accepts `on` or `off`.

`true`
`false` (default)

You can control whether `save_system` displays an error if the file has changed on disk by using the **Saving the model** option in the **Model File Change Notification** section of the Simulink Preferences dialog box. This preference is on by default.

'SaveModelWorkspace'

Logical value that indicates whether the function saves the contents of the model workspace. The model workspace `DataSource` must be a MAT-file. If the data source is not a MAT-file, `save_system` does not save the workspace. See “Specify Data Sources”. Also accepts `on` or `off`.

`true`
`false` (default)

Output Arguments

filename

`save_system` returns the full name of the file that you saved, as a string. If you saved multiple files, the return value is a cell array of strings.

Examples

The following examples assume prerequisites such as: you have loaded a model and the folder where you want to save is writeable.

Save the current system.

```
save_system
```

Save the `vdp` system with the name `vdp`.

```
save_system('vdp')
```

Save the `vdp` system to a file with the name `'myvdp'`. If you do not specify a file extension in the second argument (`.slx` or `.mdl`), then `save_system` uses the file format specified in your Simulink preferences.

```
save_system('vdp', 'myvdp')
```

Save the `vdp` system to another folder.

```
save_system('vdp', 'C:\TMP\vdp.slx')
```

Save an existing model `mymodel` to a different file and specifying the SLX file format:

```
save_system('mymodel', 'newsysname.slx')
```

Save the `vdp` system to a file with the name `'myvdp'` and replace links to library blocks with copies of the library blocks in the saved file.

```
save_system('vdp','myvdp','BreakAllLinks', true)
```

Save the current model (with its current name), and break any library links in it:

```
save_system('mymodel','mymodel','BreakAllLinks',true)
```

or

```
save_system('mymodel',[],'BreakAllLinks',true)
```

Save the current model with a new name, but display an error (instead of saving) if something with this name already exists on the MATLAB path:

```
save_system('mymodel','mynewmodel','ErrorIfShadowed',true)
```

Prevent saving the `vdp` system with a new name if something with this name already exists on the MATLAB path. In this case `save_system` displays an error (instead of saving) because `'max'` is the name of a MATLAB function.

```
save_system('vdp', 'max', 'ErrorIfShadowed', true)
```

Export the `vdp` system to Simulink Version R2008a with the name `'myvdp'`. It does not replace links to library blocks with copies of the library blocks.

```
save_system('vdp','myvdp','ExportToVersion','R2008a')
```

Save the current model with a new name, save the model workspace, break any library links, and overwrite if the file has changed on disk:

```
save_system('mymodel', 'mynewmodel', 'SaveModelWorkspace',
true, 'BreakAllLinks',true, 'OverwriteIfChangedOnDisk', true)
```

Return the full path name of the file that you saved, as a string. If you saved multiple files, the return value is a cell array of strings.

```
filename = save_system('mymodel')
```

Return the full path name of a system saved to a new file.

```
filename = save_system('mymodel', 'newmodelname')
```

See Also

`close_system` | `new_system` | `open_system`

How To

- "Save a Model"

Purpose

Set system and block parameter values

Syntax

`set_param(object,param,value)`

`set_param(object,param1,value1,...,paramN,valueN)`

Description

`set_param(object,param,value)` sets the parameter to the specified value on the specified model or block object.

`set_param(object,param1,value1,...,paramN,valueN)` sets parameters and values 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 arguments, rather than multiple `set_param` commands. This technique is efficient because using a single call requires evaluating parameters only once. It is more robust because it prevents dependency errors by setting all parameter values before checking the legality of the results.

Tip After you set parameters in the MATLAB workspace, in the Simulink editor, use **Simulation > Update Diagram** to see the changes in a model.

Input Arguments

object - Object name

string

Object name of a model or path and name of a block.

Example: `'vdp/Fcn'`

param - Parameter

string

Parameter name, specified as a string. Case is ignored for parameter names. Value strings are case sensitive.

Many block parameter values must be specified as strings.

Example: `'Position',[50 100 110 120]`

value - Parameter value
determined by parameter

Parameter value, specified in the format determined by the parameter. Values are often strings, but they can also be numeric, arrays, and so on. Value strings are case sensitive.

Many block parameter values are specified as strings. Two exceptions are the `Position`, specified as a vector, and `UserData`, specified according to the user data, parameters. These parameters are common to all blocks.

Example: `'Solver','ode15s','StopTime','3000'`

Example: `'SimulationCommand', 'start'`

Example: `'Position',[50 100 110 120]`

Examples

Set Model Configuration Parameters for the vdp System

Set the `Solver` and `StopTime` parameters of the `vdp` system. Open `vdp` first.

```
vdp
set_param('vdp','Solver','ode15s','StopTime','3000')
```

Set Model Configuration Parameters for Current System

Set the `Solver` and `StopTime` parameters of the current system. Open a model first.

```
vdp
set_param(gcs,'Solver','ode15s','StopTime','3000')
```

Set Position of Block

Set the position of the Fcn block in the `vdp` system. Open `vdp` first.

```
vdp
```

```
set_param('vdp/Fcn','Position',[50 100 110 120])
```

See Also

`get_param` | `gcs`

Related Examples

- "Associating User Data with Blocks"
- "Model Parameters" on page 6-2
- "Block-Specific Parameters" on page 6-109
- "Common Block Properties" on page 6-96
- "Use MATLAB Commands to Change Workspace Data"
- "Control Execution of a Simulation"

Purpose

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`

How To

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

Purpose

Returns difference between two vectors of `Simulink.VariableUsage` objects

Syntax

```
VarsOut  = setdiff (VarsIn1, VarsIn2)
VarsOut = VarsIn1.setdiff (VarsIn2)
```

Description

`VarsOut = setdiff (VarsIn1, VarsIn2)` inputs two vectors of `Simulink.VariableUsage` objects and returns a vector containing a `Simulink.VariableUsage` object for each workspace variable that is represented in the first vector but not the second. The function compares input objects by comparing their `Name`, `Workspace`, and `WorkspaceType` values. If all three value are the same in both objects, the two objects match. Any returned object is a new object having the same field values as the source input object.

`VarsOut = VarsIn1.setdiff (VarsIn2)` is the equivalent method syntax (`Simulink.VariableUsage.setdiff`) and gives identical results

Input Arguments

VarsIn1

A vector of `Simulink.VariableUsage` objects

VarsIn2

Another vector of `Simulink.VariableUsage` objects

Output Arguments

VarsOut

A vector of `Simulink.VariableUsage` objects containing a copy of any object in *`VarsIn1`* that does not match any object in *`VarsIn2`*. Two objects match if their `Name`, `Workspace`, and `WorkspaceType` values are the same.

Examples

Given two models, discover which variables are needed by the first model but not the second model:

```
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
```

```
commonVars = setdiff(model1Vars, model2Vars);
```

Locate all variables in the base workspace that are not being used by any recently-compiled open model:

```
models = find_system('type', 'block_diagram', 'LibraryType', 'None');
base_vars = Simulink.VariableUsage(who);
used_vars = Simulink.findVars(models, 'WorkspaceType', 'base');
unusedVars = setdiff(base_vars, used_vars);
```

See Also

`Simulink.findVars` | `Simulink.VariableUsage` | `intersect`

Purpose

Create MATLAB structure describing signed generalized fixed-point data type

Syntax

```
a = sfix(TotalBits)
```

Description

`sfix(`*`TotalBits`*`)` returns a MATLAB structure that describes the data type of a signed generalized fixed-point number with a word size given by *`TotalBits`*.

`sfix` is automatically called when a signed generalized fixed-point data type is specified in a block dialog box.

Note A default binary point is not included in this data type description. Instead, the scaling must be explicitly defined in the block dialog box.

Examples

Define a 16-bit signed generalized fixed-point data type:

```
a = sfix(16)
a =
       Class: 'FIX'
    IsSigned: 1
    MantBits: 16
```

See Also

`fixdt` | `float` | `sfrac` | `sint` | `ufix` | `ufrac` | `uint`

Purpose Create MATLAB structure describing signed fractional data type

Syntax

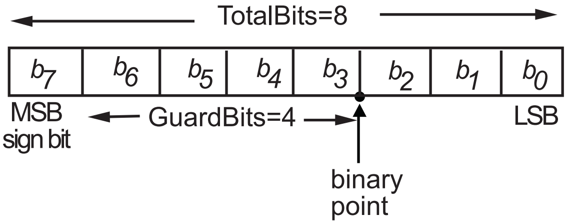
```
a = sfrac(TotalBits)
a = sfrac(TotalBits, GuardBits)
```

Description `sfrac(`*`TotalBits`*`)` returns a MATLAB structure that describes the data type of a signed fractional number with a word size given by *`TotalBits`*.

`sfrac(`*`TotalBits`*`, `*`GuardBits`*`)` returns a MATLAB structure that describes the data type of a signed fractional number. The total word size is given by *`TotalBits`* with `GuardBits` bits located to the left of the binary point.

`sfrac` is automatically called when a signed fractional data type is specified in a block dialog box.

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:



Examples Define an 8-bit signed fractional data type with 4 guard bits. Note that the range of this number is $-2^4 = -16$ to $(1 - 2^{(1 - 8)}).2^4 = 15.875$:

```
a = sfrac(8,4)
a =
        Class: 'FRAC'
     IsSigned: 1
     MantBits: 8
    GuardBits: 4
```

See Also

`fixdt` | `float` | `sfix` | `sint` | `ufix` | `ufrac` | `uint`

signalbuilder

Purpose

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(block, 'append', time,
data, signames,
   groupnames)
block = signalbuilder(block, 'appendgroup', time, data,
   signames, groupnames)
signalbuilder(block,'appendsignal', time, data, signames)
signalbuilder(block, 'showsignal', signal, group)
signalbuilder(block, 'hidesignal', signal, group)
[time, data] = signalbuilder(block, 'get', signal, group)
signalbuilder(block, 'set', signal, group, time, data)
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 2-343
- “Adding New Groups” on page 2-345

- "Working with Signals" on page 2-345
- "Using Get/Set Methods for Specific Signals and Groups" on page 2-346
- "Querying, Labelling, and Setting the Active Group" on page 2-347
- "Enabling Current Group Display" on page 2-347
- "Printing Signal Groups" on page 2-348

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 string or a cell array of strings.

`[time, data, signames, groupnames] = signalbuilder(`*`block`*`)` returns the group names, `groupnames`, in a string or a cell array of strings.

`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.

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.

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.

`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.

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]}`

Field	**Description**	**Example Value**
`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).

`figh = signalbuilder(`*`block`*`, 'print', `*`config`*`, 'figure')` prints the currently active signal group or the signal group that *`config`* specifies to a new hidden figure handle, `figh`.

Examples

Example 1

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 5], {[2 2];[0 2]});
```

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])
```

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)
```

Purpose

Simulate dynamic system

Syntax

```
simOut = sim('model', 'ParameterName1',Value1,
   'ParameterName2', Value2...);
simOut = sim('model', ParameterStruct);
simOut = sim('model', ConfigSet);
```

Description

`simOut = sim('model', 'ParameterName1',Value1,'ParameterName2', Value2...);` causes Simulink software to simulate the block diagram, *`model`*, using parameter name-value pairs `ParameterName1`, `Value1` and `ParameterName2`, `Value2`.

`simOut = sim('model', ParameterStruct);` causes Simulink software to simulate the block diagram, *`model`*, using the parameter values specified in the structure *`ParameterStruct`*.

`simOut = sim('model', ConfigSet);` causes Simulink software to simulate the block diagram, *`model`*, using the configuration settings specified in the model configuration set, *`ConfigSet`*.

Input Arguments

model

Name of model to simulate.

ParameterName

Name of a simulation parameter. Get a list of simulation parameters for the model `vdp` by enter the following in the MATLAB Command Window:

```
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' ``.

In addition, the `sim` command accepts the following parameters:

Parameter	Description
`ConcurrencyResolvingToFileSuffix`	(Rapid Accelerator mode only) Appends this suffix string 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`.
`Debug 'on' | {'off'} |` *`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. Default is `'off'`.
`RapidAcceleratorParameterSets`	(Rapid Accelerator mode only) Returns structure that contains run-time parameters for running Rapid Accelerator simulations in `parfor`. See "sim in parfor with Rapid Accelerator Mode".
`RapidAcceleratorUpToDateCheck {'on'} | 'off'`	(Rapid Accelerator mode only) Enables/disables up-to-date check. If you set this value to `'off'`, the software does not perform an up-to-date check. If you call the `sim` command from `parfor`, set this value to `'off'`. Default is `'on'`.
`SrcWorkspace {base} | 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. Default is the base workspace.

Parameter	Description
`TimeOut` *`timeout`*	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 an error.
`Trace '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 software issues a warning message and continues the simulation.

Value

Value of the simulation parameter. Get the value of the simulation parameter `StopTime` by entering:

```
configSetParamValue = get_param(configSet, 'StopTime')
```

ParameterStruct

A structure containing parameter settings

ConfigSet

A configuration set

Output Arguments

simOut

`Simulink.SimulationOutput` object containing the simulation outputs—logged time, states, and signals

Definitions

For all three formats of the `sim` command, the input(s) are parameter specifications that override those defined on the 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 block, the parameter specifications are applied to the top model.

For additional details about the `sim` command, see “Run Simulation Using the sim Command”.

Examples

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*.

Simulate Model with sim Command Line Options

Specify parameter name value-pairs to the `sim` command:

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...
            'StopTime', '30', ...
            'ZeroCross','on', ...
            'SaveTime','on','TimeSaveName','tout', ...
            'SaveState','on','StateSaveName','xoutNew',...
            'SaveOutput','on','OutputSaveName','youtNew',...
            'SignalLogging','on','SignalLoggingName','logsout')
```

```
Simulink.SimulationOutput:
       tout: [95x1 double]
    xoutNew: [95x2 double]
    youtNew: [95x2 double]
```

Simulate Model with sim Command Line Options in Structure

Specify parameter 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)
```

```
Simulink.SimulationOutput:
    xoutNew: [65x2 double]
    youtNew: [65x2 double]
```

Simulate Model with sim Command Line Options in Configuration Set

Specify parameter name-value pairs in configuration set `mdl_cs` for the `sim` command:

```
mdl = 'vdp';
load_system(mdl)
simMode = get_param(mdl, 'SimulationMode');
set_param(mdl, 'SimulationMode', 'rapid')
cs = getActiveConfigSet(mdl);
mdl_cs = cs.copy;
set_param(mdl_cs,'AbsTol','1e-5',...
         'SaveState','on','StateSaveName','xoutNew',...
         'SaveOutput','on','OutputSaveName','youtNew')
simOut = sim(mdl, mdl_cs);
set_param(mdl, 'SimulationMode', simMode)
```

See Also

`parfor` | Rapid Accelerator Simulations Using PARFOR | “sim in parfor with Rapid Accelerator Mode” | `sldebug` | “Backwards Compatible Syntax” | “Configuration Parameters Dialog Box Overview”

Purpose

Plot simulation data in figure window

Syntax

```
simplot(data)
simplot(time,data)
simplot(data,ports)
simplot(data,'diff')

hfig = simplot(data)
haxes = simplot(data)
hlins = simplot(data)
```

Description

`simplot(data)` returns a Handle Graphics figure window that shows a plot of simulation data.

`simplot(time,data)` returns a figure window that shows simulation data plotted against a separate time vector you specify.

`simplot(data,ports)` plots simulation data from specified ports of a multiport scope block.

`simplot(data,'diff')` plots the differences between multiple data sets when `data` is a cell array of structures or matrices.

`hfig = simplot(data)` returns the handle for the figure window.

`haxes = simplot(data)` returns the handles for the figure axes.

`hlins = simplot(data)` returns the handles for the lines in the figure.

Input Arguments

time - Simulation time

vector

Simulation time specified as a vector. This is the vector of sample times produced by an output block when you select `Array` or `Structure` as the simulation output format. If the format of the data is `Structure with time`, `simplot` ignores this argument. For more information, see "Data Import/Export Pane".

data - Simulation data

array, structure, structure with time

Simulation data, specified as an array, structure, or structure with time. For more information, see "Data Import/Export Pane".

Example: `data = yout; simplot(data)`

ports - Port indices

vector

Port indices of a mutliport scope block, specified as a vector. The data from each port displays in a separate subplot.

Example: `ports = [1,3]; simplot(data,ports)`

Output Arguments

hfig - Figure window handle

Handle for the figure window. For more information about using figure window handles, see "Figure Windows".

haxes - Axes handle

Handle for the figure axes. For more information about using figure axes handles, see "Graphics Objects".

hlins - Lines handle

Handle for the figure lines. For more information about using figure line handles, see "Graphics Objects".

Examples

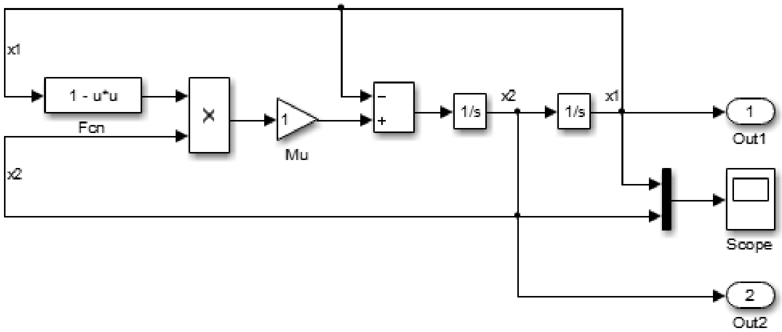
Plot Simulation Data in a Figure Window

This example shows how to plot simulation data from two output blocks in a figure window.

Open a sample model.

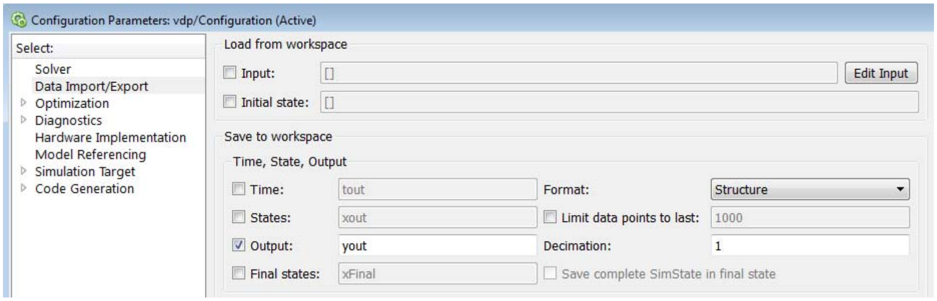
```
model_name = 'vdp';
open_system(model_name)
```

The sample model contains two Outport blocks.


x1
1 - u*u
Fcn
x2
×
1
Mu
−
+
1/s
x2
1/s
x1
1
Out1
Scope
2
Out2

In the Simulink Model Editor, set these Data Import/Export model configuration parameters:

- Select the **Output** check box to save simulation output values in the named workspace variable `yout`.
- Set **Format** to `Structure` as the format to store the output values.


Select:
Solver
Data Import/Export
Optimization
Diagnostics
Hardware Implementation
Model Referencing
Simulation Target
Code Generation
Edit Input

Configuration Parameters: vdp/Configuration (Active)

Load from workspace

☐ Input: []

☐ Initial state: []

Save to workspace

Time, State, Output

☐ Time: tout　Format: Structure

☐ States: xout　☐ Limit data points to last: 1000

☑ Output: yout　Decimation: 1

☐ Final states: xFinal　☐ Save complete SimState in final state

Click **OK**.

Simulate the model and plot the simulation output in a figure window.

```
sim(model_name);
data = yout;
simplot(data)
```

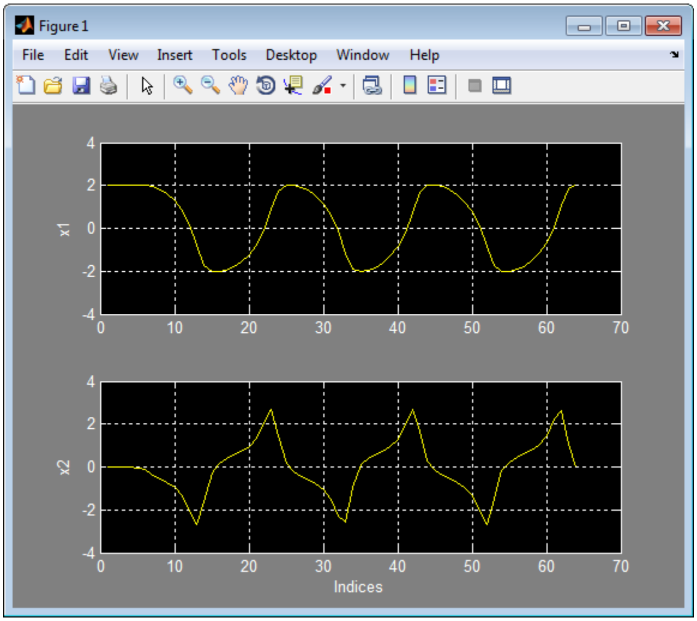


Figure 1
File Edit View Insert Tools Desktop Window Help
x1
4
2
0
-2
-4
0
10
20
30
40
50
60
70
x2
4
2
0
-2
-4
0
10
20
30
40
50
60
70
Indices

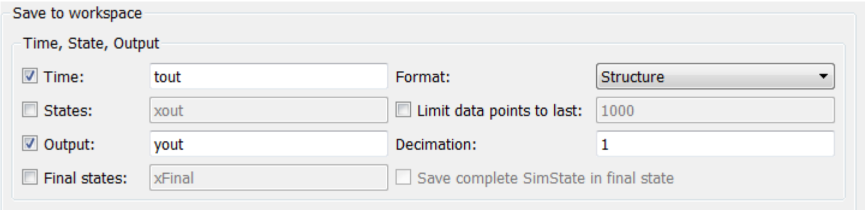
Plot Simulation Data Versus Time in a Figure Window

This example shows how to plot simulation data from two output blocks, specifying a vector of simulation times for `simplot` to use in the named variable `tout`.

Open a sample model and programmatically set model configuration parameters.

```
model_name = 'vdp';
open_system(model_name);
set_param(gcs,'SaveOutput','on');
set_param(gcs,'SaveTime','on');
set_param(gcs,'SaveFormat','Structure')
```

After you execute these commands, the Data Import/Export pane in the model Configuration Parameters dialog box looks like this:



Save to workspace

Time, State, Output

☑ Time: tout

☐ States: xout

☑ Output: yout

☐ Final states: xFinal

Format: Structure

☐ Limit data points to last: 1000

Decimation: 1

☐ Save complete SimState in final state

Simulate the model and plot the simulation output in a figure window.

```
sim(model_name);
simplot(tout,yout)
```

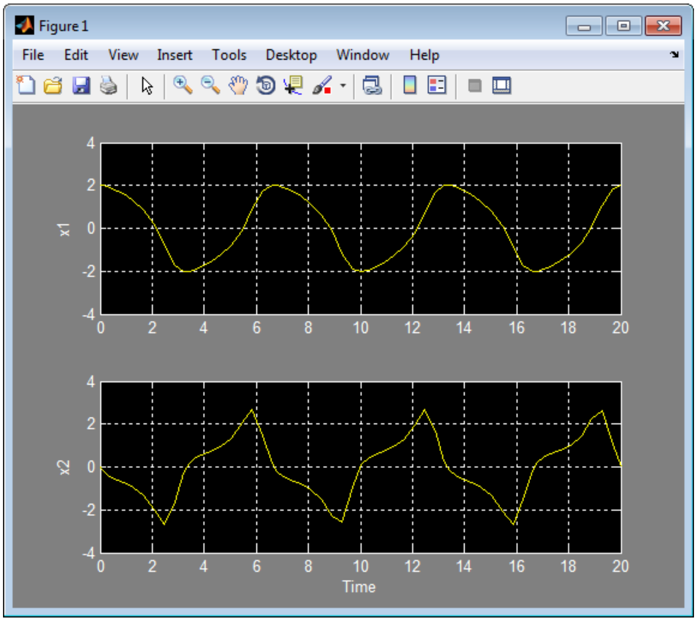


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
x1
4
2
0
-2
-4
0
2
4
6
8
10
12
14
16
18
20
x2
4
2
0
-2
-4
0
2
4
6
8
10
12
14
16
18
20
Time

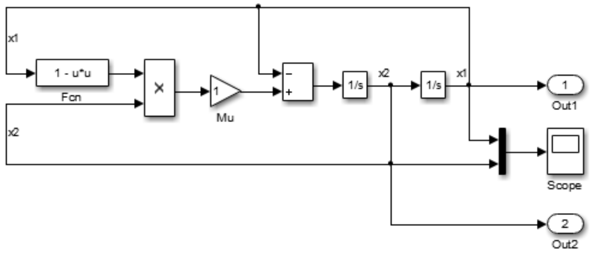
Overlay Plots in a Figure Window

This example shows how to plot simulation data from two output blocks, overlaying the data from each block in a single plot.

Open a sample model.

```
model_name = 'vdp';
open_system(model_name)
```

The sample model contains two Outport blocks.



In the Simulink Model Editor, set these Data Import/Export model configuration parameters:

- Select the **Output** check box. In the **Output** value text box, enter comma-separated variable names, `yout` and `yout1`. This setting saves simulation data from the two Outport blocks in these workspace variables.
- Set **Format** to `Structure` to save simulation output data in this format.

Simulate the model.

```
sim(model_name);
```

Store `yout` and `yout1` in a single cell array, `data`.

```
data = {yout,yout1};
```

Plot the simulation output in a figure window.

```
simplot(data)
```

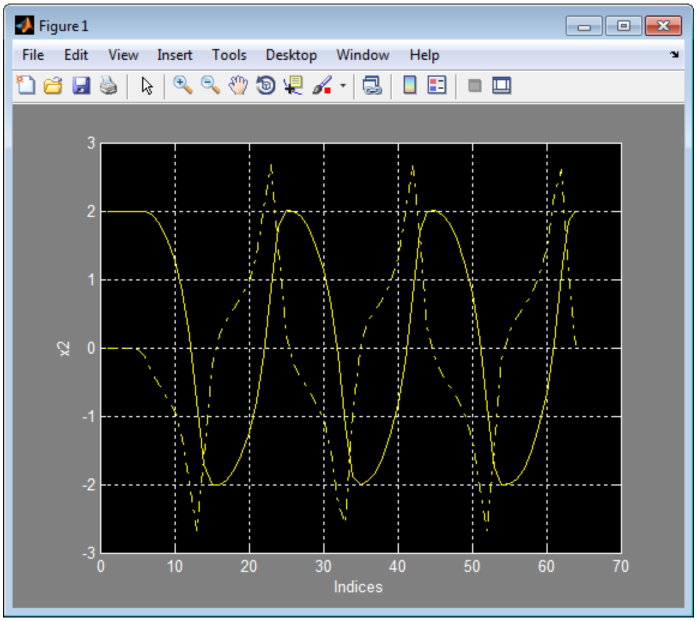


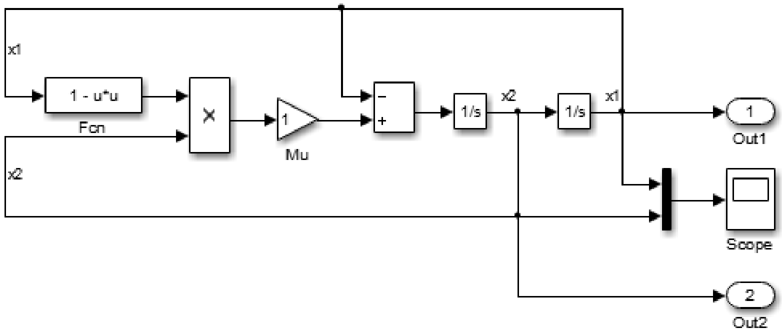
Figure 1
File Edit View Insert Tools Desktop Window Help
x2
Indices
3
2
1
0
-1
-2
-3
0
10
20
30
40
50
60
70

Plot Data From a Multiport Scope

This example shows how to plot simulation data from specified ports of a multiport scope block.

Open a sample model.

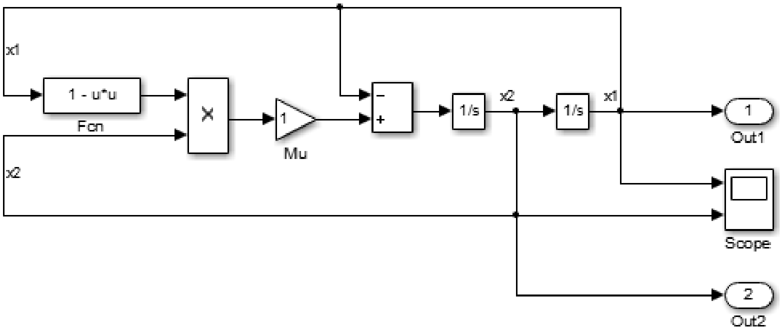
```
model_name = 'vdp';
open_system(model_name)
```



Modify the model to replace the Mux input to the Scope block with separate signals:

1. Delete the Mux block.
2. Double-click the Scope block.
3. On the **Scope** window toolbar, click the **Parameters** button ⚙.
4. In the **General** tab, set **Number of axes** to `2`.
5. In the **History** tab, select the **Save data to workspace** check box and set **Format** to `Structure`.
6. Click **OK**.

Connect signal lines to the modified Scope block as shown:


x1
1 - u*u
Fcn
x2
Mu
1
1/s
x2
1/s
x1
1
Out1
Scope
2
Out2

Save the model on your computer with a new name, for example, `vdp_modified`.

Simulate the model. Plot simulation data from only the first input port of the Scope block.

```
model_name = 'vdp_modified';
sim(model_name);
simplot(ScopeData,[1])
```

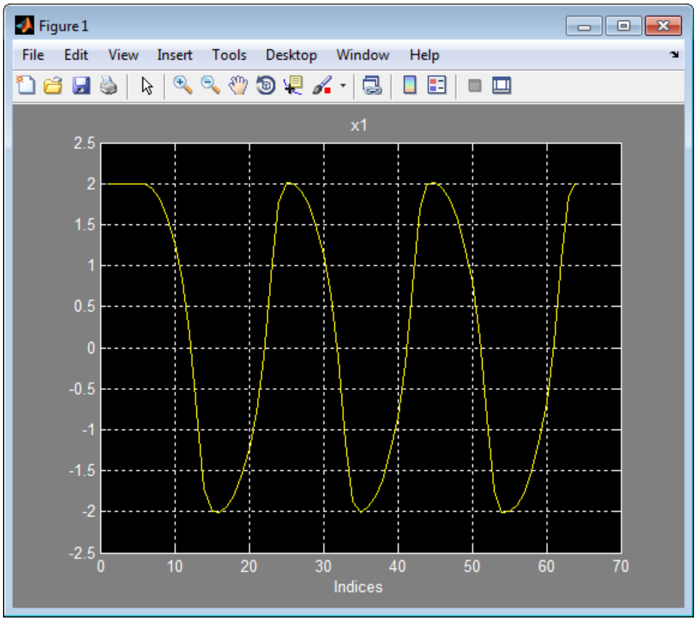


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
x1
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
10
20
30
40
50
60
70
Indices

Plot simulation data from both input ports of the Scope block.

```
simplot(ScopeData,[1,2])
```

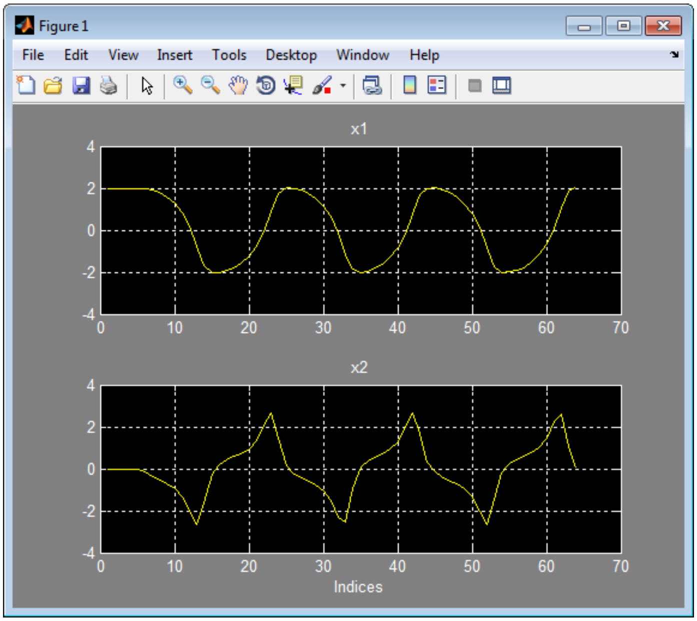


Figure 1
File Edit View Insert Tools Desktop Window Help
x1
4
2
0
-2
-4
0 10 20 30 40 50 60 70
x2
4
2
0
-2
-4
0 10 20 30 40 50 60 70
Indices

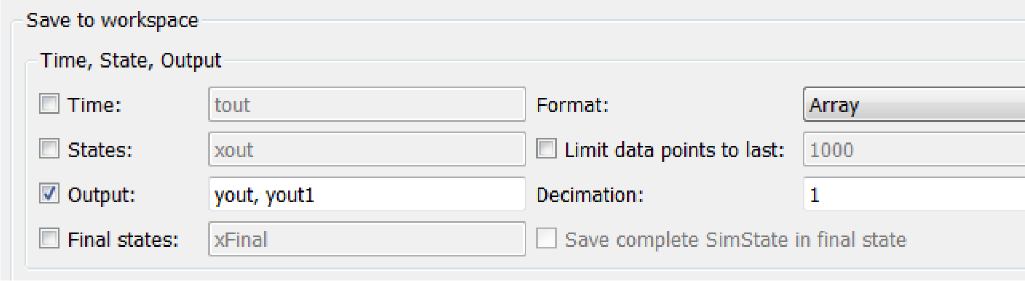
Plot Differences Between Data Sets in a Figure Window

This example shows how to plot the differences between two sets of simulation data.

Open a sample model and set model configuration parameters.

```
model_name = 'vdp';
open_system(model_name);
set_param(gcs,'SaveOutput','on');
set_param(gcs,'OutputSaveName','yout,yout1')
set_param(gcs,'SaveFormat','Array');
```

When you execute these commands, the model Configuration Parameters dialog box looks like this:



Save to workspace

Time, State, Output

☐ Time: tout

Format: Array

☐ States: xout

☐ Limit data points to last: 1000

☑ Output: yout, yout1

Decimation: 1

☐ Final states: xFinal

☐ Save complete SimState in final state

Simulate the model.

```
sim(model_name);
```

Store `yout` and `yout1` in a single cell array, `data`.

```
data = {yout,yout1};
```

Plot the differences between the two data sets in a figure window.

```
simplot(data,'diff')
```

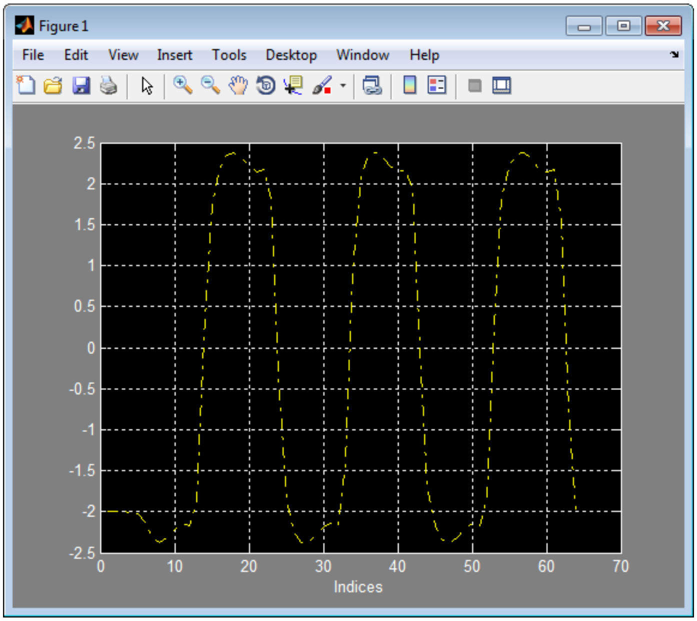


Figure 1
File Edit View Insert Tools Desktop Window Help
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
10
20
30
40
50
60
70
Indices

Use Figure Handles Returned by simplot

This example shows some sample uses of figure handles returned by `simplot`.

Open a sample model and set model configuration parameters.

```
model_name = 'vdp';
open_system(model_name);
set_param(gcs,'SaveOutput','on');
set_param(gcs,'SaveTime','on');
set_param(gcs,'SaveFormat','Structure')
```

Simulate the model for 20 seconds.

```
sim(model_name, 20);
```

Plot the simulation output against time in a figure window and get figure handles.

```
[hfig,haxes,hlines] = simplot(tout,yout);
```

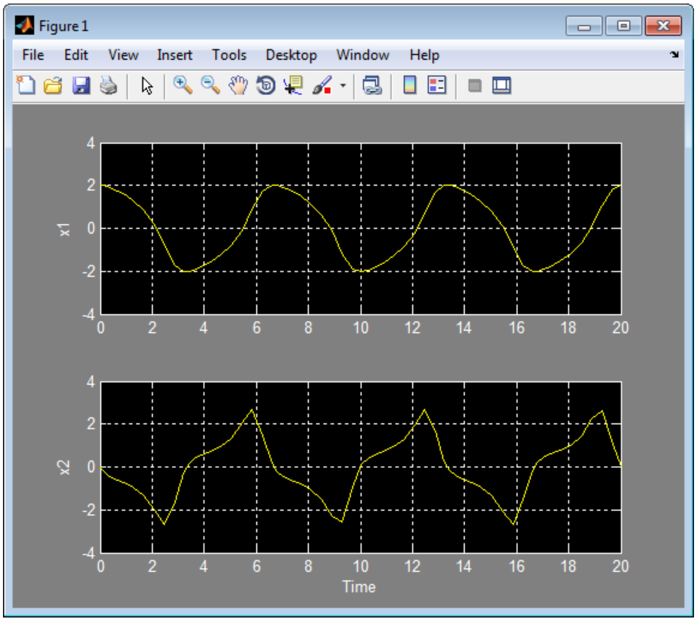


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
x1
4
2
0
-2
-4
0
2
4
6
8
10
12
14
16
18
20
x2
4
2
0
-2
-4
0
2
4
6
8
10
12
14
16
18
20
Time

Use the figure handle to set a white figure background.

```
whitebg(hfig)
```

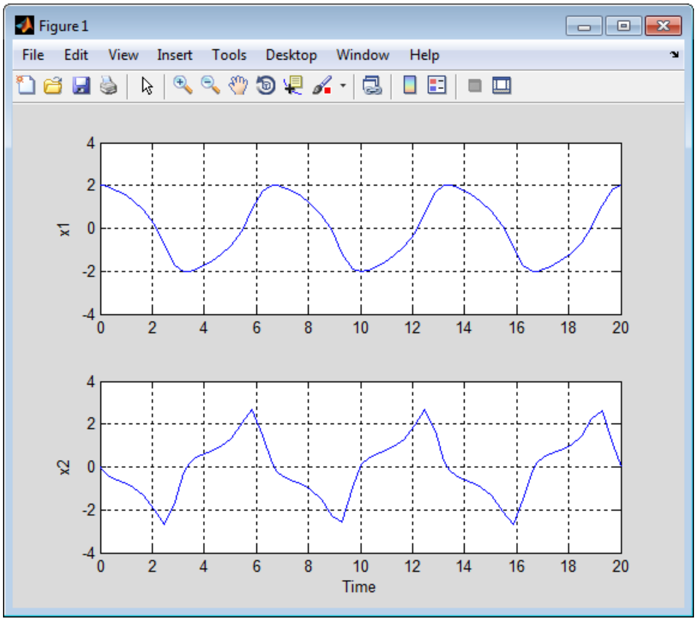


Figure 1
File Edit View Insert Tools Desktop Window Help
x1
4
2
0
-2
-4
0 2 4 6 8 10 12 14 16 18 20
x2
4
2
0
-2
-4
0 2 4 6 8 10 12 14 16 18 20
Time

Use the axes handle with the `NextPlot` property.

```
set(haxes,'NextPlot','add')
```

The software holds existing plots in place when you make another call to `simplot`.

Resimulate the model for 10 seconds.

```
sim(model_name, 10);
```

Replot the simulation data versus time.

```
[hfig,haxes,hlines] = simplot(tout,yout);
```

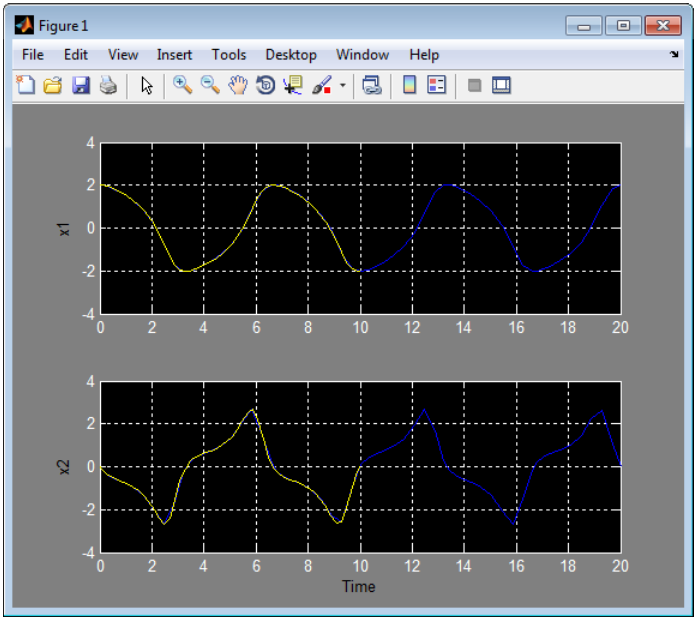


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
x1
4
2
0
-2
-4
0
2
4
6
8
10
12
14
16
18
20
x2
4
2
0
-2
-4
0
2
4
6
8
10
12
14
16
18
20
Time

When `simplot` plots the new simulation data, because you set the `NextPlot` property, it retains the previous plots. The figure window displays the new data in yellow. The previous simulation data remains in blue. However, the background color reverts to its default of black.

Use the plot lines handle to set all line styles to dashed.

```
set(hlines, 'LineStyle', '--')
```

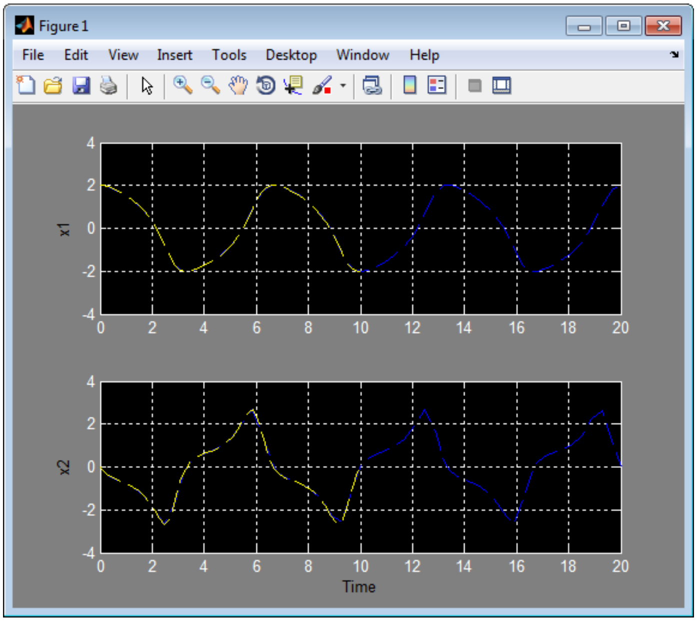


Figure 1
File Edit View Insert Tools Desktop Window Help
x1
4
2
0
-2
-4
0 2 4 6 8 10 12 14 16 18 20
x2
4
2
0
-2
-4
0 2 4 6 8 10 12 14 16 18 20
Time

See Also `plot`

Concepts

- “Data Import/Export Pane”
- “Figure Windows”
- “Graphics Objects”

Purpose Open Simulink block library

Syntax

```
simulink
simulink('open')
simulink('close')
```

Description `simulink` or `simulink('open')` opens the Simulink Library Browser. `simulink('close')` closes the Library Browser.

simulinkproject

Purpose

Open Simulink Project Tool

Syntax

```
simulinkproject
simulinkproject(projectPath)

proj = simulinkproject
proj = simulinkproject(projectPath)
```

Description

`simulinkproject` opens the Simulink Project Tool 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 the `.prj` file or folder in `projectPath`.

`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.

`proj = simulinkproject(projectPath)` opens the Simulink project specified by `projectPath` and returns a project object.

Input Arguments

projectPath - Full path to project file or folder

string

Full path to project `.prj` file, or the path to the project root folder, specified as a string.

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	Value
Name	Project name	String
Categories	Categories of project labels	Cell array of strings
Files	Paths and names of project files	Cell array of strings
RootFolder	Full path to project root folder	String
Shortcuts	Shortcut files in project	Cell array of strings

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. For example:

```
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. For example,

```
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'
   Categories: [1x2 slproject.Category]
     Shortcuts: [1x6 slproject.Shortcut]
            Files: [1x24 slproject.ProjectFile]
   RootFolder: 'C:\Temp\20121107T135235\airframe'
```

Find Project Commands

Find out what you can do with your project.

```
methods(proj)

Methods for class slproject.ProjectManager:

addFile
export
isLoaded
removeFile
close
findCategory
reload
removeLabel
createCategory
findFile
removeCategory
```

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 =

  1x24 ProjectFile array with properties:

    Path
    Labels
```

Use indexing to access files in this list. The following command gets file number 8. Each file has two properties describing its path and attached labels.

```
proj.Files(8)

ans =

  ProjectFile with properties:

      Path: 'C:\Temp\project1\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
```

Examine the labels of the eighth file.

```
proj.Files(8).Labels

ans =

  Label with properties:

File: 'C:\Temp\project1\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\project1\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
```

Find out what you can do with the file.

```
methods(myfile)

Methods for class slproject.ProjectFile:

addLabel
removeLabel
findLabel
```

Tips

Alternatively, you can use `slproject.loadProject` to load a project, and `slproject.getCurrentProject` to get a project object. Use `simulinkproject` to open projects and explore projects interactively. Use `slproject.getCurrentProject` and `slproject.loadProject` for project automation scripts.

See Also

`slproject.getCurrentProjectslproject.loadProjectaddFileremoveFilefindFilefindL`

Related Examples

- “Create a New Project to Manage Existing Files”
- “Open Recent Projects”
- “Retrieve a Working Copy of a Project from Source Control”
- “Automate Project Management Tasks”

Concepts

- “What Are Simulink Projects?”

Purpose

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 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 string that represents the annotation of a specific sample time (e.g., 'D1')

- `OwnerBlock` — For asynchronous and variable sample times, a string containing the full path to the block that controls the sample time. For all other types of sample times, an empty 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`

Simulink.BlockDiagram.addBusToVector

Purpose

Add Bus to Vector blocks to convert virtual bus signals into vector signals

Syntax

```
[DstBlocks,
   BusToVectorBlocks] = Simulink.BlockDiagram.addBusToVector('
   model')
[DstBlocks,
   BusToVectorBlocks] = Simulink.BlockDiagram.addBusToVector('
   model', includeLibs)
[DstBlocks,
   BusToVectorBlocks] = Simulink.BlockDiagram.addBusToVector('
   model', includeLibs, reportOnly)
```

Description

`[`*`DstBlocks`*`, `*`BusToVectorBlocks`*`] = 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. Before executing this function, you must do the following:

1. Set **Simulation > Model Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to `error`, or equivalently, execute `set_param (`*`model`*`, 'StrictBusMsg', 'ErrorLevel1')`.
2. Ensure that the model compiles without error.
3. Save the model.

`[`*`DstBlocks`*`, `*`BusToVectorBlocks`*`] = Simulink.BlockDiagram.addBusToVector('`*`model`*`', `*`includeLibs`*`)` is equivalent to `[`*`DstBlocks`*`, `*`BusToVectorBlocks`*`] = Simulink.BlockDiagram.addBusToVector(`*`model`*`)` if *`includeLibs`* is `false`.

If *`includeLibs`* is `true`, the function searches library blocks rather than excluding them.

`[DstBlocks, BusToVectorBlocks] = Simulink.BlockDiagram.addBusToVector('model', includeLibs, reportOnly)` is equivalent to `[DstBlocks, BusToVectorBlocks] = Simulink.BlockDiagram.addBusToVector(model, includeLibs)` if *`reportOnly`* is `true`.

If *`reportOnly`* is `false`, the function inserts a Bus to Vector block into each bus that is used as a vector in any block that it searches. The search excludes or includes library blocks as specified by *`includeLibs`*. The insertion replaces the implicit use of a bus as a vector with an explicit conversion of the bus to a vector. The signal's source and destination blocks are unchanged by this insertion.

If `Simulink.BlockDiagram.addBusToVector` adds Bus to Vector blocks to the model or any library, the function permanently changes the saved copy of the diagram. Be sure to back up the model and any libraries before calling the function with *`reportOnly`* specified as `false`.

If `Simulink.BlockDiagram.addBusToVector` changes a library block, the change affects every instance of that block in every Simulink model that uses the library. To preview the effects of the change on blocks in all models, call `Simulink.BlockDiagram.addBusToVector` with *`includeLibs`* = `true` and *`reportOnly`* = `true`, then examine the information returned in *`DstBlocks`*.

The Bus to Vector block is intended only for use in existing models to facilitate the elimination of implicit conversion of buses into vectors. New models and new parts of existing models should avoid implicitly using buses as vectors, and should not use Bus to Vector blocks for any purpose. See Avoiding Mux/Bus Mixtures for more information about using `Simulink.BlockDiagram.addBusToVector`.

Input Arguments

model

Model name or handle

includeLibs

Boolean specifying whether to search library blocks (`true`) or only the top-level model (`false`).

Default: `false`

reportOnly

Boolean specifying whether to change the model (`false`) or just generate a report (`true`).

Default: `true`

Output Arguments

DstBlocks

An array of structures that contain information about blocks that are connected to buses but treat the buses as vectors. If no such blocks exist the array has 0 length. Each structure in the array contains the following fields:

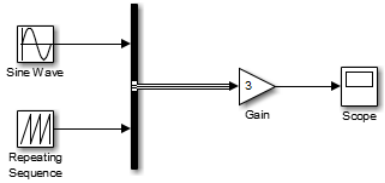
`BlockPath`	String specifying the path to the block to which the bus connects
`InputPort`	Integer specifying the input port to which the bus connects
`LibPath`	If the block is a library block instance, and *`includeLibs`* is `true`, the path to the source library block. Otherwise, `LibPath` is empty ([]).

BusToVectorBlocks

If *`reportOnly`* is `false`, and *`model`* contains any buses used as vectors, a cell array containing the path to each Bus to Vector block that was added to the model. Otherwise, *`BusToVectorBlocks`* is empty ([]).

Examples

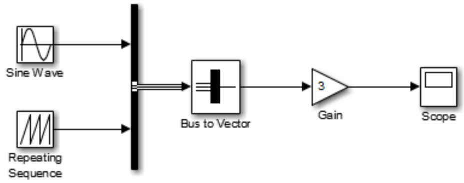
The following model simulates correctly, but the input to the Gain block is a bus, while the output is a vector. Thus the Gain block uses a block as a vector.


Sine Wave
Repeating Sequence
3
Gain
Scope

If the model shown is open as the current model, you can eliminate the implicit conversion with the following command:

```
Simulink.BlockDiagram.addBusToVector(gcs, false, false)
```

Rebuilding and simulating the model then gives this result:


Sine Wave
Repeating Sequence
Bus to Vector
3
Gain
Scope

The Gain block no longer implicitly converts the bus to a vector; the inserted Bus to Vector block performs the conversion explicitly. Note that the results of simulation are the same for both models. The Bus to Vector block is virtual, and never affects simulation results, code generation, or performance.

See Also

Bus Assignment | Bus Creator | Bus Selector | Bus to Vector | `Simulink.Bus` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createObject` | `Simulink.BusElement` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save`

How To

- “Mux Signals”
- “Composite Signals”
- Avoiding Mux/Bus Mixtures

Simulink.BlockDiagram.buildRapidAcceleratorTarget

Purpose

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><code>modelChecksum</code></td><td colspan="2">1x4 vector that encodes the structure of the model.</td></tr>
<tr><td rowspan="7"><code>parameters</code></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>
<tr><td><code>dataTypeName</code></td><td>The data type name, for example, <code>double</code>.</td></tr>
<tr><td><code>dataTypeId</code></td><td>Internal data type identifier for use by Simulink Coder.</td></tr>
<tr><td><code>complex</code></td><td>Complex type or real type specification. Value is <code>0</code> if real, <code>1</code> if complex.</td></tr>
<tr><td><code>dtTransIdx</code></td><td>Internal data type identifier for use by Simulink Coder.</td></tr>
<tr><td><code>values</code></td><td>All values associated with this entry in the <code>parameters</code> substructure.</td></tr>
</table>

Element	Description		
	`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: vdp
### Successfully built the rapid accelerator target for model: vdp

rtp =

    modelChecksum: [2.6812e+09 2.7198e+09 589261472 4.0180e+09]
       parameters: [1x1 struct]
```

Concepts

- “How Acceleration Modes Work”
- “Choosing a Simulation Mode”
- “Design Your Model for Effective Acceleration”

Simulink.BlockDiagram.copyContentsToSubSystem

Purpose

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`

How To

- “Systems and Subsystems”
- “Create a Subsystem”

Simulink.BlockDiagram.createSubSystem

Purpose

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 within the current model to the new subsystem.

Input Arguments

blocks

An array of block handles

Default: `[]`

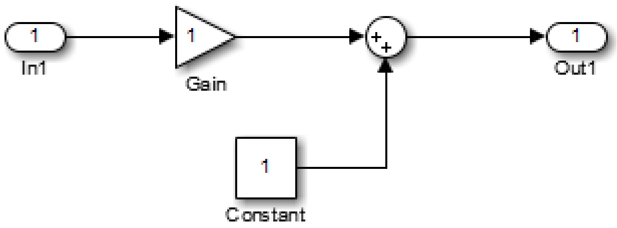
Examples

The following 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
```

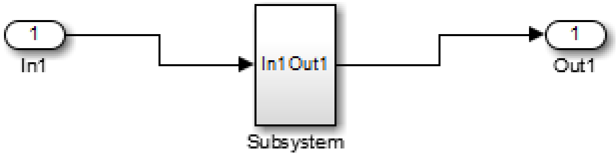
For example, suppose you create the following model and save it as `initial_model.slx`.


1
In1
1
Gain
+
+
1
Out1
1
Constant

Executing

```
convert2subsys('initial_model');
```

converts this model to create a subsystem:


1
In1
In1
Out1
Subsystem
1
Out1

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.BlockDiagram.deleteContents` | `Simulink.SubSystem.convertToModelReference` | `Simulink.SubSystem.copyContentsToBlockDiagram`

How To

- “Systems and Subsystems”
- “Create a Subsystem”

Simulink.BlockDiagram.deleteContents

Purpose

Delete contents of block diagram

Syntax

`Simulink.BlockDiagram.deleteContents(bdiag)`

Description

`Simulink.BlockDiagram.deleteContents(bdiag)` deletes the contents of the block diagram *`bdiag`*. The block diagram must have already been loaded. The function affects only blocks, lines, and annotations. It does not affect nongraphical information such as configuration sets.

Input Arguments

bdiag

Block diagram name or handle

Examples

Delete the graphical content of an open block diagram named `f14`, including all subsystems:

```
Simulink.BlockDiagram.deleteContents('f14');
```

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.SubSystem.convertToModelReference` | `Simulink.SubSystem.copyContentsToBlockDiagram` | `Simulink.SubSystem.deleteContents`

How To

- “Modeling”
- “Create a Subsystem”

Purpose

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 string 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)
  ```

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.BlockDiagram.getInitialState

Purpose Return initial state structure of block diagram

Syntax

```
x0 = Simulink.BlockDiagram.getInitialState('model')
```

Description

`x0 = Simulink.BlockDiagram.getInitialState('model')` returns the initial state structure of the block diagram specified by the input argument *`model`*. This state structure can be used to specify the initial state vector in the **Configuration Parameters** dialog box or to provide an initial state condition to the linearization commands.

The command returns `x0`, 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 will be shown for S-function blocks

  The name of the discrete state will be 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 state

Using the state structure simplifies specifying initial state values for models with multiple states, as each state is associated with the full path to its parent block.

See Also

`linmod`

Simulink.BlockDiagram.getSampleTimes

Purpose

Return all sample times associated with model

Syntax

`ts = Simulink.BlockDiagram.getSampleTimes('`*`model`*`')`

Input Arguments

`model`

Name or handle of a Simulink model

Output Arguments

`ts`

The command returns `ts` which is 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 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 string that represents the annotation of a specific sample time (e.g., 'D1')

- `OwnerBlock` — For asynchronous and variable sample times, a string containing the full path to the block that controls the sample time. For all other types of sample times, an empty 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.BlockDiagram.getSampleTimes('`*`model`*`')` performs an update diagram and then returns the sample times associated with the block diagram specified by the input argument *`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.

See Also

`Simulink.Block.getSampleTimes`

Simulink.BlockDiagram.loadActiveConfigSet

Purpose 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.

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"** parameter to `ert.tlc`.

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')
```

See Also

`Simulink.BlockDiagram.saveActiveConfigSet` | `Simulink.ConfigSet` | `attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

How To

- "Load a Saved Configuration Set"

Simulink.BlockDiagram.saveActiveConfigSet

Purpose

Save active configuration set of model

Syntax

`Simulink.BlockDiagram.saveActiveConfigSet('`*`model`*`',`
`'`*`filename`*`')`

Description

`Simulink.BlockDiagram.saveActiveConfigSet('`*`model`*`', '`*`filename`*`')` saves the active configuration set of a model to a `.m` or `.mat` file. *`model`* is the name or handle of the model. *`filename`* is the name of the file to save the model configuration set. If you specify a `.m` extension, the file contains a function that creates a configuration set object. If you specify a `.mat` extension, the file contains a configuration set object. If you do not provide a file extension, the active configuration set is saved to a file with a `.m` extension. Do not specify *`filename`* to be the same as a model name; otherwise the software cannot determine which file contains the configuration set object when loading the file.

Examples

Save the configuration set from the `sldemo_counters` model to `my_config_set.m`.

```
% Open the sldemo_counters model
sldemo_counters
% Save the active configuration set to my_config_set.m
Simulink.BlockDiagram.saveActiveConfigSet('sldemo_counters', 'my_config_set.m')
```

See Also

`Simulink.BlockDiagram.loadActiveConfigSet` | `Simulink.ConfigSet` | `attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

How To

- “Save a Configuration Set”

Purpose

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. The inverse function is `Simulink.Bus.objectToCell`.

Input Arguments

busCells

A cell array of cell arrays in which each subordinate array represents a bus object and contains the following data:

```
{BusName, HeaderFile, Description, DataScope,
Alignment, Elements}
```

The *`Elements`* field is an array containing the following data for each element:

```
{ElementName, Dimensions, DataType,
SampleTime, Complexity, SamplingMode, DimensionsMode,
Min, Max, DocUnits, Description}
```

See Also

Bus Assignment | Bus Creator | Bus Selector | Bus to Vector | `Simulink.Bus` | `Simulink.Bus.createMATLABStruct` | `Simulink.Bus.createObject` | `Simulink.BusElement` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save`

How To

- “Composite Signals”

Simulink.Bus.createMATLABStruct

Purpose

Create MATLAB structures with same hierarchy and attributes as bus signals

Syntax

```
structForBus = Simulink.Bus.createMATLABStruct(busSource)
structForBus = Simulink.Bus.createMATLABStruct(busSource,
   partialValues)
structForBus = Simulink.Bus.createMATLABStruct(busSource,
   partialValues,dims)

cellStructsForBuses =
Simulink.Bus.createMATLABStruct(arrayOfP
   ortHandles)
cellStructsForBuses =
Simulink.Bus.createMATLABStruct(arrayOfP
   ortHandles,cellOfPartialStructs)
```

Description

`structForBus = Simulink.Bus.createMATLABStruct(busSource)` creates a MATLAB structure with the same hierarchy and attributes (such as type and dimension) as a bus signal `busSource`. Specify either the name of a bus object or the handle of a bus signal port. The resulting structure uses the ground values of the bus signal.

`structForBus = Simulink.Bus.createMATLABStruct(busSource,partialValues)` creates a structure that uses specified values `partialValues` instead of the corresponding ground values of the bus signal.

`structForBus = Simulink.Bus.createMATLABStruct(busSource,partialValues,dims)` creates a structure with the specified dimensions. For the `busSource` argument, use the name of a bus object.

`cellStructsForBuses = Simulink.Bus.createMATLABStruct(arrayOfPortHandles)` creates a cell array of structures for bus signal ports `arrayOfPortHandles`.

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.

```
cellStructsForBuses =
Simulink.Bus.createMATLABStruct(arrayOfPortHandles,cellOfPartialStruct
```

creates a cell array of structures. The corresponding fields in the structures use the specified values `cellOfPartialStructs` instead of the ground values.

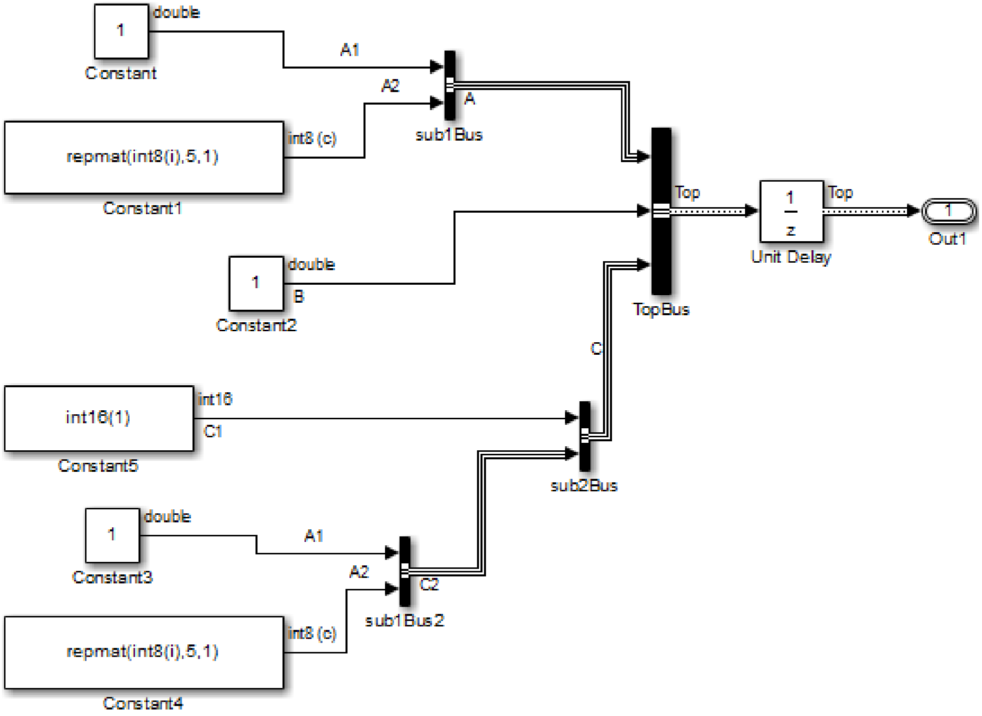
Examples

MATLAB Structure from Bus Object

Open a Simulink model and simulate it.

```
run([docroot '/toolbox/simulink/ug/examples/signals/busic_example.mdl
sim('busic_example')
```

The simulated `busic_example` model looks like this:


1
double
Constant
A1
A2
A
sub1Bus
repmat(int8(i),5,1)
int8 (c)
Constant1
Top
1
z
Unit Delay
Top
1
Out1
TopBus
1
double
B
Constant2
C
int16(1)
int16
C1
Constant5
sub2Bus
1
double
Constant3
A1
A2
C2
sub1Bus2
repmat(int8(i),5,1)
int8 (c)
Constant4

Create a MATLAB structure using the bus object `Top`, which the `busic_example` model loads.

```
mStruct = Simulink.Bus.createMATLABStruct('Top')
```

```
mStruct =

    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 =

    A1: 3
    A2: [5x1 int8]sim
('busic_example')
```

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.

Close the system.

```
close_system('busic_example')
```

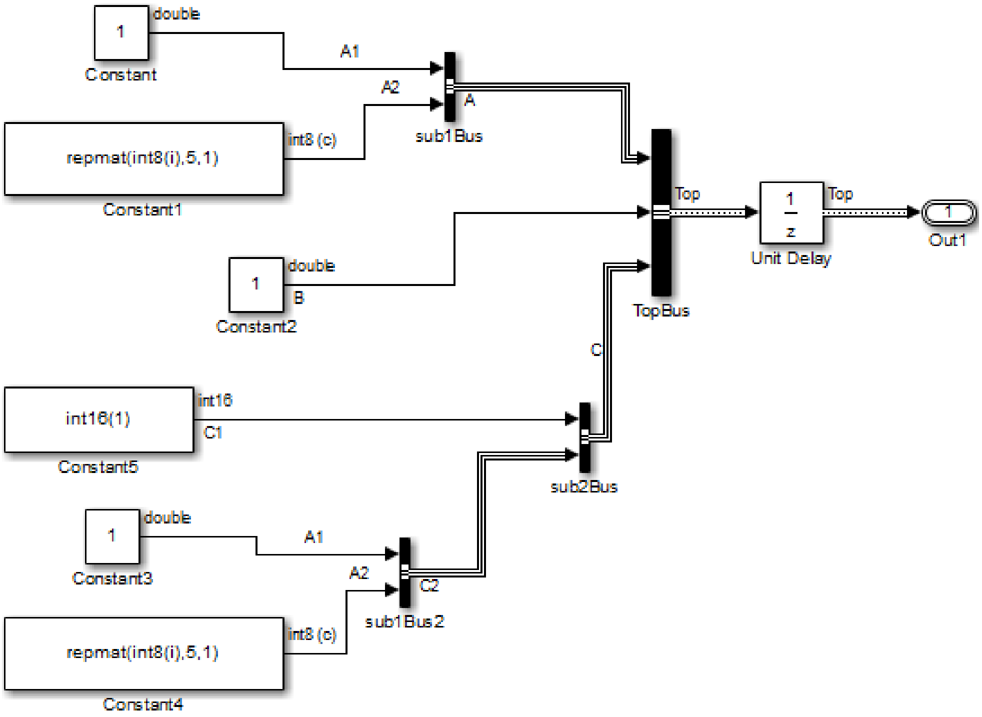
MATLAB Structure from Bus Port and Partial Structure

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')
```

The simulated `busic_example` model looks like this:


double
1
Constant
A1
A2
repmat(int8(i),5,1)
int8 (c)
Constant1
A
sub1Bus
Top
Top
1
z
Unit Delay
1
Out1
double
1
B
Constant2
TopBus
C
int16(1)
int16
C1
Constant5
sub2Bus
double
1
Constant3
A1
A2
C2
sub1Bus2
repmat(int8(i),5,1)
int8 (c)
Constant4

Find the port handle for the Bus Creator block port that produces the `Top` bus signal.

```
ph = get_param('busic_example/TopBus','PortHandles')
```

```
ph = 

     Inport: [143.0013 144.0013 145.0013]
    Outport: 34.0013
     Enable: []
    Trigger: []
      State: []
      LConn: []
```

```
       RConn: []
    Ifaction: []
```

The `Outport` handle is the handle that you need.

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.

You can use a partial structure as an optional argument when creating a MATLAB structure from a bus object or from a bus port.

Create a MATLAB structure using the port handle (`ph`) for the `TopBus` block. Override the ground values for the `A.A1` and `B` bus elements.

```
mStruct = Simulink.Bus.createMATLABStruct(ph.Outport,PartialStructFor
```

```
mStruct =

    A: [1x1 struct]
    B: 3
    C: [1x1 struct]
```

Close the system.

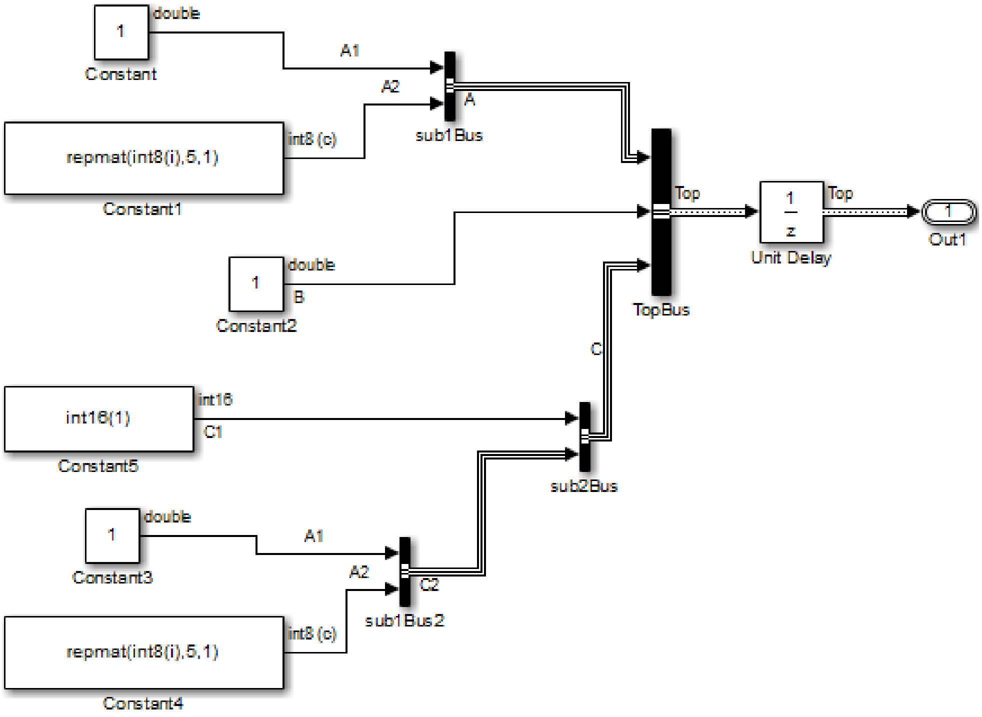
```
close_system('busic_example')
```

MATLAB Structure with Specified Dimensions

Open a Simulink model and simulate it.

```
run([docroot '/toolbox/simulink/ug/examples/signals/busic_example.mdl']);
sim('busic_example')
```

The simulated `busic_example` model looks like this:


1
double
Constant
A1
A2
A
sub1Bus
repmat(int8(i),5,1)
int8 (c)
Constant1
Top
1
z
Top
Unit Delay
1
Out1
1
double
B
Constant2
TopBus
C
int16(1)
int16
C1
Constant5
sub2Bus
1
double
Constant3
A1
A2
C2
sub1Bus2
repmat(int8(i),5,1)
int8 (c)
Constant4

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
```

Create a MATLAB structure using the bus object `Top` (which the `busic_example` model loads), a partial structure, and dimensions for the resulting structure.

```
structForBus = Simulink.Bus.createMATLABStruct...
     ('Top',PartialStructForK,[2 3])

structForBus =

2x3 struct array with fields:

    A
    B
    C
```

Close the system.

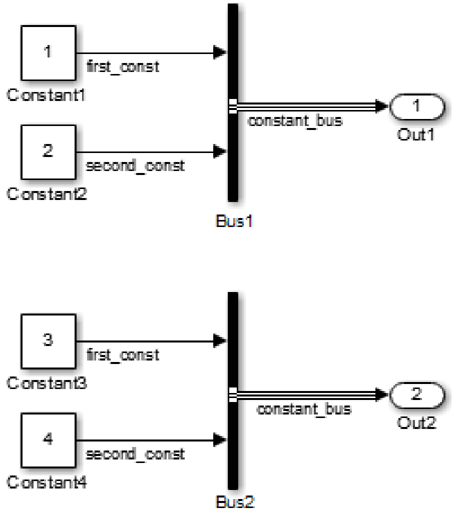
```
close_system('busic_example')
```

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')
```

The simulated `ex_two_outports_create_struct` model looks like this:


1
first_const
Constant1
constant_bus
1
Out1
2
second_const
Constant2
Bus1
3
first_const
Constant3
constant_bus
2
Out2
4
second_const
Constant4
Bus2

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')
```

Input Arguments

busSource - Source representing a bus signal

name of bus object | port handle

Source representing a bus signal for which to create a MATLAB structure, specified as the name of a bus object (a `Simulink.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, then that bus object must be in the MATLAB base workspace. The data type for the bus object is a `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: `structForBus = 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 "Match IC Structure Values to Corresponding Bus Element Data Characteristics".

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. To create a structure for an array of buses, use `dims`.

If you use `dims`, it must be the third argument.

Each dimension element must be an integer that is equal to, or greater than, 1. If you specify `partialValues`, then each dimension element in `dims` must be equal to, or greater than, its corresponding dimension element in the partial structure.

To use ground values, for `partialValues` use an empty matrix (that is, `[]`).

Data Types
`double`

arrayOfPortHandles - Handles of bus signal ports

array

Handles of bus signal ports, specified as an array. If you use the `cellOfPartialStructs` argument, then the number of port handles that you specify in `arrayOfPortHandles` must be the same as the number of partial structures.

Data Types
`double`

cellOfPartialStructs - Handles of bus signal ports

cell array

Handles of bus signal ports, specified as a cell array.

Data Types
`cell`

Output Arguments

structForBus - Bus signal hierarchy and attributes, returned as a MATLAB structure

MATLAB structure

Structure of data with same hierarchy and attributes as a bus signal that you specify with a either a bus object or port handle. The structure uses ground values of the bus signal. To override ground values for specific nodes, specify a `partialValues`partial structure.

The dimensions of `structForBus` 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`.

cellStructsForBuses - Structures with same hierarchy and attributes as bus signals

cell array

Structures with same hierarchy and attributes as bus signals, specified 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. To override other values for specific nodes in the resulting structures specify a `cellOfPartialStructs` cell array of partial structures.

The dimensions of `cellStructsForBuses` depend on the input arguments:

- If you specify only `arrayOfPortHandles`, then the dimension is 1.
- If you also specify `cellOfPartialStructs`, then the dimensions match the dimensions of `cellOfPartialStructs`.

Tips

- 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** icon (⊞).

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

Bus Assignment | Bus Creator | Bus to Vector | `Simulink.Bus` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createObject` | `Simulink.BusElement` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save` | `Simulink.SimulationData.createStructOfTimeseries` | "Specify Initial Conditions for Bus Signals" | "Composite Signals" |

Simulink.Bus.createObject

Purpose

Create bus objects for blocks

Syntax

```
busInfo = Simulink.Bus.createObject('modelName', blks)
busInfo = Simulink.Bus.createObject('modelName', blks,
   fileName)
busInfo = Simulink.Bus.createObject('modelName', blks,
   fileName, 'format')
busInfo = Simulink.Bus.createObject(structVar, fileName,
   'format')
```

Description

`busInfo = Simulink.Bus.createObject('modelName', blks)` 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('modelName', blks, fileName)` saves the bus objects in a MATLAB file that contains a cell array of cell arrays. Each subordinate array represents a bus object and contains the following data:

```
{BusName, HeaderFile, Description, DataScope,
Alignment, Elements}
```

The `Elements` field is an array containing the following data for each element:

```
{ElementName, Dimensions, DataType,
SampleTime, Complexity, SamplingMode, DimensionsMode,
Min, Max,DocUnits, Description}
```

`busInfo = Simulink.Bus.createObject('modelName', blks, fileName, 'format')` saves the bus objects in a file that contains either a cell array of bus information, or the bus objects themselves.

`busInfo = Simulink.Bus.createObject(structVar, fileName, 'format')` creates bus objects in the MATLAB workspace from the numeric MATLAB structure and optionally saves the bus objects in the specified file.

Tips

If you specify a model name, the model must compile successfully before you use the `Simulink.Bus.createObject` command.

Input Arguments

modelName

Name or handle of a model

blks

List of subsystem-level Inport blocks, root-level or subsystem-level Outport blocks or Bus Creator blocks in the specified model. If only one block needs to be specified, this argument can be the full pathname of the block. Otherwise, this argument can be either a cell array containing block pathnames or a vector of block handles.

fileName

Name of the file in which to save the bus objects created by this function. If this argument is omitted, this function does not save the created bus objects in a file. By default, the function saves the objects in a cell array.

format

Format used to store the bus objects. The value can be `'cell'` or `'object'`. Use cell array format to save the objects in a compact form.

Default: `'cell'`

structVar

Numeric structure variable used to create bus objects

Output Arguments

busInfo

A structure array containing bus information for the specified blocks. Each element of the structure array corresponds to one block and contains the following fields:

`block`	Handle of the block
`busName`	Name of the bus object associated with the block

See Also

Bus Assignment | Bus Creator | Bus Selector | Bus to Vector | `Simulink.Bus` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createMATLABStruct` | `Simulink.BusElement` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save`

How To

- "Composite Signals"

Simulink.Bus.objectToCell

Purpose

Convert bus objects to cell array containing bus information

Syntax

`busCells = Simulink.Bus.objectToCell(busNames)`

Description

`busCells = Simulink.Bus.objectToCell(busNames)` inputs a cell array of names of bus objects in the MATLAB base workspace, and returns a cell array of cell arrays in which each subordinate array contains the bus information defined by one of the bus objects. The order of the elements in the output array corresponds to the order of the names in the input array. If `busNames` is empty, the function converts all bus objects in the base workspace. The inverse function is `Simulink.Bus.cellToObject`.

Input Arguments

busNames

A cell array of names of bus objects in the MATLAB base workspace

Output Arguments

busCells

A cell array of cell arrays in which each subordinate array represents a bus object and contains the following data:

```
{BusName, HeaderFile, Description, DataScope,
Alignment, Elements}
```

The `Elements` field is an array containing the following data for each element:

```
{ElementName, Dimensions, DataType,
SampleTime, Complexity, SamplingMode, DimensionsMode,
Min, Max, DocUnits, Description}
```

See Also

Bus Assignment | Bus Creator | Bus Selector | Bus to Vector | `Simulink.Bus` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createMATLABStruct` | `Simulink.Bus.createObject` | `Simulink.BusElement` | `Simulink.Bus.save`

How To

- “Composite Signals”

Purpose

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 residing in the MATLAB base workspace) in a MATLAB file that contains a cell array of cell arrays. Each subordinate array represents a bus object and contains the following data:

```
{BusName, HeaderFile, Description, DataScope,
Alignment, Elements}
```

The *`Elements`* field is an array containing the following data for each element:

```
{ElementName, Dimensions, DataType,
SampleTime, Complexity, SamplingMode, DimensionsMode,
Min, Max,DocUnits, Description}
```

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.

`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`*.

Input Arguments

fileName

Name of the file in which to store the bus objects

format

Format used to store the bus objects. The value can be `'cell'` or `'object'`. Use cell array format to save the objects in a compact form.

Default: `'cell'`

busNames

A cell array containing names of bus objects to be saved. If the cell array is empty or omitted, this function saves all bus objects in the MATLAB workspace.

Default: {}

See Also

Bus Assignment | Bus Creator | Bus Selector | Bus to Vector | `Simulink.Bus` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createMATLABStruct` | `Simulink.Bus.createObject` | `Simulink.BusElement` | `Simulink.Bus.objectToCell`

How To

- "Composite Signals"

Purpose

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,
   'HeaderFile', FileName)
Simulink.defineIntEnumType(ClassName,
CellOfEnums, IntValues,
   'AddClassNameToEnumNames', Flag)
```

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 (string).

`Simulink.defineIntEnumType(`*`ClassName`*`, `*`CellOfEnums`*`, `*`IntValues`*`, 'DefaultValue', `*`DefValue`*`)` defines a default value for the enumeration, which is one of the strings you specify for *`CellOfEnums`*.

`Simulink.defineIntEnumType(`*`ClassName`*`, `*`CellOfEnums`*`, `*`IntValues`*`, 'HeaderFile', `*`FileName`*`)` specifies a header file into which the code generator places the enumeration class definition.

`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.

Input Arguments

ClassName

The name of the enumerated data type.

CellOfEnums

A cell array of strings 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 string that describes the enumeration data type.

'DefaultValue', DefValue

Specifies the default enumeration value.

'HeaderFile', FileName

Specifies a string naming the header file that is to contain the data type definition.

'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.

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:

```
Simulink.defineIntEnumType('BasicColors', ...
{'Red', 'Yellow', 'Blue'}, ...
[0;1;2], ...
'Description', 'Basic colors', ...
'DefaultValue', 'Blue', ...
'HeaderFile', 'mybasiccolors.h', ...
'DataScope', 'Imported', ...
'AddClassNameToEnumNames', true);
```

See Also

`enumeration`

How To

- "Import Enumerations Defined Externally to MATLAB"
- "Define Simulink Enumerations"

Simulink.findVars

Purpose

Find variables that are used or not used by models and blocks

Syntax

```
[variables] = Simulink.findVars(context)
[variables] = Simulink.findVars(context,Name,Value)
[variables] = Simulink.findVars
(context,variablefilter,Name,
   Value)
```

Description

`[variables] = Simulink.findVars(context)` returns zero or more `Simulink.VariableUsage` objects.

The search for variables includes all subsystems and referenced models in *`context`*.

`[variables] = Simulink.findVars(context,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[variables] = Simulink.findVars(context,variablefilter,Name,Value)` returns results only for variables specified by `variablefilter`.

Input Arguments

context - Search context

string | cell array

Search context, specified in one of the following ways:

- the name of a model (`'MyModel'`)
- the name or path of a block, which could possibly be masked (`'MyModel/Gain'`)
- a cell array of model or block names

The search for variables includes all subsystems and referenced models in the search context.

Example: `[variables] = Simulink.findVars('MyModel')`

variablefilter - Scope to filter results by variable

`Simulink.VariableUsage` object | array of `Simulink.VariableUsage` objects

Scope to filter search results by variable, specified as a `Simulink.VariableUsage` object or an array of `Simulink.VariableUsage` objects.

Example:

```
vars = [Simulink.VariableUsage('k','base workspace') Simulink.VariableUsa
variablefilter = Simulink.FindVars('MyModel',vars)
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FindUsedVars','true','Regexp','on','Name','^mod'` specifies that the search find used variables with names that match the pattern `^mod`.

'FindUsedVars' - Option to search for used or unused variables

`true` (default) | `false`

Option to search for used or unused variables, specified as a Boolean.

Example: `'FindUsedVars',false`

'RegExp' - Option to enable regular expression matching

`'off'` (default) | `'on'`

Option to enable regular expression matching, specified as a Boolean. Only input arguments with string values can be matched.

Example: `'RegExp','on'`

'SearchMethod' - Search mechanism

`'compiled'` (default) | `'cached'`

Search mechanism, specified as a string that can have the following values:

- `'compiled'`: Return up to date results by compiling every model within the search context prior to search.
- `'cached'`: Return quicker results by reusing results cached during the previous compile.

Example: `'SearchMethod','compiled'`

'Name' - Name of variable

string

Name of variable, specified as a string. This input argument supports regular expression matching.

Example: `'Name','^trans'`

'Users' - Names of blocks that use the variable

string

Names of blocks that use the variable, specified as a string.

Example: `'Users','MyModel/Gain1'`

'SourceType' - Type of workspace or source containing the variable

`'base'`

Type of workspace or source containing the variable, specified as a string.

Example: `'SourceType','base'`

Output Arguments

variables - Result of search

vector

Result of search, returned as a vector containing zero or more `Simulink.VariableUsage` objects.

Each element of the vector describes a workspace variable that satisfies both these conditions:

- It meets search criteria defined by the input arguments.
- It is referenced directly or indirectly within the search context.

The vector is empty if no variables match the criteria.

Examples

Find Used Variables without Compiling Model

Find variables used by `MyModel` without recompiling the model first.

```
[variables] = Simulink.findVars('MyModel');
```

Find a Specific Used Variable

Find all uses of the base workspace variable `k` by `MyModel` without recompiling the model.

```
[variables] = Simulink.findVars('MyModel','Name','k','SearchMethod','cach
```

Enable Regular Expression Matching

Find any usage of a variable whose names match a regular expression.

```
[variables] = Simulink.findVars('MyModel','Regexp','on','Name','^trans');
```

Determine Variables Common to Two Models

Given two models, discover which variables are needed by the first, the second, or both.

```
[model1Vars] = Simulink.findVars('model1');
[model2Vars] = Simulink.findVars('model2');
[commonVars] = intersect(model1vars, model2Vars);
```

Find Unused Variables

Search for all variables in the base workspace that are not being used by `MyModel`.

```
unusedVars = Simulink.findVars('MyModel','FindUsedVars',false,'SourceType
```

Determine Usage of a Specific Variable

Determine if the base workspace variable `k` is not used by `MyModel`.

```
varObj = Simulink.VariableUsage('k','base workspace');
unusedVar = Simulink.findVars('MyModel',varObj,'FindUsedVars',false);
```

Determine the Variables Used by a Block

Find the variables that are used by the `Gain1` block in `MyModel`.

```
[variables] = Simulink.findVars('MyModel','Users','MyModel/Gain1');
```

Limitations

`Simulink.findVars` does not work with the following constructs:

- MATLAB code in scripts, initialization and callback functions, etc.
- Libraries and blocks within libraries
- MATLAB Function blocks, except for input arguments
- Calls directly to MATLAB from the Stateflow action language
- S-functions that use data type variables registered via `ssRegisterDataType`.

  To make the variables findable, use `ssRegisterTypeFromNamedObject` instead.
- Variables of enumerated classes
- Variables that are used by inactive variant subsystems
- Variables referenced by machine-parented data in Stateflow

See Also

`Simulink.VariableUsage` | `find_system` | `intersect`

Related Examples

- “Search Using Model Explorer”

Simulink.ModelDataLogs.convertToDataset

Purpose

Convert logging data from `Simulink.ModelDataLogs` format to `Simulink.SimulationData.Dataset` format

Syntax

```
convertedDataset =
sourceModelDataLogsObject.convertToDataset(
   convertedDatasetName)
```

Description

Note The `ModelDataLogs` format is supported for backwards compatibility. The `ModelDataLogs` format will be removed in a future release. For an existing model that uses the `ModelDataLogs` format, you should migrate the model to use `Dataset` format. For details, see "Migrate from ModelDataLogs to Dataset Format".

For new models, use the `Dataset` logging format, which stores logged data in `Simulink.SimulationData.Dataset` objects.

`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 appear not on the MATLAB path, the conversion fails. If the model has changed, or the model's ancestors have changed, after Simulink logged the data, the conversion may 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 activeas 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

Log signal data, simulate the model, and then convert the logged dataset to the `Simulink.SimulationData.Dataset` format.

1. Open the `vdp` model by typing:

   ```
   vdp
   ```

2. Right-click the `x1` signal and select the **Signal Properties** menu item.

3. Select the **Log signal data** property.

4. Repeat steps 2 and 3 for the `x2` signal.

5. Simulate the model.

   The `vdp` model has the **Configuration Parameters > Data Import/Export > Signal logging** parameter selected, and the signal logging object name set to `logsout`. Also, the **Signal Logging Format** parameter is set to `ModelDataLogs`.

Simulink generates a warning about the use of the `ModelDataLogs` format.

6 At the MATLAB command line, type:

```
logsout

logsout =

Simulink.ModelDataLogs (vdp):
  Name                   Elements  Simulink Class

  x1                        1      Timeseries
  x2                        1      Timeseries
```

7 Convert the logged dataset to use the `Simulink.SimulationDataset` format:

```
dataset = logsout.convertToDataset('vdp_dataset')

dataset =
  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'vdp_dataset'
    Total Elements: 2

  Elements:
    1: 'x1'
    2: 'x2'

  Use getElement to access elements by index, name or block path.

  Methods, Superclasses
```

See Also

`Simulink.ModelDataLogs` | `Simulink.SimulationData.Dataset` | `Simulink.SimulationData.updateDatasetFormatLogging`

How To

- “Export Signal Data Using Signal Logging”
- “Specify the Signal Logging Data Format”

Simulink.SimState.ModelSimState.getBlockSimState

Purpose

Access SimState of individual Stateflow Chart, MATLAB Function, or S-function block

Syntax

`blockSimState = getBlockSimState(x, 'blockpath')`

Description

`blockSimState = getBlockSimState(x, 'blockpath')` returns the SimState of the block specified as *`blockpath`*. *`blockpath`* must be either a Stateflow Chart, MATLAB Function, or S-function block. For other types of blocks, see the `loggedStates` property of the Simulink.SimState.ModelSimState class.

Input Arguments

x

The x argument is a `Simulink.SimState.ModelSimState` object.

blockpath

The path to the block for which you are requesting the SimState values.

Output Arguments

blockSimState

The SimState of the block specified.

Examples

```
chartState = getBlockSimState(x, 'mymodel/chart')
```

See Also

`Simulink.SimState.ModelState.setBlockSimState`

Simulink.SimState.ModelSimState.setBlockSimState

Purpose

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`

Purpose

Save workspace variables and their values in MATLAB code format

Syntax

```
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" 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 strings. The strings name all variables (if any) that were saved to a MATLAB file.
- *`r2`* — A cell array of strings. The strings name all variables (if any) that were saved to a MAT-file.

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.

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" 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 **Simulink Preferences** pane to change the defaults for the `-maxnumel`, `-maxlevels`, `'-2dslice'`, and `-textwidth` configuration options. In the tree view section of the **Simulink Preferences** pane, select the **Variable Export Defaults** pane.

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'
```

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:
  - `fi` objects
  - `Simulink.Timeseries` and `Simulink.SimulationData.Dataset` objects
  - `Simulink.ConfigSet` objects with custom target components. (Use the `Simulink.ConfigSet` method `saveAs` instead.)

See Also

`save` | `Simulink.Bus.save` | `Simulink.ConfigSet` | `Simulink.Bus.save`

Simulink.sdi.addToRun

Purpose

Add new simulation data to existing run

Syntax

```
signalIDs =
Simulink.sdi.addToRun(runID,'base',{varName1,...,
   varNameN})
signalIDs = Simulink.sdi.addToRun(runID,'model',
   modelNameOrHandle)
signalIDs =
Simulink.sdi.addToRun(runID,'vars',var1,...,varN)
signalIDs = Simulink.sdi.addToRun(runID,'namevalue',{Name1,
   ...,NameN},{Value1,...,ValueN})
```

Description

`signalIDs = Simulink.sdi.addToRun(runID,'base',{varName1,...,varNameN})` creates a run with data from the base workspace. `{var1Name,...,var2Name}` is a cell array of strings, which are the names of variables in the base workspace.

`signalIDs = Simulink.sdi.addToRun(runID,'model',modelNameOrHandle)` creates a run with the model simulation data, as specified on the **Data Import/Export** pane of the Configuration Parameters dialog box. `modelNameOrHandle` is a string, specifying the model name, or a model handle. When calling this function, the model must be open.

`signalIDs = Simulink.sdi.addToRun(runID,'vars',var1,...,varN)` creates a run with data stored as variables, `var1,...,varN`. These variables are assumed to be in the calling workspace.

`signalIDs = Simulink.sdi.addToRun(runID,'namevalue',{Name1,...,NameN},{Value1,...,Val` creates a run from simulation data, `Value1`, and provides the ability to specify a name, `Name1`, for the data in the run.

Input Arguments

runID

Run ID, a unique number identifying a run in the Simulation Data Inspector.

Output Arguments

signalIDs

Array of numbers where each element is a unique ID for a signal added to the run.

Examples

Add simulation data from the base workspace:

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat','Str
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run')

% Add simulation output from the base workspace
Simulink.sdi.addToRun(runID,'base',{'simOut'});

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add simulation data as specified in a model:

```
sldemo_absbrake;

% Run simulation using the Simulink menu or the toolbar.
% Create a Data Inspector run
runID = Simulink.sdi.createRun('My Run');
Simulink.sdi.addToRun(runID, 'model', 'sldemo_absbrake');

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add simulation data by passing variables directly to `Simulink.sdi.addToRun`:

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat','Struct
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run');
Simulink.sdi.addToRun(runID,'vars',simOut);

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add simulation data and name the data:

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat','Struct
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run');
% Name simulation output passed to Simulink.sdi.addToRun
Simulink.sdi.addToRun(runID, 'namevalue', {'MyData'}, {simOut});

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.view`

How To

- “Record and Inspect Signal Data Programmatically”

Purpose

Clear all data from Simulation Data Inspector

Syntax

`Simulink.sdi.clear`

Description

`Simulink.sdi.clear` clears all run data from the Simulation Data Inspector.

Examples

Remove all runs from the Simulation Data Inspector.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat','Str

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

Simulink.sdi.clear;

% The number of runs is now zero.
runCount = Simulink.sdi.getRunCount()
```

Simulink.sdi.compareRuns

Purpose

Compare signal data between two simulation runs

Syntax

```
diff = Simulink.sdi.compareRuns(runID1,runID2)
diff = Simulink.sdi.compareRuns(runID1,runID2,
   alignmentMethods)
```

Description

`diff = Simulink.sdi.compareRuns(runID1,runID2)` compares the matched signals between two simulation runs and returns their differences in a `Simulink.sdi.DiffRunResult` object.

`diff = Simulink.sdi.compareRuns(runID1,runID2,alignmentMethods)` compares the matched signals between two simulation runs using specified alignment algorithms and returns their differences in a `Simulink.sdi.DiffRunResult` object.

Input Arguments

runID1

Run ID, a unique number identifying the first run for comparison.

runID2

Run ID, a unique number identifying the second run for comparison.

alignmentMethods

An array specifying three alignment algorithms. Data is aligned by the value of the first element of the array, then by the second element, and then by the third element. Only the first three values in the array are considered. The array takes the following values.

Value	Align By
`Simulink.sdi.AlignType.BlockPath`	Path to the source block for the signal
`Simulink.sdi.AlignType.DataSource`	Data name (`logsout.Stick.Data`)
`Simulink.sdi.AlignType.SID`	"Simulink Identifier" on page 7-2
`Simulink.sdi.AlignType.SignalName`	Signal name for the data (`Stick`)

Output Arguments

diff

Instance of `Simulink.sdi.DiffRunResult` that describes the differences between two simulation runs.

Examples

Compare two simulation runs using the specified alignment algorithms. The function `Simulink.sdi.compareRuns` returns a `Simulink.sdi.DiffRunResult` object containing the comparison results.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
              'SaveFormat', 'StructureWithTime', ...
              'ReturnWorkspaceOutputs', 'on');

% Create a run and get signal IDs
run1ID = Simulink.sdi.createRun('My Run', 'namevalue',...
                                 {'simOut'}, {simOut});

% Change one of the parameters of the model
Mq = 3*Mq;

% Simulate again
simOut = sim('slexAircraftExample', 'SaveOutput','on',          ...
              'SaveFormat', 'StructureWithTime', ...
              'ReturnWorkspaceOutputs', 'on');

% Create another run and get signal IDs
run2ID = Simulink.sdi.createRun('New Run', 'namevalue',...
                                 {'simOut'}, {simOut});

% Define the alignment algorithms for comparison.
% Align the data first by data name, then by block path, then by SID.
algorithms = [Simulink.sdi.AlignType.DataSource
               Simulink.sdi.AlignType.BlockPath
               Simulink.sdi.AlignType.SID];
```

```
% Compare the two runs
difference = Simulink.sdi.compareRuns(run1ID, run2ID, algorithms);

% Number of comparisons in result
numComparisons = difference.count;

% Iterate through each result element
for i = 1:numComparisons
    % Get result at index i
    resultAtIdx = difference.getResultByIndex(i);

    % Get signal IDs for each comparison result
    sig1 = resultAtIdx.signalID1;
    sig2 = resultAtIdx.signalID2;

    % Display if signals match or not
    displayStr = 'Signals with IDs %d and %d %s \n';
    if resultAtIdx.match
       fprintf(displayStr, sig1, sig2, 'match');
    else
       fprintf(displayStr, sig1, sig2, 'do not match');
    end

    % Plot tolerance and difference results in a figure
    f1 = figure;
    plot(resultAtIdx.tol, 'Color', 'r');
    hold on;
    plot(resultAtIdx.diff, 'Color', 'g');
    legend('Tolerance', 'Difference');
end
```

Compare normal mode simulation with code generation results for model `slexAircraftExample`.

```
% Load the model 'slexAircraftExample'
load_system('slexAircraftExample');
```

```
% Configure model "slexAircraftExample" for logging
set_param('slexAircraftExample', 'SolverType', 'Fixed-Step', 'SaveOutp
          'SaveFormat', 'StructureWithTime', 'ReturnWorkspaceOutputs',

% CD to temporary directory and build
cd(tempdir);
rtwbuild('slexAircraftExample');

% Run the executable
if ispc
   system('slexAircraftExample');
elseif unix
   system('./slexAircraftExample');
end

% Create a run using the slexAircraftExample.mat placed in the current
[run1ID, ~, ~] = Simulink.sdi.createRun('My Run','file','slexAircraftE

% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on',...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on',...
             'SolverType', 'Fixed-Step');

% Create another run from the simulation
[run2ID, ~, ~] = Simulink.sdi.createRun('My Run', 'namevalue',...
                                        {'MyData'}, {simOut});

% Compare the two runs
difference = Simulink.sdi.compareRuns(run1ID, run2ID);

% Number of comparisons in result
numComparisons = difference.count;

% Iterate through each result element
for i = 1:numComparisons
```

```
   % Get result at index i
   resultAtIdx = difference.getResultByIndex(i);

   % Get signal IDs for each comparison result
   sig1 = resultAtIdx.signalID1;
   sig2 = resultAtIdx.signalID2;

   % Display if signals match or not
   displayStr = 'Signals with IDs %d and %d %s \n';
   if resultAtIdx.match
      fprintf(displayStr, sig1, sig2, 'match.');
   else
      fprintf(displayStr, sig1, sig2, 'do not match.');
end

% Plot tolerance and difference results in a figure
f1 = figure;
plot(resultAtIdx.tol, 'Color', 'r');
hold on;
plot(resultAtIdx.diff, 'Color', 'g');
legend('Tolerance', 'Difference');
end
```

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.DiffRunResult`

How To

- "Record and Inspect Signal Data Programmatically"

Purpose

Compare data from two signals

Syntax

`diff = Simulink.sdi.compareSignals(signalID1,signalID2)`

Description

`diff = Simulink.sdi.compareSignals(signalID1,signalID2)` compares two signals and returns the results in a `Simulink.sdi.DiffSignalsResult` object.

Input Arguments

signalID1

Signal ID, a unique number identifying the first signal for comparison.

signalID2

Signal ID, a unique number identifying the second signal for comparison.

Output Arguments

diff

`Simulink.sdi.diffSignalsResult` object containing the results of the comparison.

Examples

Call `Simulink.sdi.createRun` to get signal IDs for a simulation run in the Simulation Data Inspector. The function `Simulink.sdi.compareSignals` returns a `Simulink.sdi.DiffSignalResult` object containing the result data of the comparison. From this object you can determine if the signals are different.

Comparing two signals:

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run and get signal IDs
[~, ~, signalIDs] = Simulink.sdi.createRun('My Run', 'namevalue', {'My
sig1 = signalIDs(1);
sig2 = signalIDs(2);

% Compare two signals, which returns the results in an instance of Simuli
diff = Simulink.sdi.compareSignals(sig1, sig2);

% Find if the signal data match
match = diff.match;

% Get the tolerance used in Simulink.sdi.compareSignals
tolerance = diff.tol;
```

Comparing signals from two different runs:

```
% Load the model 'slexAircraftExample'
load_system('slexAircraftExample');

% Configure model "slexAircraftExample" for logging
set_param('slexAircraftExample', 'SolverType', 'Fixed-Step', 'SaveOutput'
          'SaveFormat', 'StructureWithTime', 'ReturnWorkspaceOutputs',...
           'on');

% CD to temporary directory and build
cd(tempdir);
rtwbuild('slexAircraftExample');

% Run the executable
if ispc
   system('slexAircraftExample');
elseif unix
   system('./slexAircraftExample');
end

% Create a Data Inspector run using slexAircraftExample.mat created in th
% directory
```

```
[~, ~, signalIDs] = Simulink.sdi.createRun('My Run', 'file', 'slexAirc

% Get first signal id to compare
sig1 = signalIDs(1);

% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on',          ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a Data Inspector run and get signal IDs
[~, ~, signalIDs] = Simulink.sdi.createRun('My Run', 'namevalue',...
                                            {'MyData'}, {simOut});

% Get second signal id to compare
sig2 = signalIDs(1);

% compare two signals
result = Simulink.sdi.compareSignals(sig1, sig2);
if result.match
   disp('****The signals match****');
else
   disp('****The signals did not match****');
end

% Plot results in a figure
plot(result.tol, 'Color', 'r');
hold on;
plot(result.diff, 'Color', 'g');
legend('Tolerance', 'Difference');
```

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.DiffSignalResult`

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.copyRun

Purpose

Create copy of run including simulation output data

Syntax

```
runIDcopy = Simulink.sdi.copyRun(runID)
[runIDcopy, runIndex] = Simulink.sdi.copyRun(runID)
[runIDcopy, runIndex,
   signalIDs] = Simulink.sdi.copyRun(runID)
```

Description

`runIDcopy = Simulink.sdi.copyRun(runID)` copies the run associated with `runID` and returns a run ID, `runIDcopy`, associated with the new run. The new run contains all of the simulation output data and metadata from the original run.

`[runIDcopy, runIndex] = Simulink.sdi.copyRun(runID)` copies the run associated with `runID` and returns the run ID, `runIDcopy`, and the `runIndex` for the new run.

`[runIDcopy, runIndex, signalIDs] = Simulink.sdi.copyRun(runID)` copies the run associated with `runID` and returns the run ID, run index, and array of new signal IDs, `signalIDs`, for signals in the new run.

Input Arguments

runID

Run ID, a unique number identifying a run in the Simulation Data Inspector.

Output Arguments

runIDcopy

Run ID, a unique number identifying the copied run.

runIndex

Number representing the new run's index to the list of runs currently in the Simulation Data Inspector.

signalIDs

Vector of numbers, where each element is a unique ID for a signal in this run. The signal IDs are different in the new run.

Examples

Copy a `Simulink.sdi.Run` object representing a run currently in the Simulation Data Inspector.

```
% Configure model 'slexAircraftExample' for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

[newRunID, runIndex, signalIDs] = Simulink.sdi.copyRun(runID);

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.view`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.sdi.createRun

Purpose Create run in Simulation Data Inspector

Syntax

```
runID = Simulink.sdi.createRun()
runID = Simulink.sdi.createRun(runName)
runID =
Simulink.sdi.createRun(runName,'base',{varName1,...,
   varNameN})
runID = Simulink.sdi.createRun(runName,'model',
   modelNameOrHandle)
runID = Simulink.sdi.createRun(runName, 'vars', var1,...,
   varN)
runID =
Simulink.sdi.createRun(runName,'namevalue',{varName1,
   ...,varNameN},{varValue1,...,varValueN})
runID = Simulink.sdi.createRun(runName,'file',fileName)
[runID,runIndex] = Simulink.sdi.createRun(...)
[runID,runIndex,signalIDs] = Simulink.sdi.createRun(...)
```

Description

`runID = Simulink.sdi.createRun()` creates an empty unnamed run in the Simulation Data Inspector and returns the corresponding run ID.

`runID = Simulink.sdi.createRun(runName)` creates an empty run named `runName` in the Simulation Data Inspector repository and returns the corresponding run ID.

`runID = Simulink.sdi.createRun(runName,'base',{varName1,...,varNameN})` creates a run with data from the base workspace. `var1Name` to `var2Name` are names of variables in the base workspace.

`runID = Simulink.sdi.createRun(runName,'model',modelNameOrHandle)` creates a run with the model simulation output data, as specified on the **Data Import/Export** pane of the Configuration Parameters dialog box. `modelNameOrHandle` is a string specifying the model name of a model handle. When calling this function, the model must be open.

`runID = Simulink.sdi.createRun(runName, 'vars', var1,...,varN)` creates a run with data stored in variables, `var1,...,varN`. These variables must be in the calling workspace.

`runID = Simulink.sdi.createRun(runName,'namevalue',{varName1,...,varNameN},{va` creates a run from simulation data. `varValue1` contains the data and `varName1` specifies a name for the data.

`runID = Simulink.sdi.createRun(runName,'file',fileName)` creates a run with data extracted from a MAT-file, `fileName`.

`[runID,runIndex] = Simulink.sdi.createRun(...)` creates a run according to the input arguments specification and returns the run ID and the run index to the run currently stored in the Simulation Data Inspector.

`[runID,runIndex,signalIDs] = Simulink.sdi.createRun(...)` creates a run according to the input arguments specification and returns the run ID, the run index, and the signal IDs. The signal IDs are stored in a vector of numbers, where each element is a unique signal ID for each signal in the new run.

Tips

- Before calling `Simulink.sdi.createRun` with either `` `base' `` or `` `model' `` as an input argument, you must configure the model for logging and simulate the model.
- When you create and add a run, the Simulation Data Inspector maintains a list of these runs. The first run in the list is given a `runIndex` of 1. If you delete a run from the Simulation Data Inspector, the subsequent runs move up the list and each `runIndex` changes. However, the run IDs remain the same.

Input Arguments

runName

A string specifying a name for the run.

Output Arguments

runID

Run ID, a unique number identifying a run in the Simulation Data Inspector.

runIndex

Number representing an index to the list of runs currently in the Simulation Data Inspector.

signalIDs

Vector of numbers, where each element is a unique ID for a signal in a run.

Examples

Empty Run With No Name

Create an empty run in the Simulation Data Inspector.

```
runID = Simulink.sdi.createRun();
```

Empty Run With a Name

Create an empty run in the Simulation Data Inspector and give it a name.

```
runID = Simulink.sdi.createRun('My Run');
```

Simulation Output Data in the Base Workspace

Create a run in the Simulation Data Inspector from output data in the base workspace.

Configure the model, `slexAircraftExample`, for logging and simulate.

```
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');
```

Create a Simulation Data Inspector run from the simulation output data in the base workspace.

```
Simulink.sdi.createRun('My Run','base',{'simOut'});
```

Open the Simulation Data Inspector tool to view the data.

```
Simulink.sdi.view;
```

Simulation Output Data for a Model

Create a run from simulation output data as specified in the model. The model must be open to use this function signature.

Open the model sldemo_absbrake.

```
sldemo_absbrake;
```

Run the simulation using the model diagram menu or the toolbar. The model is already configured for signal logging.

Create a Simulation Data Inspector run named, `My Run`, using simulation output data from the model.

```
Simulink.sdi.createRun('My Run','model','sldemo_absbrake');
```

Open the Simulation Data Inspector tool to view the data.

```
Simulink.sdi.view;
```

A run named, `My Run`, appears in the Simulation Data Inspector.

Passed Variables

Create a run in the Simulation Data Inspector by directly passing data through variables.

Configure model `slexAircraftExample` for logging and simulate.

```
simOut = sim('slexAircraftExample','SaveOutput','on', ...
              'SaveFormat','StructureWithTime', ...
              'ReturnWorkspaceOutputs','on');
```

Create a Simulation Data Inspector run named, `My Run`, by passing the simulation output variable, `simOut`.

```
Simulink.sdi.createRun('My Run','vars',simOut);
```

Open the Simulation Data Inspector tool to view the data.

```
Simulink.sdi.view;
```

A run named, `My Run`, appears in the Simulation Data Inspector.

Include Name of Simulation Data in the Run

Create a Simulation Data Inspector run and provide a name for the passed data.

Configure model `slexAircraftExample` for logging and simulate.

```
simOut = sim('slexAircraftExample','SaveOutput','on', ...
              'SaveFormat','StructureWithTime', ...
              'ReturnWorkspaceOutputs','on');
```

Create a Simulation Data Inspector run named, `My Run`, by passing the simulation output variable, `simOut` and a name for the data, `MyData`.

```
Simulink.sdi.createRun('My Run', 'namevalue', {'MyData'}, {simOut});
```

Open the Simulation Data Inspector tool to view the data.

```
Simulink.sdi.view;
```

A run named, `My Run`, appears in the Simulation Data Inspector.

Logged MAT-file of a Code Generation Build (Requires Simulink Coder)

Create a run in the Simulation Data Inspector from data in a MAT-file, which includes data from a code generation build.

Load the model `slexAircraftExample`.

```
load_system('slexAircraftExample');
```

Configure the model for logging and simulate.

```
set_param('slexAircraftExample','SolverType','Fixed-Step',...
          'SaveOutput','on',...
          'SaveFormat','StructureWithTime', ...
          'ReturnWorkspaceOutputs','on');
```

Build the model to a temporary directory.

```
cd(tempdir);
rtwbuild('slexAircraftExample');
```

Run the executable.

```
if ispc
   system('slexAircraftExample');
elseif unix
   system('./slexAircraftExample');
end
```

A MAT-file is generated in the current directory.

Create a Simulation Data Inspector run using the data in the MAT-file.

```
Simulink.sdi.createRun('My Run','file','slexAircraftExample.mat');
```

Open the Simulation Data Inspector tool to view the data.

```
Simulink.sdi.view;
```

A run named, `My Run`, appears in the Simulation Data Inspector.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.getRun` | `Simulink.sdi.deleteRun`

How To

- “Record and Inspect Signal Data Programmatically”

Purpose

Delete run from Simulation Data Inspector

Syntax

`Simulink.sdi.deleteRun(runID)`

Description

`Simulink.sdi.deleteRun(runID)` deletes a run associated with the run ID in the Simulation Data Inspector. Deleting the run removes all signal data included in the run. The Simulation Data Inspector maintains a list of runs. After deleting a run, the subsequent runs move up the list and the run index for each run changes. However, the run IDs remain the same.

Input Arguments

runID

Run ID, a unique number identifying a run.

Examples

Remove a run from the Simulation Data Inspector.

```
% Configure model 'slexAircraftExample' for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat','Str

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

Simulink.sdi.deleteRun(runID);
```

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.copyRun` | `Simulink.sdi.clear`

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.getRun

Purpose Return `Simulink.sdi.Run` object containing simulation output data

Syntax

```
runObj = Simulink.sdi.getRun(runID)
```

Description `runObj = Simulink.sdi.getRun(runID)` returns a handle to the `Simulink.sdi.Run` object for the run corresponding to `runID` in the Simulation Data Inspector.

Input Arguments

runID

Run ID, a unique number identifying a run in the Simulation Data Inspector.

Output Arguments

runObj

A handle to the `Simulink.sdi.Run` object containing the signal data and metadata.

Examples Get the `Simulink.sdi.Run` object for a run in the Simulation Data Inspector.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
[runID, runIndex, signalIDs] = Simulink.sdi.createRun('My Run','base',{'s

runObj = Simulink.sdi.getRun(runID);
```

See Also `Simulink.sdi.Run` | `Simulink.sdi.createRun`

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.getRunCount

Purpose

Return number of runs in Simulation Data Inspector

Syntax

```
runCount = Simulink.sdi.getRunCount()
```

Description

`runCount = Simulink.sdi.getRunCount()` returns the number of runs that are in the Simulation Data Inspector.

Output Arguments

runCount

Number of runs that exist in the Simulation Data Inspector.

Examples

Call `Simulink.sdi.getRunCount` to get the number of runs currently in the Simulation Data Inspector.

```
runCount = Simulink.sdi.getRunCount();
```

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun`

Simulink.sdi.getRunIDByIndex

Purpose

Return the run ID corresponding to run index

Syntax

`runID = Simulink.sdi.getRunIDByIndex(runIndex)`

Description

`runID = Simulink.sdi.getRunIDByIndex(runIndex)` returns the run ID for the run corresponding to the run index.

Input Arguments

runIndex

Number representing an index to the list of runs currently in the Simulation Data Inspector.

Output Arguments

runID

Run ID, a unique number identifying a run in the Simulation Data Inspector.

Examples

Get the run IDs for all runs currently in the Simulation Data Inspector.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
[runID, runIndex, signalIDs] = Simulink.sdi.createRun('Run1','base',{'sim

% Get the ID of the previously created run by index
runID2 = Simulink.sdi.getRunIDByIndex(runIndex);

% Both runID and runID2 reference the same run and should be equal
isequal(runID, runID2)
```

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRunCount` | `Simulink.sdi.isValidRunID`

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.getSignal

Purpose

Return `Simulink.sdi.Signal` object for signal in Simulation Data Inspector

Syntax

```
signal = Simulink.sdi.getSignal(signalID)
```

Description

`signal = Simulink.sdi.getSignal(signalID)` returns the `Simulink.sdi.Signal` object for the signal corresponding to the signal ID. The `Simulink.sdi.Signal` object manages the signal's time series data and metadata.

Input Arguments

signalID

Signal ID, a unique number identifying a signal in the Simulation Data Inspector.

Output Arguments

signal

A handle to the `Simulink.sdi.Signal` object.

Examples

Get the `Simulink.sdi.Signal` object for a signal in the Simulation Data Inspector. With the signal object you can modify its comparison and visualization properties.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat','St

% Create a Simulation Data Inspector run
[runID, runIndex, signalIDs] = Simulink.sdi.createRun('My Run','base'

signalObj = Simulink.sdi.getSignal(signalIDs(1));

% Specify the comparison and visualization signal properties
signalObj.absTol = .5;
signalObj.syncMethod = 'intersection';
signalObj.interpMethod = 'linear';
signalObj.lineColor = [1,0.4,0.6];
signalObj.lineDashed = '-';
signalObj.checked = true;

% view the signals in Simulation Data Inspector GUI
Simulink.sdi.view
```

See Also

`Simulink.sdi.Signal` | `Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.getRun`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.sdi.getSource

Purpose

Return location for storing simulation data

Syntax

`source = Simulink.sdi.getSource()`

Description

`source = Simulink.sdi.getSource()` return the location of the Simulation Data Inspector repository for storing simulation data. Call this function before running multiple simulations in a `parfor` loop.

Output Arguments

source

Location of the Simulation Data Inspector repository.

Examples

Record Data During Parallel Simulations

This example shows how to run multiple simulations in a `parfor` loop and record each run in the Simulation Data Inspector tool.

Open the Simulation Data Inspector.

```
Simulink.sdi.view;
```

Load the model.

```
mdl = 'slexAircraftExample';
load_system(mdl);
```

Get the location of the simulation data repository.

```
src = Simulink.sdi.getSource();
```

Open MATLAB pool with 4 workers.

```
matlabpool(4);
```

Run the simulation in a `parfor` loop.

```
parfor i=1:4
   % Set the location of the simulation data repository of this
   % worker to be the same for aggregating the data
   Simulink.sdi.setSource(src);
   % Run the simulation
   simOut = sim(mdl,'SaveOutput','on',...
               'SaveFormat','StructureWithTime',...
               'ReturnWorkspaceOutputs','on');
   % Create a simulation run in the Simulink Data Inspector
   Simulink.sdi.createRun(['Run' num2str(i)],'namevalue',...
                          {'simout'},{simOut});
end
```

Close the MATLAB pool and all of the models.

```
matlabpool close;
bdclose all;
```

Refresh the Simulation Data Inspector

```
Simulink.sdi.refresh();
```

See Also

`Simulink.sdi.setSource` | `Simulink.sdi.refresh`

Purpose

Determine if run ID is valid

Syntax

`isValid = Simulink.sdi.isValidRunID(runID)`

Description

`isValid = Simulink.sdi.isValidRunID(runID)` returns `true` if the run ID, `runID`, corresponds to a run currently in the Simulation Data Inspector. Otherwise, it returns `false`.

Input Arguments

runID

Run ID, a unique number identifying a run in the Simulation Data Inspector.

Output Arguments

isValid

Boolean value: `true`, if the run exists; `false`, otherwise.

Examples

Before comparing the simulation data of two runs, you can verify that the run IDs are valid.

```
 % Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a Data Inspector run
run1ID = Simulink.sdi.createRun('First Run', 'base', {'simOut'});
run2ID = Simulink.sdi.createRun('Second Run', 'base', {'simOut'});

% Check if run IDs are valid in Simulation Data Inspector
run1ID_valid = Simulink.sdi.isValidRunID(run1ID);
run2ID_valid = Simulink.sdi.isValidRunID(run2ID);

if run1ID_valid & run2ID_valid
% Compare two runs
   difference = Simulink.sdi.compareRuns(run1ID, run2ID);
end
```

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.compareRuns`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.sdi.load

Purpose

Load saved Simulation Data Inspector session

Syntax

`Simulink.sdi.load(fileName)`

Description

`isValidSDIMatFile` = `Simulink.sdi.load(fileName)` loads the runs, signals, tolerances, signal selections, and the saved view from a MAT-file, `fileName`. You can include a full or partial path in `fileName`.

Input Arguments

fileName

A string specifying the file and optionally a full or partial path.

Output Arguments

isValidSDIMatFile

Boolean value specifying whether the file is a valid Simulation Data Inspector MAT-file.

Examples

Load a previous Simulation Data Inspector session saved to a MAT-file.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a run in the Simulation Data Inspector
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Save the current Simulation Data Inspector session
Simulink.sdi.save('my_runs.mat');

% Clear all data from the Simulation Data Inspector
Simulink.sdi.clear;

% Import saved MAT-file into the Simulation Data Inspector
Simulink.sdi.load('my_runs.mat');
```

```
% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Alternatives

In the Simulation Data Inspector tool, to load a saved session, select **File > Import Data** and enter the name for the MAT-file.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.save` | `Simulink.sdi.view`

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.refresh

Purpose Refresh Simulation Data Inspector

Syntax `Simulink.sdi.refresh`

Description `Simulink.sdi.refresh` refresh the Simulation Data Inspector repository and the tool.

Examples

Record Data During Parallel Simulations

This example shows how to run multiple simulations in a `parfor` loop and record each run in the Simulation Data Inspector tool.

Open the Simulation Data Inspector.

```
Simulink.sdi.view;
```

Load the model.

```
mdl = 'slexAircraftExample';
load_system(mdl);
```

Get the location of the simulation data repository.

```
src = Simulink.sdi.getSource();
```

Open MATLAB pool with 4 workers.

```
matlabpool(4);
```

Run the simulation in a `parfor` loop.

```
parfor i=1:4
   % Set the location of the simulation data repository of this
   % worker to be the same for aggregating the data
   Simulink.sdi.setSource(src);
   % Run the simulation
   simOut = sim(mdl,'SaveOutput','on',...
                'SaveFormat','StructureWithTime',...
                'ReturnWorkspaceOutputs','on');
   % Create a simulation run in the Simulink Data Inspector
   Simulink.sdi.createRun(['Run' num2str(i)],'namevalue',...
                          {'simout'},{simOut});
end
```

Close the MATLAB pool and all of the models.

```
matlabpool close;
bdclose all;
```

Refresh the Simulation Data Inspector

```
Simulink.sdi.refresh();
```

See Also

`Simulink.sdi.getSource` | `Simulink.sdi.setSource`

Purpose

Generate report from Simulation Data Inspector tool

Syntax

```
Simulink.sdi.report()
Simulink.sdi.report(Name,Value)
```

Description

`Simulink.sdi.report()` creates a report of the current view and data in the Inspect Signals tab in the Simulation Data Inspector.

`Simulink.sdi.report(Name,Value)` creates a report using the parameter `Name, Value` pairs, which specifies the information to include in the report.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'ReportToCreate'

Specify which Simulation Data Inspector tab view to include in the report

Value	Description
`'Inspect Signals'` (default)	Information in the Inspect Signals tab view
`'Compare Signals'`	Information in the Compare Signals tab view
`'Compare Runs'`	Information in the Compare Runs tab view

Default: `'Inspect Signals'`

'ReportOutputFolder'

MATLAB string specifying the location of the generated report.

Default: <current working folder>/slprj/sdi

'ReportOutputFile'

MATLAB string specifying the report file name.

Default: `'SDI_report.html'`

'PreventOverwritingFile'

Boolean value. If the report file exists and the value is true, then the report generator increments the file name. If false, the report generator overwrites the report file, if it exists.

Default: `true`

'ColumnsToReport'

Matrix specifying values from the enumeration class, `Simulink.sdi.SignalMetaData`, which lists all signal metadata available in the Simulation Data Inspector. For example, to include the **Run** and **Synchronization Method** columns in the Compare Signals report, create a variable `signal_metadata`:

```
signal_metadata = [Simulink.sdi.SignalMetaData.Run, Simulink.sdi.SignalMe
```

Table Columns available for Inspect Signals and Compare Signals tabs:

Enumeration Element	Table Column Description
`BlockPath` (default)	Block path
`SignalName` (default)	Signal Name
`Line` (default)	Line style
`AbsTol` (default)	Positive number (user-specified)
`RelTol` (default)	Positive number (user-specified)

Enumeration Element	Table Column Description
`SyncMethod`	Method to align time vector: **union**, **intersection**, **uniform** (user-specified)
`DataSource`	String signifying the source of data (`logsout.Stick.Data`)
`TimeSeriesRoot`	String signifying the name of the Simulink.Timeseries object (`logsout.Stick.Time`)
`TimeSource`	String signifying the array containing the time data (`logsout.Stick.Time`)
`InterpMethod`	Method to align data: **zoh**, **linear** (user-specified)
`Port`	Index of the output port that emits the signal logged
`Dimensions`	Number of dimensions of the signal
`Channel`	Channel of matrix data
`Run`	Name of a simulation run
`Model`	Model name for the signal data
`SID`	"Simulink Identifier" on page 7-2 • **Run 1: SID** • **Run 2: SID**

Table columns available for the Compare Run tab:

Value	Description
`Result` (default)	Result of the comparison for the signal across the specified runs
`AbsTol1` (default)	Absolute tolerance

Value	Description
`RelTol1` (default)	Relative tolerance
`AlignedBy` (default)	Metadata used to align signal data between simulation runs
`LinkToPlot` (default)	Link to a plot of each comparison result
`BlockPath1`	Block path for signal from first run
`BlockPath2`	Block path for signal from second run
`SignalName1`	Signal name from first run
`SignalName2`	Signal name from second run
`DataSource1`	Name for the data from first run
`DataSource2`	Name for the data from second run
`SID1`	"Simulink Identifier" on page 7-2 for the data from first run
`SID2`	"Simulink Identifier" on page 7-2 for the data from second run
`SyncMethod1`	Synchronization method specified for the first run
`InterpMethod1`	Interpolation method specified for the first run
`Channel1`	Channel specified for the first run

'ShortenBlockPath'

Boolean value. If the value is `true` and the block path name is too long, the Simulation Data Inspector shortens the name in the report. If the value is `false`, the entire block path name is displayed in the report.

Default: `true`

'LaunchReport'

Boolean value. If the value is `true` after creation, the generated report opens.

Default: `true`

'SignalsToReport'

For the `'Compare Runs'` view only.

Value	Description
`'ReportOnlyMismatchedSignals'`	Includes only the mismatched signals from comparing two runs
`'ReportAllSignals'`	Includes all signals in the two compared runs

Default: `'ReportOnlyMismatchedSignals'`

Examples

Create a report from the Inspect Signals view

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a Data Inspector run
[~, ~, signalIDs] = Simulink.sdi.createRun('My Run', 'base', {'simOut

% Select signals for plotting
for i = 1:length( signalIDs )
   signal = Simulink.sdi.getSignal(signalIDs(i));
   signal.checked = true;
end
```

```
% Create default report, which is the Inspect Signals view
Simulink.sdi.report();
```

Create a report from the Compare Signals view

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a Data Inspector run
[~, ~, signalIDs] = Simulink.sdi.createRun('My Run', 'base', {'simOut'});

% Compare two signals
Simulink.sdi.compareSignals( signalIDs(1), signalIDs(2) );

% Report on the signal comparison
Simulink.sdi.report('ReportToCreate', 'Compare Signals', ...
                    'ReportOutputFolder', tempdir, 'ReportOutputFile', 'm
```

Create a report from the Compare Runs view

```
% Configure model "slexAircraftExample" for logging and simulate
set_param('slexAircraftExample/Pilot','WaveForm','square');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
                  'SaveFormat', 'StructureWithTime', ...
                  'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run, Simulink.sdi.Run, from simOut i
runID1 = Simulink.sdi.createRun('First Run','namevalue',{'simOut'},{simOu

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create another Simulation Data Inspector run
runID2 = Simulink.sdi.createRun('Second Run','namevalue',{'simOut'},{s

% Compare two runs
difference = Simulink.sdi.compareRuns(runID1, runID2);

% Specify columns to include in the report
metaDataOfInterest = [Simulink.sdi.SignalMetaData.Result, ...
                    Simulink.sdi.SignalMetaData.BlockPath1, ...
                    Simulink.sdi.SignalMetaData.RelTol1];
% Report on the run comparison
Simulink.sdi.report('ReportToCreate', 'Compare Runs', ...
                    'ColumnsToReport', metaDataOfInterest, ...
                    'SignalsToReport', 'ReportAllSignals' );
```

Alternatives

In the Simulation Data Inspector tool, to generate a report, select **File > Generate Report**.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.compareRuns` | `Simulink.sdi.compareSignals`

How To

- "Create Simulation Data Inspector Report"

Simulink.sdi.save

Purpose

Save current Simulation Data Inspector session

Syntax

`Simulink.sdi.save(fileName)`

Description

`Simulink.sdi.save(fileName)` saves all runs, signals, tolerances, signal selections, and the current view to a MAT-file, `fileName`.

Input Arguments

fileName

A string specifying the target file to save.

Examples

Save the Simulation Data Inspector simulation runs, specified tolerances, and current view to a MAT-file. You can later load the information back in to the Simulation Data Inspector using the `Simulink.sdi.load` function.

```
% Configure model "sldemo_fuelsys" for logging and simulate
simOut = sim('sldemo_fuelsys', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');
% Create a run in the Simulation Data Inspector
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Save the current Simulation Data Inspector session
Simulink.sdi.save('my_runs.mat');
```

Alternatives

To save the session to a file, in the Simulation Data Inspector tool, on the menu bar, select **File > Save** and enter a name for the MAT-file.

See Also

`Simulink.sdi.createRun`

How To

- "Record and Inspect Signal Data Programmatically"

Purpose

Specify a run naming rule

Syntax

```
Simulink.sdi.setRunNamingRule(rule)
```

Description

`Simulink.sdi.setRunNamingRule(rule)` specifies a template for naming a run in the Simulation Data Inspector.

Input Arguments

rule

A string using predefined tokens and regular characters to create a template for run names. The available tokens are

Token	Description
`<run_index>`	Sequential number of each run
`<model_name>`	Name of model
`<time_stamp>`	Run creation time
`<sim_mode>`	Simulation mode for recorded run

Examples

Specify a run naming rule.

```
Simulink.sdi.setRunNamingRule(...
   'Run <run_index> : <model_name> : <sim_mode>');
```

After recording a run, in the Signal Browser table of the Simulation Data Inspector, the run appears with a name similar to the following:

```
Run 1 :  slexAircraftExample :  normal
```

Alternatives

In the Simulation Data Inspector, in the Run Configuration Options dialog box, enter a string in the **Run naming rule** box.

How To

- “Run Management Configuration”

Simulink.sdi.setRunOverwrite

Purpose

Mark simulation run for overwrite

Syntax

`Simulink.sdi.setOverwriteRun(runID,overwrite)`

Description

`Simulink.sdi.setOverwriteRun(runID,overwrite)` marks a run identified by `runID` for overwriting in the Simulation Data Inspector with the next recorded simulation run.

Input Arguments

runID

Run ID, a unique number identifying a run in the Simulation Data Inspector.

Output Arguments

overwrite

Boolean value. When set to `true`, the next recorded simulation overwrites the run.

Examples

This example shows how to mark a run for overwriting in the Simulation Data Inspector.

1

Open `sldemo_fuelsys`.

2

On the Simulink Editor toolbar, click the **Record** button on.

3

Simulate the model.

4

Open the Simulation Data Inspector.

5

In the Signal Browser table, right-click the run name and select `Properties`.

6

At the MATLAB command line, create a `runID` variable that uses the value of the **Run ID** for a run. You can find **Run ID** in the Properties dialog box in the Simulation Data Inspector.

```
runID = 1;
```

7

Set the overwrite condition to `true`.

```
Simulink.sdi.setRunOverwrite(runID,true);
```

In the Simulation Data Inspector, you can see the run is now marked for overwriting.

8

Simulate `sldemo_fuelsys`.

In the Simulation Data Inspector, the new recorded data replaces the previous recorded run.

Alternatives

In the Simulation Data Inspector tool, click a run in the Signal Browser table and then click the **Overwrite Run** button.

How To

- “Run Management Configuration”

Simulink.sdi.setSource

Purpose Set location for storing simulation data

Syntax

```
Simulink.sdi.setSource(source)
```

Description `Simulink.sdi.setSource(source)` sets the location of the Simulation Data Inspector repository for storing simulation data. Before calling this function, call `Simulink.sdi.getSource` to get the location. Call this function in a `parfor` loop for recording multiple simulations in the Simulation Data Inspector repository.

Output Arguments

source

Location of the Simulation Data Inspector repository.

Examples

Record Data During Parallel Simulations

This example shows how to run multiple simulations in a `parfor` loop and record each run in the Simulation Data Inspector tool.

Open the Simulation Data Inspector.

```
Simulink.sdi.view;
```

Load the model.

```
mdl = 'slexAircraftExample';
load_system(mdl);
```

Get the location of the simulation data repository.

```
src = Simulink.sdi.getSource();
```

Open MATLAB pool with 4 workers.

```
matlabpool(4);
```

Run the simulation in a `parfor` loop.

```
parfor i=1:4
```

```
    % Set the location of the simulation data repository of this
    % worker to be the same for aggregating the data
    Simulink.sdi.setSource(src);
    % Run the simulation
    simOut = sim(mdl,'SaveOutput','on',...
                 'SaveFormat','StructureWithTime',...
                 'ReturnWorkspaceOutputs','on');
    % Create a simulation run in the Simulink Data Inspector
    Simulink.sdi.createRun(['Run' num2str(i)],'namevalue',...
                           {'simout'},{simOut});
end
```

Close the MATLAB pool and all of the models.

```
matlabpool close;
bdclose all;
```

Refresh the Simulation Data Inspector

```
Simulink.sdi.refresh();
```

See Also

`Simulink.sdi.getSource` | `Simulink.sdi.refresh`

Simulink.sdi.view

Purpose

Open Simulation Data Inspector tool

Syntax

```
Simulink.sdi.view
Simulink.sdi.view(tabType)
```

Description

`Simulink.sdi.view` opens the Simulation Data Inspector tool.

`Simulink.sdi.view(tabType)` opens the Simulation Data Inspector tool and displays the specified tab, `tabType`.

Input Arguments

tabType

An enumeration value that specifies the tab view in the Simulation Data Inspector.

Value	Description
`Simulink.sdi.GUITabType.InspectSignals`	Display **Inspect Signals** tab view
`Simulink.sdi.GUITabType.CompareSignals`	Display **Compare Signals** tab view
`Simulink.sdi.GUITabType.CompareRuns`	Display **Compare Runs** tab view

Examples

Create a run in the Simulation Data Inspector and open the tool to view the simulation output.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a run in the Simulation Data Inspector
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Compare two runs and open the tool to view the comparison.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on',...
             'SaveFormat', 'StructureWithTime',...
             'ReturnWorkspaceOutputs', 'on');

% Create a run in the Simulation Data Inspector and get signal IDs
run1ID = Simulink.sdi.createRun('My Run', 'namevalue',...
                                {'simOut'}, {simOut});

% Change one of the parameters of the model
Mq = 3*Mq;

% Simulate again
simOut = sim('slexAircraftExample', 'SaveOutput','on',...
             'SaveFormat', 'StructureWithTime',...
             'ReturnWorkspaceOutputs', 'on');

% Create another another run get signal IDs
run2ID = Simulink.sdi.createRun('New Run', 'namevalue',...
                                 {'simOut'}, {simOut});

% Define alignment algorithms
algorithms = [Simulink.sdi.AlignType.DataSource
              Simulink.sdi.AlignType.BlockPath
              Simulink.sdi.AlignType.SID];

% Compare the two runs
difference = Simulink.sdi.compareRuns(run1ID, run2ID, algorithms);

% See the results in Simulation Data Inspector on the Compare Runs tab
Simulink.sdi.view(Simulink.sdi.GUITabType.CompareRuns);
```

Alternatives

To open the Simulation Data Inspector tool, in the Simulink Editor window, click the **Record** button arrow and select **Simulation Data Inspector**.

See Also `Simulink.sdi.createRun`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.SimulationData.createStructOfTimeseries

Purpose

Create a structure with MATLAB `timeseries` object leaf nodes

Syntax

```
struct_of_ts = Simulink.SimulationData.createStructOfTimeserie
   s(TsArrayObject)

struct_of_ts = Simulink.SimulationData.createStructOfTimeserie
   s(busObj,structOfTimeseries)

struct_of_ts = Simulink.SimulationData.createStructOfTimeserie
   s(busObj,cellOfTimeseries)
struct_of_ts = Simulink.SimulationData.createStructOfTimeserie
   s(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 models that use the `ModelDataLogs` signal logging format.

`struct_of_ts =`
`Simulink.SimulationData.createStructOfTimeseries(busObj,structOfTimese`
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,cellOfTimeseri`
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,cellOfTimeseri`

creates a structure with the dimensions `dims`. Use this syntax to create a structure to load into an array of buses.

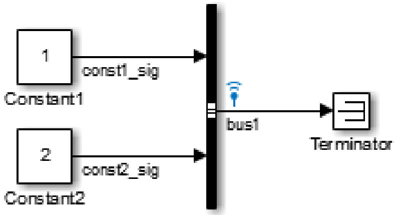
Examples

Structure Based on Simulink.TsArray

Log signal data to create a `Simulink.TsArray` object.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink','examples',...
'ex_log_modeldatalogs_errwarn')))
sim('ex_log_modeldatalogs_errwarn')
```

The simulated `ex_log_modeldatalogs_errwarn` model looks like this:



Use the `Simulink.TsArray` object to create a structure of MATLAB `timeseries` objects. The model uses `ModelDataLogs` format for the signal logging data.

```
logsout
```

```
logsout =

Simulink.ModelDataLogs (ex_log_modeldatalogs_errwarn):
  Name                     Elements  Simulink Class

  bus1                          2      TsArray
```

```
struct_of_ts = ...
```

```
Simulink.SimulationData.createStructOfTimeseries(logsout.bus1)
```

```
struct_of_ts =

    const1-sig: [1x1 timeseries]
    const2_sig: [1x1 timeseries]
```

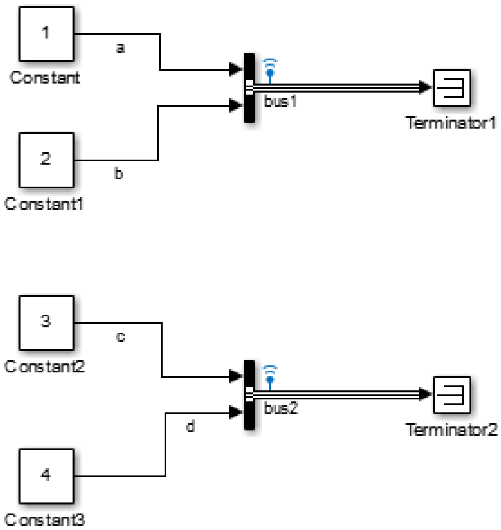
Structure Based on Bus Object and a Partial Structure of Timeseries Data

Create a structure of MATLAB `timeseries` objects based on a `Simulink.Bus` object and a partial structure of MATLAB `timeseries`objects. Use this structure to load into another model.

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:


1
Constant
a
2
Constant1
b
bus1
Terminator1
3
Constant2
c
4
Constant3
d
bus2
Terminator2

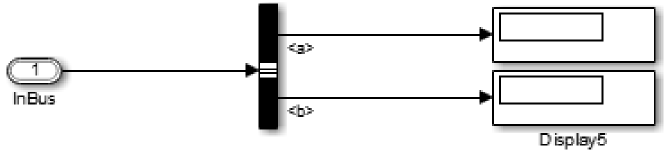
View the logged signal data.

```
ex_log_structTimeSeries_logsout
```

```
ex_log_structTimeSeries_logsout = 

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'ex_log_structTimeSeries_logsout'
    Total Elements: 2

  Elements:
    1: 'bus1'
    2: 'bus2'
```

Open the model to load the logged signal data into.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_load_structTimeSeries_Bus')))
```



The `ex_load_structTimeSeries_Bus` model's **Configuration Parameters > Data Import/Export > Input** parameter lists the `ex_load_structTimeSeries_inputBus` variable. However, you have not yet defined that variable in the MATLAB workspace. Use `Simulink.SimulationData.createStructOfTimeseries` to define that variable.

```
ex_load_structTimeSeries_inputBus = ...
Simulink.SimulationData.createStructOfTimeseries...
('bus', ex_log_structTimeSeries_logsout.get(2).Values)
```

```
ex_load_structTimeSeries_inputBus =

    a: [1x1 timeseries]
    b: [1x1 timeseries]
```

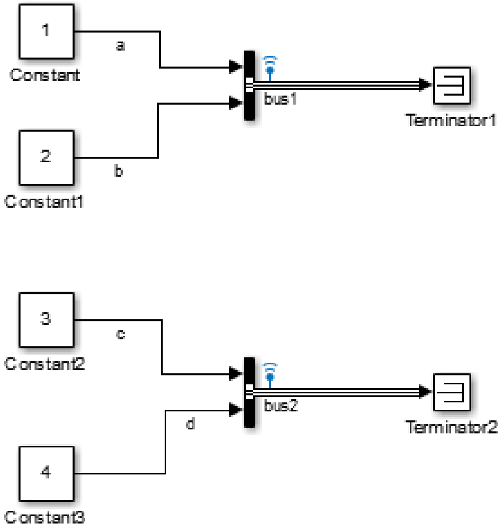
Structure to Use with an Array of Buses

Create a structure of MATLAB `timeseries` objects to load into an array of buses. Specify the dimensions of the created structure and a cell array of MATLAB `timeseries` objects.

Open a model and simulate it, producing signal logging data.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_log_structTimeSeries')))
sim('ex_log_structTimeSeries')
```

The simulated `ex_log_structTimeseries` model looks like this:


1
Constant
a
2
Constant1
b
bus1
Terminator1
3
Constant2
c
4
Constant3
d
bus2
Terminator2

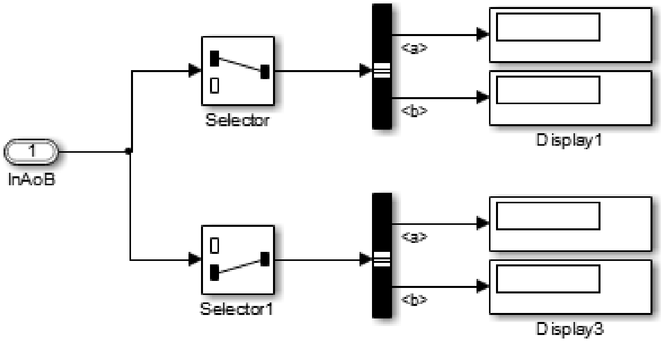
View the logged signal data.

```
ex_log_structTimeSeries_logsout
```

```
ex_log_structTimeSeries_logsout =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'ex_log_structTimeSeries_logsout'
    Total Elements: 2

  Elements:
    1: 'bus1'
    2: 'bus2'
```

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 - Object to convert
`Simulink.TsArray` object

Object to convert to a structure of MATLAB `timeseries` objects, specified as a `Simulink.TsArray` object.

When you log signals using the `ModelDataLogs` format, the logged data consists of `Simulink.TsArray` objects.

busObj - Object for creating a structure of MATLAB `timeseries` objects
`Simulink.Bus` object

Object for creating a structure of MATLAB `timeseries` objects, specified as the name of a `Simulink.Bus` object.

Data Types
`char`

structOfTimeseries - Objects for values to override ground values, specified as a structure of MATLAB `timeseries` objects.
structure of MATLAB `timeseries` objects

Objects 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 - Objects for values to override ground values, specified as a cell array of MATLAB `timeseries` objects.
cell array of MATLAB `timeseries` objects

Objects 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 - MATLAB `timeseries` objects, returned as a structure.

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 `structForBus` depend on the input arguments:

- If you specify `tsArrayObject`, then the dimension is 1.
- If you specify the `busObject` and a structure of MATLAB `timeseries`, then the dimension matches the dimensions of the specified structure.
- If you specify only the `busObject` and a cell array of MATLAB `timeseries`, then the dimension is 1.
- If you specify the `busObject` argument, a cell array of MATLAB `timeseries`, and the `dims` argument, then the dimensions match the dimensions of `dims`.

Related Links

`Simulink.Bus` |`Simulink.TsArray` |`Simulink.ModelDataLogs` |`Simulink.ModelDataLogs.convertToDataset`

Purpose

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

String 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.SimulationData.Dataset` | `Simulink.ModelDataLogs`

How To

- "Override Signal Logging Settings"

Simulink.SimulationData.updateDatasetFormatLogging

Purpose

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

`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. 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 2-513

Tips

- Simulink checks signal logging data format consistency for certain model referencing configurations. For details, see “Model Reference Signal Logging Format Consistency”. You can use the Upgrade Advisor (with the `upgradeadvisor` function) to upgrade a model to use `Dataset` format.
- An alternative approach for handling reported inconsistencies is to use the `Simulink.SimulationData.updateDatasetFormatLogging` function to update the models to use `Dataset` format. This approach sets the **Model Configuration Parameters > Data Import/Export > Signal logging format** parameter to `Dataset` for each referenced model and each variant.
- 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.
- For new models, use the `Dataset` format for signal logging. The `ModelDataLogs` format will be removed in a future release. For an existing model that uses the `ModelDataLogs` format, you should migrate the model to use `Dataset` format. For details, see “Migrate from ModelDataLogs to Dataset Format”.

  After conversion, to reset the format back to `ModelDataLogs`, you must set the **Signal logging format** parameter individually for each model. If you want to use the `ModelDataLogs` format for the model in the future, consider using one of these approaches:

  - Save a version of the model that uses the `ModelDataLogs` format.
  - Use the conversion function to convert the model to use `Dataset` format for signal logging and run the simulation in Normal mode. However, do not save the changes.

Input Arguments

top_model

String that specifies the name of the top-level model.

variants

String 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.

See Also

`Simulink.SimulationData.Dataset` | `Simulink.ModelDataLogs` | `Simulink.ModelDataLogs.convertToDataset`

How To

- “Specify the Signal Logging Data Format”

Simulink.SimulationOutput.find

Purpose

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.

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

`Simulink.SimulationOutput.get`

See Also

`Simulink.SimulationOutput.get` | `Simulink.SimulationOutput.who`

Simulink.SimulationOutput.get

Purpose

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.

Input Arguments

VarName

Name of logged variable for which you seek values.

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

`Simulink.SimulationOutput.find`

See Also

`Simulink.SimulationOutput.find` |
`Simulink.SimulationOutput.who`

Simulink.SimulationOutput.who

Purpose

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

String array of output variable names of simulation.

Examples

Simulate `vdp` and store the string 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
```

See Also

`Simulink.SimulationOutput.find` |
`Simulink.SimulationOutput.get`

Simulink.SoftwareTarget.concurrentExecution

Purpose

Create configuration for concurrent execution

Syntax

```
configSet =
Simulink.SoftwareTarget.concurrentExecution(config
   Set,'ConvertForConcurrentExecution');
configSet =
Simulink.SoftwareTarget.concurrentExecution(modelN
   ame,'ConvertForConcurrentExecution',configSetName);
configSet =
Simulink.SoftwareTarget.concurrentExecution(modelN
   ame,'AddConfigurationForConcurrentExecution');
Simulink.SoftwareTarget.concurrentExecution(modelName,
   'OpenDialog',configSetName);
```

Description

`configSet =`
`Simulink.SoftwareTarget.concurrentExecution(configSet,'ConvertForConcurre`
creates a configuration set object, `configSet`, from an existing configuration set object, `configSet`, for concurrent execution.

`configSet =`
`Simulink.SoftwareTarget.concurrentExecution(modelName,'ConvertForConcurre`
creates an existing configuration set object for the model, `modelName`, which has a configuration set named `configSetName`.

`configSet =`
`Simulink.SoftwareTarget.concurrentExecution(modelName,'AddConfigurationFo`
copies the active configuration set object of model `modelName` and converts it for concurrent execution.

`Simulink.SoftwareTarget.concurrentExecution(modelName,'OpenDialog',config`
opens the task configuration dialog for a model configuration.

Input Arguments

configSetName

Name of original configuration set. Same as `configSet.Name`.

Default: none

modelName

Name of configuration set model.

Default: none

ConvertForConcurrentExecution

Convert existing configuration set to one for concurrent execution.

Default: none

AddConfigurationForConcurrentExecution

Convert existing configuration set to one for concurrent execution.

Default: none

OpenDialog

Open task configuration dialog.

Default: none

Output Arguments

configSet

New or affected configuration set.

Examples

Create a concurrent execution configuration set.

```
% Create a new model
new_system('MyModel');
% Open the new model
open_system('MyModel');
configSet = Simulink.SoftwareTarget.concurrentExecution('MyModel',...
            'AddConfigurationForConcurrentExecution');
```

Alternatives

"Configure Your Model"

See Also `Simulink.SoftwareTarget.TaskConfiguration`

Simulink.SoftwareTarget.profile

Purpose

Profile options for model configured for concurrent execution

Syntax

```
Simulink.SoftwareTarget.profile(modelName)
Simulink.SoftwareTarget.profile(modelName,numSamples)
```

Description

`Simulink.SoftwareTarget.profile(modelName)` generates a profile report for a model configured for concurrent execution,

`Simulink.SoftwareTarget.profile(modelName,numSamples)` specifies the number of samples to generate a profile report. Subsequent calls to the command for the same model name overwrite the existing profile report.

Input Arguments

modelName - Model to profile

Model to profile, specified as a string. 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 number determines the number of steps to collect data for the profiled model.

Data Types
`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Examples

Generate profile report

Generate profile report for the model `sldemo_concurrent_execution`.

```
Simulink.SoftwareTarget.profile('sldemo_concurrent_execution');
```

The command creates the file `sldemo_concurrent_execution_ProfileReport.html` in the current folder.

Generate profile report for 120 time steps

Generate profile report for the model `sldemo_concurrent_execution` with data for 120 time steps.

```
Simulink.SoftwareTarget.profile('sldemo_concurrent_execution',120);
```

The command creates the file `sldemo_concurrent_execution_ProfileReport.html` in the current folder.

Concepts

- “Profile and Evaluate”

Simulink.SubSystem.convertToModelReference

Purpose

Convert atomic subsystem or function call subsystem to model reference

Syntax

```
[success,
   mdlRefBlkH] = Simulink.SubSystem.convertToModelReference(su
   bsys, mdlRef, 'opt1', 'val1', ... , 'optN', 'valN')
```

Description

`[success,mdlRefBlkH] = Simulink.SubSystem.convertToModelReference(subsys, mdlRef, 'opt1', 'val1', ...  , 'optN', 'valN')` converts an atomic subsystem or function call subsystem to a referenced model. The function creates a new model, copies the contents of the subsystem into the model, sets the new model's configuration parameters, and configures the model's root level `Inport` and `Outport` blocks. The function then replaces the subsystem block with a Model block that references the new model, or creates another, temporary model containing a Model block that references the new model, depending on the input option `ReplaceSubsystem`.

Converting a subsystem to a referenced model requires your model to have the following configuration parameter settings:

- **Configuration Parameters > Optimization > Inline parameters** must be `On`.
- **Configuration Parameters > Diagnostics > Data Validity > Signal resolution** must be `Explicit only`.
- **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** must be `Error`.

You can use the following commands to set these parameters to the values required by this function:

```
set_param(mdlName, 'InlineParams', 'on');
set_param(mdlName, 'SignalResolutionControl', 'UseLocalSettings');
set_param(mdlName, 'StrictBusMsg', 'ErrorLevel1');
```

This function produces error or warning messages for models and subsystems that it cannot handle.

- For some errors, a message box appears that gives you the choice of cancelling or continuing.
- If continuing is impossible, Simulink cancels the conversion without offering a choice to continue.

Even if conversion is successful, you may still need to reconfigure the resulting model to meet your requirements.

In the new model, the **Interpolate data** parameter of each root input port is selected by default. You can clear the parameter wherever this default is not appropriate. See the Inport block documentation for information about **Interpolate data**.

Input Arguments

subsys

Full name or handle of the atomic subsystem block to be converted

mdlRef

Name of the new model to which the subsystem is to be converted

opt1, val1, ... , optN, valN

Zero or more pairs of options and values that control the conversion process. The options and values are:

`'ReplaceSubsystem'`	The option value is a Boolean. If the value is `true`, the function replaces the subsystem block with a Model block that references the model created from the subsystem. If this option is omitted or specified as `false`, the function creates and opens a model containing a Model block that references the model derived from the subsystem block. Default: `false`.
`'BusSaveFormat'`	The option value can be `'Cell'` or `'Object'`. If the option is specified, the function saves any bus objects that it creates in a file. The value of the option specifies the format of the file. Use cell array format to save the objects in a compact form.

	If this option is omitted, the function does not save bus objects to a file.
`'BuildTarget'`	The option value can be `'Sim'` or `'RTW'`. If the option is specified, the function generates a model reference Sim or RTW target for the new model, depending on the option value. If this option is omitted, the function does not generate a model reference target.
`'Force'`	The option value is a Boolean. If the value is `true`, the function reports some errors that would halt the conversion process as warnings and continues with the conversion. This allows you to use the function to do the initial steps of conversion, and then complete the conversion process yourself. If this option is omitted or specified as `false`, the function halts the conversion if an error occurs. Default: `false`

Output Arguments

success

The value is `true` if the conversion completed successfully, and `false` otherwise.

mdlRefBlkH

Handle of the `Model` block that references the new model.

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.Bus.save` | `Simulink.SubSystem.copyContentsToBlockDiagram` | “Convert a Subsystem to a Referenced Model”

Tutorials

- `sldemo_mdlref_conversion`

How To

- “Create a Subsystem”

- "Model Reference"

Simulink.SubSystem.copyContentsToBlockDiagram

Purpose

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`.

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`

How To

- “Modeling Basics”
- “Create a Subsystem”

Simulink.SubSystem.deleteContents

Purpose

Delete contents of subsystem

Syntax

```
Simulink.SubSystem.deleteContents(subsys)
```

Description

`Simulink.SubSystem.deleteContents(`*`subsys`*`)` deletes the contents of the subsystem *`subsys`*. The subsystem must have already been loaded. The function affects only blocks, lines, and annotations; it does not affect nongraphical information such as configuration sets.

Input Arguments

subsys

Subsystem name or handle

Examples

Delete the graphical contents of `Controller`, including all nested subsystems:

```
Simulink.SubSystem.deleteContents('f14/Controller');
```

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem | Simulink.BlockDiagram.deleteContents | Simulink.SubSystem.convertToModelReference | Simulink.SubSystem.copyContentsToBlockDiagram`

How To

- “Model Hierarchy”
- “Create a Subsystem”

Simulink.SubSystem.getChecksum

Purpose

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 software computes the checksum based on the subsystem parameter settings and the blocks the subsystem contains.

One use of this command is to determine why code generated for a subsystem is not being reused. For an example, see "Determine Why Subsystem Code Is Not Reused" in the Simulink Coder documentation.

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.

This command returns the following output:

- `checksum` — Structure of the form

  ```
  Value: [4x1 uint32]
  MarkedUnique: [bool]
  ```

  - `Value` — Array of four 32-bit integers that represents the subsystem's 128-bit checksum.
  - `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.

- *`details`* — Structure of the form

  ```
  ContentsChecksum: [1x1 struct]
  InterfaceChecksum: [1x1 struct]
  ContentsChecksumItems: [nx1 struct]
  InterfaceChecksumItems: [mx1 struct]
  ```

  - `ContentsChecksum` — Structure of the same form as *`checksum`*, representing a checksum that provides information about all blocks in the system.
  - `InterfaceChecksum` — Structure of the same form as *`checksum`*, representing a checksum that provides information about the subsystem's block parameters and connections.
  - `ContentsChecksumItems` and `InterfaceChecksumItems` — Structure arrays of the following form 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 string 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)
      ```

Tip

For information about the kinds of changes that affect the structural checksum, see the `Simulink.BlockDiagram.getChecksum` documentation.

See Also

`Simulink.BlockDiagram.getChecksum`

Purpose

Create MATLAB structure describing signed integer data type

Syntax

```
a = sint(TotalBits)
```

Description

`sint(`*`TotalBits`*`)` returns a MATLAB structure that describes the data type of a signed integer with a word size given by *`TotalBits`*.

`sint` is automatically called when a signed integer is specified in a block dialog box.

The default binary point for this data type is assumed to lie to the right of all bits.

Examples

Define a 16-bit signed integer data type:

```
a = sint(16)
a =
       Class: 'INT'
    IsSigned: 1
    MantBits: 16
```

See Also

`fixdt` | `float` | `sfix` | `sfrac` | `ufix` | `ufrac` | `uint`

slbuild

Purpose

Build standalone and model reference targets

Syntax

```
slbuild('model')
slbuild('model','StandaloneRTWTarget')
slbuild('model','StandaloneRTWTarget','ForceTopModelBuild',true)

slbuild('model','CleanTopModel')

slbuild('model','ModelReferenceSimTarget')
slbuild('model','ModelReferenceRTWTarget')
slbuild('model','ModelReferenceRTWTargetOnly')
```

Description

Note Except where noted, this command requires a Simulink Coder license.

`slbuild('model')` builds a standalone Simulink Coder target executable from the specified model, using the current model configuration settings. If the model has not been loaded, `slbuild` loads it before initiating the build process.

`slbuild('model','StandaloneRTWTarget')` builds a standalone Simulink Coder target executable from the specified model (same as previous).

`slbuild('model','StandaloneRTWTarget','ForceTopModelBuild',true)` builds a standalone Simulink Coder target executable from the specified model. Specify the parameter `ForceTopModelBuild` with the value `true` if you want to force regeneration of the top model code. If the parameter is omitted or set to `false`, the build process determines whether regeneration of the top model code is necessary based on model checksums and options.

`slbuild('model','CleanTopModel')` cleans the model build area enough to trigger regeneration of the top model code at the next build.

Note The following commands honor the setting of the **Rebuild** parameter on the **Model Referencing** pane of the Configuration Parameters dialog box for rebuilding the model reference target for this model and its referenced models.

`slbuild('`*`model`*`','ModelReferenceSimTarget')` builds a model reference simulation target for the model. This command does not require a Simulink Coder license.

`slbuild('`*`model`*`','ModelReferenceRTWTarget')` builds model reference simulation and Simulink Coder targets for the model.

`slbuild('`*`model`*`','ModelReferenceRTWTargetOnly')` builds a model reference Simulink Coder target for the model.

If the **Rebuild** parameter on the **Model Referencing** pane of the Configuration Parameters dialog box is set to `Never`, you can use two additional arguments, `'UpdateThisModelReferenceTarget'` and *`Buildcond`*, to specify a rebuild option for building a model reference target for the model. For example,

```
slbuild('model','ModelReferenceSimTarget', ...
       'UpdateThisModelReferenceTarget', Buildcond)
```

conditionally builds the simulation target for *`model`* based on the value of *`Buildcond`*.

Note The `'UpdateThisModelReferenceTarget'` argument rebuilds the model reference target only for the specified model, not for any models referenced by the specified model.

`Buildcond` must be one of the following:

- `'Force'`

Causes `slbuild` to always rebuild the model. This option is equivalent to the `Always` rebuild option on the **Model Referencing** pane of the Configuration Parameters dialog box.

- `'IfOutOfDateOrStructuralChange'`

  Causes `slbuild` to rebuild this model if it 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'`

  Causes `slbuild` to rebuild this model if it 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.

Purpose

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'`

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.

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.

How To

- "Models with Different Character Encodings"
- "Saving Models with Different Character Encodings"

Purpose 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`

Purpose

Display diagnostic information about Simulink system

Syntax

```
[txtRpt, sRpt] = sldiagnostics('sys')
[txtRpt, sRpt] = sldiagnostics('sys', options)
```

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`* strings. 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.

Option	Valid Inputs	Output
`RTWBuildStats`	root model	Lists the same information as the `CompileStats` diagnostic. When issued with the second output argument `sRpt`, it captures the Simulink Coder build statistics in `sRpt.rtwbuild`. You must explicitly specify this option; it is not part of the default analysis.
`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')` or `[txtRpt, sRpt] = sldiagnostics('sys',` *`options`*`)` 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`
- `rtwbuild`

If your chosen options specify just one type of analysis, then `sRpt` contains the results of only that analysis and it is not a structure.

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`

Purpose

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`

How To

- “Discretize a Model from the MATLAB Command Window”

Purpose

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
```

How To

- "Customize Build Process with STF_make_rtw_hook File"
- "Model File Change Notification"

slmdldiscui

Purpose

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`

How To

- "Discretize a Model from the Model Discretizer GUI"

Purpose

Regenerate profiler report from data, `ProfileData`, saved from previous run

Syntax

```
slprofreport(model_nameProfileData)
```

Description

When you run a model with the profiler enabled, the simulation generates the data and saves it in the variable, `model_nameProfileData`. `slprofreport(model_nameProfileData)` generates a profiler report based on the data in `model_nameProfileData`, saved from the model run.

Input Arguments

ProfileData

Variable that contains profiler data from a model run. The variable name consists of the model name and `ProfileData`, for example, `vdpProfileData`.

Default: None

Examples

Regenerate Simulink Profiler Results

Regenerate the Profiler report for model `vdp`

In the MATLAB Command Window, start the `vdp` model.

In the Simulink editor window, run `vdp` model with Simulink Profiler enabled.

Simulink stores the data to the variable `vdpProfileData`.

To review the report, in the MATLAB Command Window

```
slprofreport(vdpProfileData)
```

The Simulink Profiler Report window is displayed.

Related Examples

- "How to Save Simulink Profiler Results"

Concepts

- "Capture Performance Data"

Purpose 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. Use `slproject.getProject` for project automation scripts.

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	Value
Name	Project name	String
Categories	Categories of project labels	Cell array of strings
Files	Paths and labels of project files	Cell array of strings
RootFolder	Full path to project root folder	String
Shortcuts	Shortcut files in project	Cell array of strings

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: [1x2 slproject.Category]
     Shortcuts: [1x6 slproject.Shortcut]
             Files: [1x24 slproject.ProjectFile]
   RootFolder: 'C:\Temp\20121107T135235\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: [1x2 slproject.Category]
         Files: [1x24 slproject.ProjectFile]
    RootFolder: [1x61 char]
```

Find out what you can do with your project.

```
methods(proj)

Methods for class slproject.ProjectManager:

addFile
export
isLoaded
removeFile
close
findCategory
```

```
reload
removeLabel
createCategory
findFile
removeCategory
```

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.getCurrentProject

proj =

  ProjectManager with properties:

          Name: 'Simulink Project Airframe Example'
    Categories: [1x2 slproject.Category]
         Files: [1x24 slproject.ProjectFile]
    RootFolder: [1x61 char]
```

Examine the project files.

```
files = proj.Files

files =

  1x24 ProjectFile array with properties:

    Path
    Labels
```

Examine the labels of the eighth file.

```
proj.Files(8).Labels
```

```
ans =

  Label with properties:

File:  'C:\Temp\airframe\models\AnalogControl.mdl'
                 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]
```

Find out what you can do with the file.

```
methods(myfile)
```

```
Methods for class slproject.ProjectFile:

addLabel
removeLabel
findLabel
```

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.getProject` for project automation scripts.

See Also `simulinkprojectslproject.loadProject`

slproject.loadProject

Purpose

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.

Input Arguments

projectPath - Full path to project file or folder
string

Full path to project `.prj` file or the path to the project root folder, specified as a string.

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	Value
Name	Project name	String
Categories	Categories of project labels	Cell array of strings

Project Property	Description	Value
Files	Paths and labels of project files	Cell array of strings
RootFolder	Full path to project root folder	String
Shortcuts	Shortcut files in project	Cell array of strings

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.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: [1x2 slproject.Category]
     Shortcuts: [1x6 slproject.Shortcut]
         Files: [1x24 slproject.ProjectFile]
    RootFolder: 'C:\Temp\20121107T135235\airframe'
```

Find Project Commands

Get the Airframe project as shown in “Get Airframe Example Project” on page 2-557.

Find project commands.

```
methods(proj)

Methods for class slproject.ProjectManager:

addFile
export
isLoaded
removeFile
close
findCategory
reload
removeLabel
createCategory
findFile
removeCategory
```

Examine Project Properties

After you get a project object, you can examine project properties.

Get the airframe project as shown in "Get Airframe Example Project" on page 2-557.

Examine the project files.

```
files = proj.Files

files =

  1x24 ProjectFile array with properties:

    Path
    Labels
```

Examine the labels of the eighth file.

```
proj.Files(8).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]
```

Find out what you can do with the file.

```
methods(myfile)

Methods for class slproject.ProjectFile:

addLabel
removeLabel
findLabel
```

See Also

`simulinkprojectslproject.getCurrentProject`

Concepts

- “What Are Simulink Projects?”

slreplace_mux

Purpose

Replace Mux blocks used to create buses with Bus Creator blocks

Syntax

```
[muxes, uniqueMuxes, uniqueBds] = slreplace_mux('model')
[muxes, uniqueMuxes, uniqueBds] = slreplace_mux('model',
   reportonly)
```

Description

`[muxes, uniqueMuxes, uniqueBds] = slreplace_mux('model')` reports all Mux blocks that create buses in `model` and in libraries referenced by `model`. A signal created by a Mux block is a bus if the signal meets either or both of the following conditions:

- A Bus Selector block individually selects one or more of the signal's elements (as opposed to the entire signal).
- The signal's components have different data types, numeric types (complex or real), dimensionality, storage classes, or sampling modes.

Before running this command in any form, you should set the **Mux blocks used to create bus signals** connectivity diagnostic to `warning` or `none`. See "Connectivity Diagnostics Overview" for more information.

`[muxes, uniqueMuxes, uniqueBds] = slreplace_mux('model', reportonly)` is equivalent to `[muxes, uniqueMuxes, uniqueBds] = slreplace_mux('model')` if *reportonly* is `true`.

If *reportonly* is `false`, the function reports all Mux blocks that create buses in `model` and in libraries referenced by `model`, and replaces all such Mux blocks with Bus Creator blocks. The function saves the model, if changed, and saves and closes any changed library. You should make a backup copy of your model and libraries before using this form of the command.

Input Arguments

model

The model for which `slreplace_mux` is to report and optionally replace muxes used as buses.

reportOnly

Whether to just generate a report (`true`) or also change the model(s) (`false`).

Default: `true`

Output Arguments

muxes

All Mux blocks used as Bus Creators in the model and in libraries referenced by the model

uniqueMuxes

All Mux blocks used as Bus Creators in the model and in libraries referenced by the model, except blocks in the model that are copies of blocks in libraries

uniqueBds

All models and libraries that use Mux blocks as Bus Creators

See Also

Bus Creator | Mux

How To

- "Connectivity Diagnostics Overview"
- "Mux Signals"
- "About Composite Signals"
- "Avoid Mux/Bus Mixtures"

slupdate

Purpose

Replace blocks from previous releases with latest versions

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` — string 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`

How To

- "Model Upgrades"

targetinstaller

Purpose

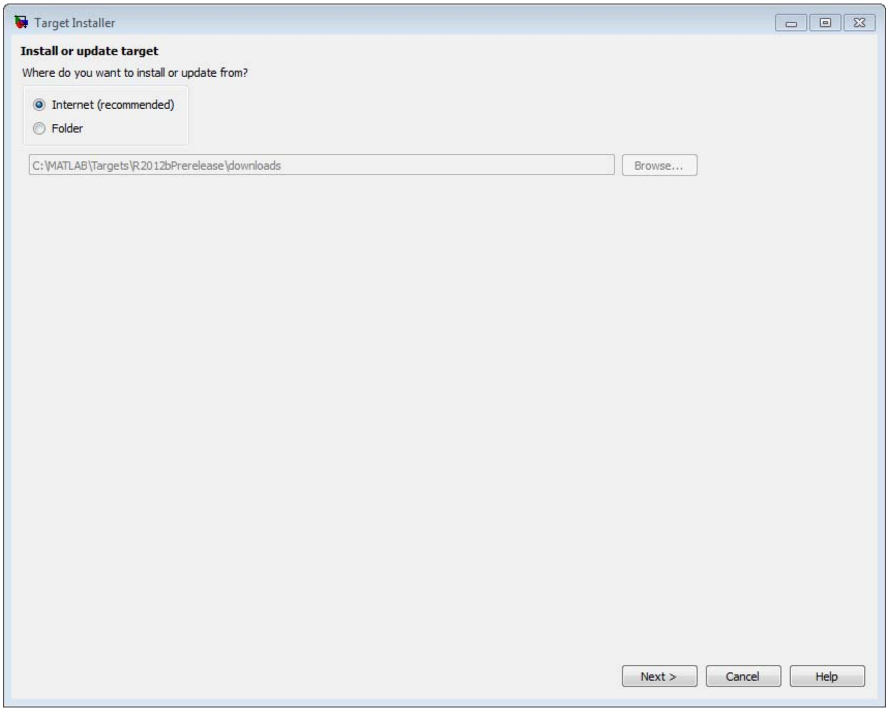
Open Target Installer and install support for third-party hardware or software

Syntax

```
targetinstaller
```

Description

The `targetinstaller` function opens Target Installer at the “Install or update target” screen. Then use Target Installer to install support for third-party hardware or software to MathWorks products.



Install or update target

Where do you want to install or update from?

● Internet (recommended)

○ Folder

C:\MATLAB\Targets\R2012bPrerelease\downloads

The term *target* is given to each collection of features and capabilities for a specific type third-party hardware or software.

The term *support package* is given to the zip file Target Installer uses to install a target.

When a target requires a specific firmware version on the hardware, Target Installer gives you the option to update the firmware automatically.

If you need to update the firmware on multiple pieces of hardware, use the `targetupdater` function. The `targetupdater` function bypasses the target installation process and opens Target Installer at the “Update firmware” screen.

See Also

“Arduino” **|** “BeagleBoard” **|** “LEGO MINDSTORMS NXT” **|** “PandaBoard” **|**

targetupdater

Purpose

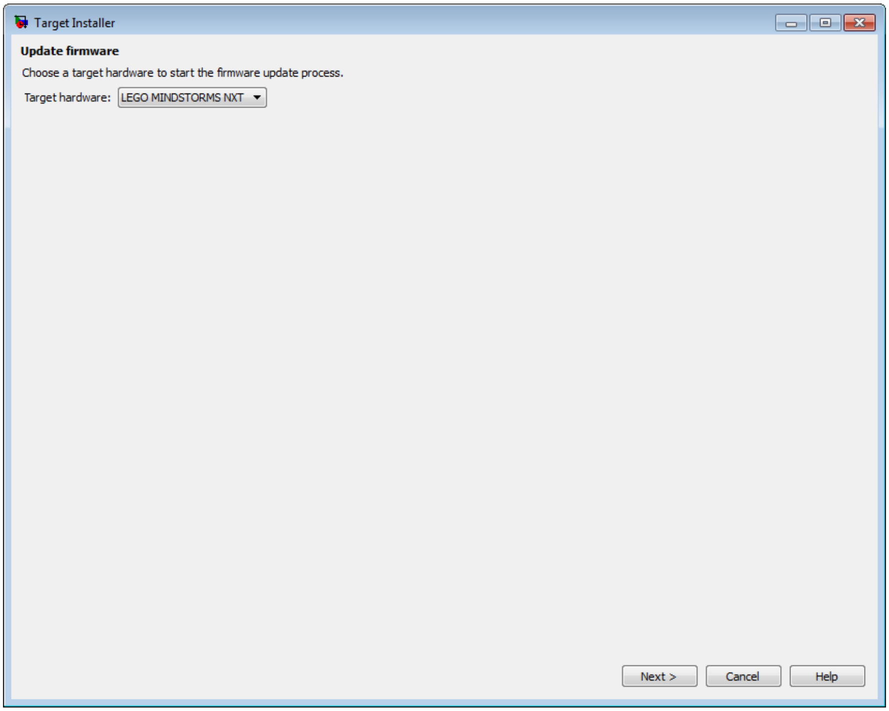
Open Target Installer and update firmware on third-party hardware

Syntax

Description

The `targetupdater` function opens Target Installer at the "Update firmware" screen. Then use Target Installer to update the firmware on third-party hardware.

The following image shows the "Update firmware" screen for a system upon which the LEGO MINDSTORMS NXT target has been installed.



Update firmware

Choose a target hardware to start the firmware update process.

Target hardware: LEGO MINDSTORMS NXT

The `targetupdater` function bypasses the target installation screens in Target Installer and starts the firmware update process directly.

This is useful for updating firmware on multiple pieces of hardware or hardware that has non-operational firmware.

The term *target* is given to each collection of features and capabilities for a specific type third-party hardware or software.

The `targetupdater` function is only available for targets that have already been installed and require special firmware. If **Target hardware** parameter does not present an option for your hardware, use the `targetinstaller` function to open Target Installer. Target Installer will guide you through the process of installing a target for your hardware and, if required, updating the firmware.

See Also

"BeagleBoard" **|** "PandaBoard" **|** "LEGO MINDSTORMS NXT" **|**

Purpose

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 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-0)`. 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.
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.

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)" for a description of this algorithm.

Purpose

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`

How To

- "Tunable Parameters"

Purpose

Create MATLAB structure describing unsigned generalized fixed-point data type

Syntax

```
a = ufix(TotalBits)
```

Description

`ufix(`*`TotalBits`*`)` returns a MATLAB structure that describes the data type of an unsigned generalized fixed-point data type with a word size given by `TotalBits`.

`ufix` is automatically called when an unsigned generalized fixed-point data type is specified in a block dialog box.

Note The default binary point is not included in this data type description. Instead, the scaling must be explicitly defined in the block dialog box.

Examples

Define a 16-bit unsigned generalized fixed-point data type:

```
a = ufix(16)
a =
       Class: 'FIX'
    IsSigned: 0
    MantBits: 16
```

See Also

`fixdt` | `float` | `sfix` | `sfrac` | `sint` | `ufrac` | `uint`

Purpose

Create MATLAB structure describing unsigned fractional data type

Syntax

```
a = ufrac(TotalBits)
a = ufrac(TotalBits, GuardBits)
```

Description

`ufrac(`*`TotalBits`*`)` returns a MATLAB structure that describes the data type of an unsigned fractional number with a word size given by `TotalBits`.

`ufrac(`*`TotalBits`*`, `*`GuardBits`*`)` returns a MATLAB structure that describes the data type of an unsigned fractional number. The total word size is given by *`TotalBits`* with *`GuardBits`* bits located to the left of the binary point.

`ufrac` is automatically called when an unsigned fractional data type is specified in a block dialog box.

The default binary point for this data type is assumed to lie immediately to the left of all bits. If guard bits are specified, then they lie to the left the default binary point.

Examples

Define an 8-bit unsigned fractional data type with 4 guard bits. Note that the range of this number is from 0 to $(1 - 2^{-8}).2^4 = 15.9375$:

```
a = ufrac(8,4)
a =
        Class: 'FRAC'
     IsSigned: 0
     MantBits: 8
    GuardBits: 4
```

See Also

`fixdt` | `float` | `sfix` | `sfrac` | `sint` | `ufix` | `uint`

uint

Purpose

Create MATLAB structure describing unsigned integer data type

Syntax

```
a = uint(TotalBits)
```

Description

`uint(`*`TotalBits`*`)` returns a MATLAB structure that describes the data type of an unsigned integer with a word size given by `TotalBits`.

`uint` is automatically called when an unsigned integer is specified in a block dialog box.

The default binary point for this data type is assumed to lie to the right of all bits.

Examples

Define a 16-bit unsigned integer:

```
a = uint(16)
a =
       Class: 'INT'
    IsSigned: 0
    MantBits: 16
```

See Also

`fixdt` | `float` | `sfix` | `sfrac` | `sint` | `ufix` | `ufrac`

Purpose

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

`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.ScopeDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` | `who` | `whos`

How To

- "Export Signal Data Using Signal Logging"

upgradeadvisor

Purpose

Open Upgrade Advisor

Syntax

```
upgradeadvisor('modelname')
```

Description

`upgradeadvisor('modelname')` opens the Upgrade Advisor for the model specified by `modelname`. If the specified model is not open, this command opens it. Use the Upgrade Advisor to help you upgrade and improve models with the current release.

Tips

- You can also open the Upgrade Advisor from the Model Editor, by selecting **Analysis > Model Advisor > Upgrade Advisor**.
- 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.

Input Arguments

modelname

String specifying the name or handle to the model.

Examples

The command

```
upgradeadvisor('vdp')
```

opens the Upgrade Advisor on the `vdp` example model.

The command

```
upgradeadvisor('f14/Aircraft Dynamics Model')
```

opens the Upgrade Advisor on the Aircraft Dynamics Model subsystem of the `f14` example model.

The command

```
upgradeadvisor(bdroot)
```

opens the Upgrade Advisor on the currently selected model.

Alternatives

You can also open the Upgrade Advisor from the Simulink Model 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`

Related Examples

- "Consult the Upgrade Advisor"
- "Consult the Model Advisor"

view_mdlrefs

Purpose

Display graph of model reference dependencies

Syntax

`view_mdlrefs('modelName')`

Description

`view_mdlrefs('modelName')` launches the Model Dependency Viewer, which displays a graph of model reference dependencies for the model specified by *`modelName`*. The nodes in the graph represent Simulink models. The directed lines indicate model dependencies.

The default display omits library blocks. You could see this same display by opening *`modelName`* and choosing **Analysis > Model Dependencies > Model Dependency Viewer > Models Only** from the model menu. Use **Analysis > Model Dependencies > Model Dependency Viewer**to see other dependency displays.

The Model Dependency Viewer is the same tool, and provides the same options, whether you launch it by typing `view_mdlrefs('modelName')` or by using the Simulink GUI. To see an example of using the Model Dependency Viewer, type `sldemo_mdlref_depgraph` in the MATLAB Command Window.

See Also

Model | `find_mdlrefs`

Tutorials

- `sldemo_mdlref_depgraph`

How To

- “Model Reference”
- “Model Dependency Viewer”

who

Purpose

List names of top-level data logging objects in Simulink data log

Syntax

```
log.who
tsarray.who
log.who('systems')
log.who('all')
```

Description

`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 whoin the MATLAB Command Window.

See Also

`Simulink.ModelDataLogs` | `Simulink.SubsysDataLogs` | `Simulink.ScopeDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` | `whos` | `unpack`

How To

- "Signal Logging Object"

whos

Purpose

List names and types of top-level data logging objects in Simulink data log

Syntax

```
log.whos
tsarray.whos
log.whos('systems')
log.whos('all')
```

Description

`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.

See Also

`Simulink.ModelDataLogs` | `Simulink.SubsysDataLogs` | `Simulink.ScopeDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` | `who` | `unpack`

How To

- “Signal Logging Object”

Mask Icon Drawing Commands

color	Change drawing color of subsequent mask icon drawing commands
disp	Display text on masked subsystem icon
dpoly	Display transfer function on masked subsystem icon
droots	Display transfer function on masked subsystem icon
fprintf	Display variable text centered on masked subsystem icon
image	Display RGB image on masked subsystem icon
patch	Draw color patch of specified shape on masked subsystem icon
plot	Draw graph connecting series of points on masked subsystem icon
port_label	Draw port label on masked subsystem icon
text	Display text at specific location on masked subsystem icon

color

Purpose

Change drawing color of subsequent mask icon drawing commands

Syntax

`color(colorstr)`

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
```

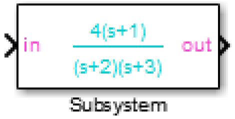
Entering any other string or specifying the color using RGB values results in a warning at the MATLAB command prompt; Simulink ignores the color change. The specified drawing color does not influence the color used by the `patch` or `image` drawing commands.

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`

disp

Purpose

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.

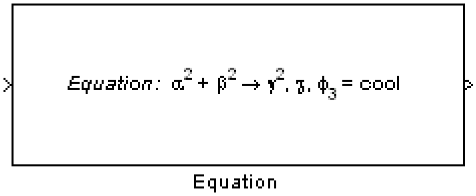
`disp(text, 'texmode', 'on')` allows you to use TeX formatting commands in *`text`*. The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See “Mathematical Symbols, Greek Letters, and TeX Characters” in the MATLAB documentation for information on the TeX formatting commands supported by Simulink software.

Examples

The following command

```
disp('{\itEquation:}  \alpha^2 + \beta^2 \rightarrow \gamma^2,
\chi, \phi_3 = {\bfcool}', 'texmode','on')
```

draws the equation that appears on this masked block icon.



See Also

`fprintf` | `port_label` | `text`

Purpose

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*.

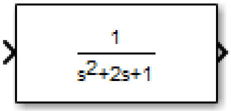
`dpoly(num, den, 'character')` specifies the name of the transfer function independent variable. The default is *s*.

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)
  ```

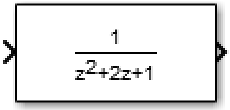
  For example, for `num = [0 0 1];` and `den = [1 2 1]` the icon looks like:

- To display a discrete transfer function in descending powers of *z*, enter

  ```
  dpoly(num, den, 'z')
  ```

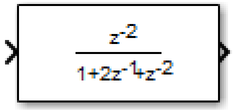
  For example, for `num = [0 0 1];` and `den = [1 2 1];` the icon looks like:

- 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` | `port_label` | `text` | `droots`

droots

Purpose

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*.

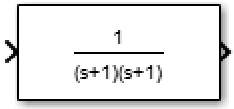
When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- 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` | `port_label` | `text` | `dpoly`

fprintf

Purpose

Display variable text centered on masked subsystem icon

Syntax

```
fprintf(text)
fprintf(format, var)
```

Description

The `fprintf` command displays formatted text centered on the icon and can display *`format`* along with the contents of *`var`*.

Note While this `fprintf` function is identical in name to its corresponding MATLAB function, it provides only the functionality described on this page.

Examples

The command

```
fprintf('Hello');
```

displays the string `'Hello'` on the icon.

The command

```
fprintf('Juhi = %d',17);
```

uses the decimal notation format (`%d`) to display the variable `17`.

See Also

`disp` | `port_label` | `text`

Purpose

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'`.

Examples

The command

```
image(imread('icon.jpg'))
```

reads the icon image from a JPEG file named `icon.jpg` in the MATLAB path.

The following commands read and convert a GIF file, `label.gif`, to the appropriate matrix format. You can type these commands in the **Initialization** pane of the Mask Editor.

```
[data, map]=imread('label.gif');
pic=ind2rgb(data,map);
```

Then type the command

```
image(pic)
```

in the **Icon** pane of the Mask Editor to read the converted label image.

See Also

`patch` | `plot`

Purpose

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])
```

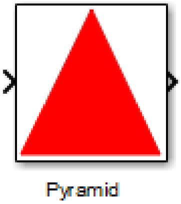
creates a red triangle on the mask's icon.

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`

plot

Purpose

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 value of the **Drawing coordinates** parameter.

Simulink software displays three question marks (`?  ?  ?`) in the block icon and issues warnings in these situations:

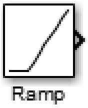
- When you have not defined values for the parameters used in the drawing commands (for example, when you first create the mask, but have not yet entered values in the Mask Settings dialog box)
- When you enter a masked block parameter or drawing command incorrectly

Examples

The command

```
plot([0 1 5], [0 0 4])
```

generates the plot that appears on the icon for the Ramp block, in the Sources library.



See Also `image`

port_label

Purpose

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

The input argument *`port_number`* is an integer, and *`label`* is a string 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 "Mathematical Symbols, Greek Letters, and TeX Characters" in the MATLAB documentation for information on the TeX formatting commands that the Simulink software supports.

Examples

The command

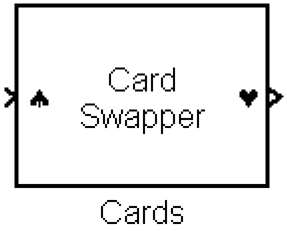
```
port_label('input', 1, 'a')
```

defines `a` as the label of input port 1.

The 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`

Purpose

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 string at a location specified by the point `(x,y)`. The units depend on the **Drawing coordinates** parameter.

`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 “Mathematical Symbols, Greek Letters, and TeX Characters” in the MATLAB documentation for information on the TeX formatting commands supported by Simulink software.

You can optionally specify the horizontal and/or vertical alignment of the text relative to the point (x, y) in the `text` command.

The `text` command offers the following horizontal alignment options.

Option	Aligns
`'left'`	The left end of the text at the specified point
`'right'`	The right end of the text at the specified point
`'center'`	The center of the text at the specified point

The `text` command offers the following vertical alignment options.

Option	Aligns
`'base'`	The baseline of the text at the specified point
`'bottom'`	The bottom line of the text at the specified point
`'middle'`	The midline of the text at the specified point

Option	Aligns
`'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

The command

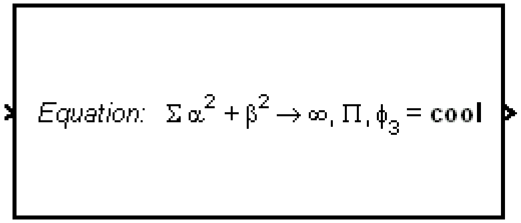
```
text(0.5, 0.5, 'foobar', 'horizontalAlignment', 'center')
```

centers `foobar` in the icon.

The command

```
text(.05,.5,'{\itEquation:}  \Sigma \alpha^2 +
\beta^2 \rightarrow \infty, \Pi, \phi_3 = {\bfcool}',
'hor','left','texmode','on')
```

draws a left-aligned equation on the icon.


Equation: $\Sigma\alpha^2 + \beta^2 \rightarrow \infty, \Pi, \phi_3 =$ **cool**

Equation

See Also

`disp` | `fprintf` | `port_label`

4

Simulink Debugger Commands

animate	Enable or disable animation mode
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
zclist	List blocks containing nonsampled zero crossings

animate

Purpose Enable or disable animation mode

Syntax `animate [delay | stop]`

Short Form `ani`

Arguments

`delay`	Length in seconds between method calls (1 second by default).
`stop`	Disable animation mode.

Description `animate` without any arguments enables animation mode. `animate delay` enables animation mode and specifies delay as the time delay in seconds between method calls. `animate` **`stop`** disables animation mode.

See Also `continue`

Purpose

Show algebraic loop

Syntax

```
ashow <gcb | s:b | s#n | clear>
```

Short Form

`as`

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`

atrace

Purpose Set algebraic loop trace level

Syntax `atrace level`

Short Form `at`

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`

Purpose

Insert breakpoint after specified method

Syntax

```
bafter
bafter m:mid
bafter <sysIdx:blkIdx | gcb> [mth] [tid:TID]
bafter <s:sysIdx | gcs> [mth] [tid:TID]
bafter model [mth] [tid:TID]
```

Short Form

`ba`

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.

`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` | `ebreak` | `tbreak` | `xbreak` | `nanbreak` | `zcbreak` | `rbreak` | `clear` | `where` | `slist` | `systems`

Purpose

Insert breakpoint before specified method

Syntax

```
break
break m:mid
break <sysIdx:blkIdx | gcb> [mth] [tid:TID]
break <s:sysIdx | gcs> [mth] [tid:TID]
break model [mth] [tid:TID]
```

Short Form

`b`

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.

`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.

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` | `systems` | `tbreak` | `where` | `xbreak` | `zcbreak` | `slist`

Purpose Show specified block

Syntax

```
bshow s:b
```

Short Form

```
bs
```

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`

clear

Purpose

Clear breakpoints from model

Syntax

```
clear
clear m:mid
clear id
clear <sysIdx:blkIdx | gcb>
```

Short Form

```
cl
```

Arguments

`mid`	Method ID
`id`	Breakpoint ID
`sysIdx``:`*`blkIdx`*	Block ID
`gcb`	Currently selected block

Description

`clear` clears a breakpoint from the current method.

`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

`break` | `bafter` | `slist`

Purpose Continue simulation

Syntax `continue`

Short Form `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 `run` | `stop` | `quit` | `animate`

Purpose

Display block's I/O when simulation stops

Syntax

```
disp
disp gcb
disp s:b
```

Short Form

```
d
```

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.

See Also

`undisp` | `slist` | `probe` | `trace`

Purpose Enable (or disable) breakpoint on solver errors

Syntax `ebreak`

Short Form `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 `break` | `bafter` | `tbreak` | `xbreak` | `nanbreak` | `zcbreak` | `rbreak` | `clear` | `where` | `slist` | `systems`

elist

Purpose

List simulation methods in order in which they are executed during simulation

Syntax

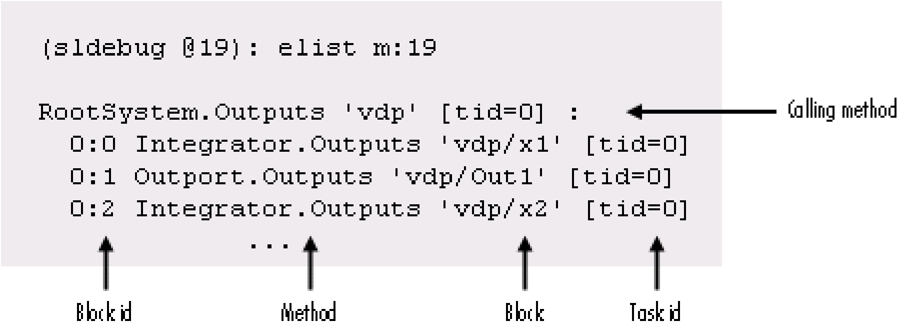
```
elist m:mid [tid:TID]
elist <gcs | s:sysIdx> [mth] [tid:TID]
elist <gcb | sysIdx:blkIdx> [mth] [tid:TID]
```

Short Form

`el`

Description

`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.

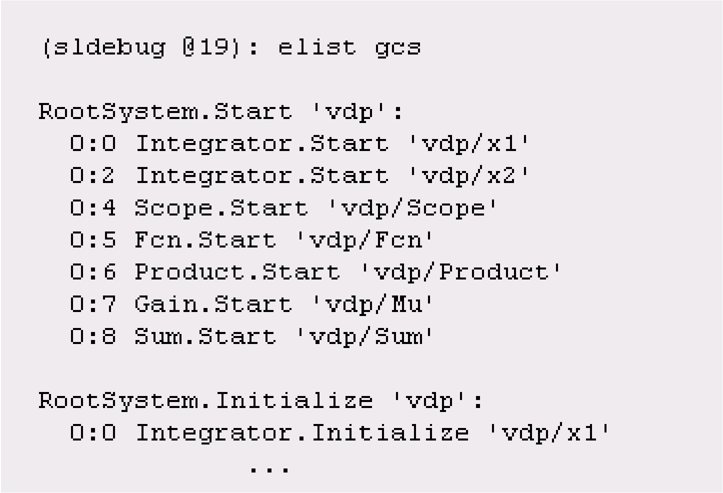
- The name of the method

  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.,

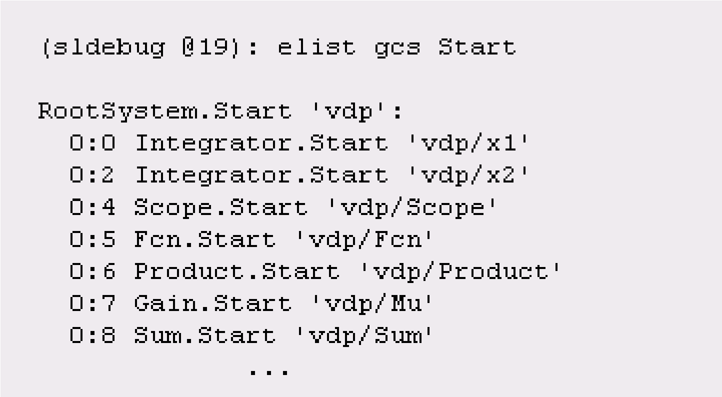


```
(sldebug @19): elist gcs

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

`where` | `slist` | `systems`

emode

Purpose Toggle model execution between accelerated and normal mode

Syntax `emode`

Short Form `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”.

Purpose

Enable or disable method tracing

Syntax

```
etrace level level-number
```

Short Form

```
et
```

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` | `where` | `trace`

help

Purpose Display help for debugger commands

Syntax `help`

Short Form `?` or `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.

Purpose

Set or clear nonfinite value break mode

Syntax

```
nanbreak
```

Short Form

```
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

`break` | `bafter` | `rbreak` | `tbreak` | `xbreak` | `zcbreak`

How To

- `ebreak`

Purpose Advance simulation to start of next method at current level in model's execution list

Syntax `next`

Short Form `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`

Purpose

I/O and state data for blocks

Syntax

```
probe
probe s:b
probe gcb
probe level level-type
p
```

Description

`probe` sets the Simulink debugger to interactive probe mode. In this mode, the debugger displays the I/O of a selected block. To exit interactive probe mode, enter a debugger command or press the **Enter** key.

`probe` *`s:b`* displays the I/O of the block whose system index is *`s`* and block index is *`b`*.

`probe gcb` displays the I/O of the currently selected block.

`probe level` *`level-type`* sets the verbosity level for `probe`, `trace`, and `dis`. If *`level-type`* is `io`, the debugger displays block I/O. If *`level-type`* is `all` (default), the debugger displays all information for the current state of a block, including inputs and outputs, states, and zero crossings.

`p` is the short form of the command.

Examples

Display I/O for the currently selected block `Out2` in the model `vdp` using the Simulink debugger.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   ```
   probe gcb
   ```

The MATLAB Command Window displays:

```
probe: Data of 0:3 Outport block 'vdp/Out2':
U1     = [0]
```

See Also

`disp` | `trace`

Purpose

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`

rbreak

Purpose

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

`break` | `bafter` | `nanbreak` | `ebreak` | `tbreak` | `xbreak` | `zcbreak`

Purpose

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` | `stop` | `quit`

slist

Purpose

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

`systems | elist`

states

Purpose

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')
   ```

Purpose

Debugging options in effect

Syntax

`status`

Description

`status` displays a list of the debugging options in effect.

Examples

Display status 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:

   ```
   status
   ```

   The MATLAB Command Window displays:

   ```
   %----------------------------------------------------------------%
   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
   Window reuse                                 : not supported
   ```

```
Execution Mode                          : Normal
Display level for etrace                : 0 (disabled)
Break points                            : none installed
Display points                          : none installed
```

Purpose

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.

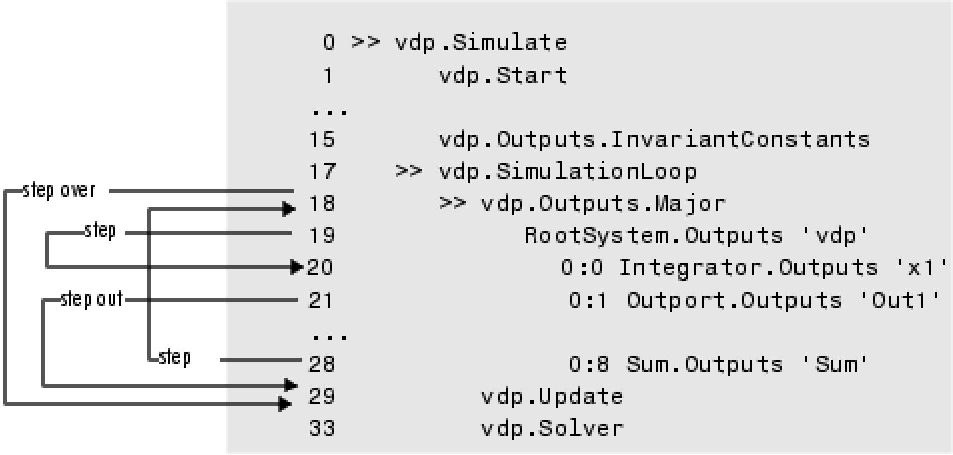
`s` is the short form of the command.

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 `next` | `where` | `elist`

stimes

Purpose 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)
   ```

Purpose

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` | `run` | `quit`

Purpose 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
 TzR = <time while in major step at or just after (at right post of
 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`

systems

Purpose

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'
    1   'sldemo_enginewc/Compression'
    2   'sldemo_enginewc/Controller/TmpAtomicSubsysAtSwitchInport3'
    3   'sldemo_enginewc/Controller/TmpAtomicSubsysAtSwitchInport1'
    4   'sldemo_enginewc/Controller'
    5   'sldemo_enginewc/Throttle & Manifold/Throttle/TmpAtomicSubsysAtSwi
    6   'sldemo_enginewc/valve timing/positive edge to dual edge conversic
   ```

See Also

`slist`

Purpose Set or clear time breakpoint

Syntax

```
tbreak
tbreak t
```

Short Form `tb`

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.

See Also `break` | `bafter` | `xbreak` | `nanbreak` | `zcbreak` | `rbreak`

How To

- `ebreak`

trace

Purpose

Display block's I/O each time block executes

Syntax

```
trace gcb
trace s:b
```

Short Form

`tr`

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` | `untrace` | `slist` | `strace`

Purpose

Remove block from debugger's list of display points

Syntax

```
undisp gcb
undisp s:b
```

Short Form

```
und
```

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`

untrace

Purpose Remove block from debugger's list of trace points

Syntax

```
untrace gcb
untrace s:b
```

Short Form

```
unt
```

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.

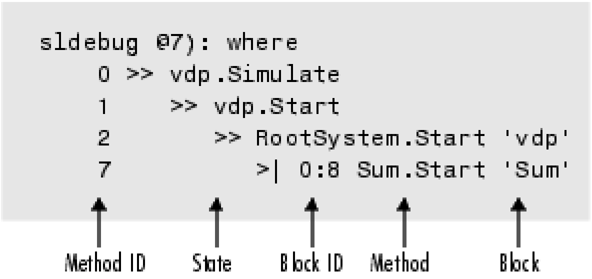
See Also `trace` | `slist`

Purpose Display current location of simulation in simulation loop

Syntax `where [detail]`

Short Form `w`

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

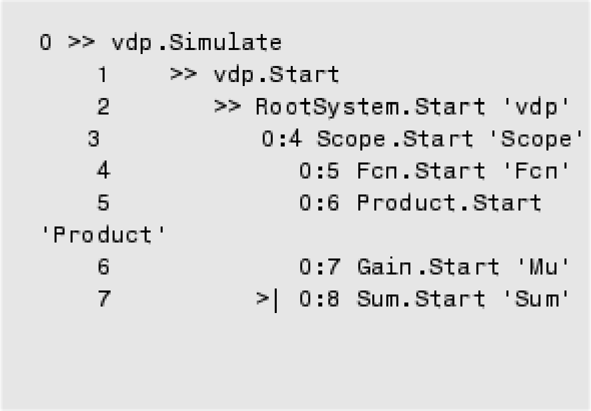
  The method ID identifies a specific invocation of a method.

- 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`

Purpose

Break when debugger encounters step-size-limiting state

Syntax

```
xbreak
```

Short Form

```
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

`break` | `bafter` | `zcbreak` | `tbreak` | `nanbreak` | `rbreak`

How To

- `ebreak`

zcbreak

Purpose Toggle breaking at nonsampled zero-crossing events

Syntax `zcbreak`

Short Form `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 `break` | `bafter` | `xbreak` | `tbreak` | `nanbreak` | `zclist`

Purpose List blocks containing nonsampled zero crossings

Syntax `zclist`

Short Form `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`

Simulink Classes

eventData	Provide information about block method execution events
getDescription	Extract file description without loading block diagram into memory
getMetadata	Extract file metadata without loading block diagram into memory
Simulink.AliasType	Create alias for signal and/or parameter data type
Simulink.Annotation	Specify properties of model annotation
Simulink.BlockCompDworkData	Provide postcompilation information about block's DWork vector
Simulink.BlockCompInputPortData	Provide postcompilation information about block input port
Simulink.BlockCompOutputPortData	Provide postcompilation information about block output port
Simulink.BlockData	Provide run-time information about block-related data, such as block parameters
Simulink.BlockPath	
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.Bus	Specify properties of signal bus
Simulink.BusElement	Describe element of signal bus
Simulink.ConfigSet	Access model configuration set
Simulink.ConfigSetRef	Link model to configuration set stored independently of any model
Simulink.GlobalDataTransfer	Configure concurrent execution data transfers
Simulink.Mask	Control masks programmatically
Simulink.MaskParameter	Control mask parameters programmatically
Simulink.MaskUIElement	Control mask user interface elements programmatically.
Simulink.MDLInfo	Extract model file information without loading block diagram into memory
Simulink.ModelAdvisor	Run Model Advisor from MATLAB file
Simulink.ModelDataLogs	Container for model's signal data logs
Simulink.ModelManagement.Project.CurrentProject	(To be removed) Manipulate the currently open Simulink project from the command line
Simulink.ModelWorkspace	Describe model workspace
Simulink.MSFcnRunTimeBlock	Get run-time information about Level-2 MATLAB S-function block
Simulink.NumericType	Specify data type
Simulink.ParamCoderInfo	Specify information needed to generate code for parameter
Simulink.Parameter	Specify value, value range, data type, and other properties of block parameter

Simulink.RunTimeBlock	Allow Level-2 MATLAB S-function and other MATLAB programs to get information about block while simulation is running
Simulink.SampleTime	Object containing sample time information
Simulink.ScopeDataLogs	Store data logged by Scope signal viewer
Simulink.scopes.TimeScopeConfiguration	Configure Time Scope for programmatic access
Simulink.sdi.DiffRunResult	Results from comparing two simulation runs
Simulink.sdi.DiffSignalResult	Results from comparing two signals
Simulink.sdi.Run	Manages signal data and metadata of simulation run
Simulink.sdi.Signal	Manages signal time series data and metadata
Simulink.Signal	Specify attributes of signal
Simulink.SignalCoderInfo	Specify information needed to generate code for signal
Simulink.SimState.ModelSimState	Access SimState snapshot data
Simulink.SimulationData.BlockPath	Fully specified Simulink block path
Simulink.SimulationData.DataStoreMemory	
Simulink.SimulationData.LoggingInfo	Signal logging override settings
Simulink.SimulationData.ModelLoggingInfo	Signal logging override settings for a model
Simulink.SimulationData.Signal	
Simulink.SimulationData.SignalLoggingInfo	Signal logging override settings for signal
Simulink.SimulationOutput	Access object values of simulation results

Simulink.SiulationData.Dataset	Container for logged data using MATLAB `timeseries` objects
Simulink.SoftwareTarget.AperiodicTrigger	Describe aperiodic trigger
Simulink.SoftwareTarget.PeriodicTrigger	Describe periodic trigger
Simulink.SoftwareTarget.PosixSignalHandler	Describe aperiodic trigger for POSIX targets
Simulink.SoftwareTarget.Task	Describe task that models unit of concurrent execution
Simulink.SoftwareTarget.TaskConfiguration	Configure model for concurrent execution
Simulink.SoftwareTarget.Trigger	Base class for PeriodicTrigger and AperiodicTrigger
Simulink.SoftwareTarget.WindowsEventHandler	Describe aperiodic trigger for Windows® targets
Simulink.SubsysDataLogs	Container for subsystem's signal data logs
Simulink.TimeInfo	Provide information about time data in `Simulink.Timeseries` object
Simulink.Timeseries	Store data for any signal except mux or bus signal
Simulink.TsArray	Store data for mux or bus signal
Simulink.VariableUsage	Get information about workspace variables and blocks that use them
Simulink.Variant	Specify a model reference variant and its execution environment
Simulink.WorkspaceVar	Contains information about workspace variables and blocks that use them

Purpose

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"	Type of method execution event that occurred.
"Source"	Block that triggered the event.

Properties

`Type`

Description

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.

Data Type

string

Access

`RO`

`Source`

Description

Block that triggered the event

Data Type

`Simulink.RunTimeBlock`

Access

`RO`

Purpose

Create alias for signal and/or parameter data type

Description

This class allows you to designate MATLAB variables as aliases for signal and parameter data types. You do this by creating instances of this class and assigning them to variables in the MATLAB or model workspaces (see "Creating a Data Type Alias" on page 5-7). The MATLAB variable to which a `Simulink.AliasType` object is assigned is called a data type alias. The data type to which an alias refers is called its base type. Simulink software allows you to set the `BaseType` property of the object that the variable references, thereby designating the data type for which it is an alias.

Simulink software lets you use aliases instead of actual type names in dialog boxes and `set_param` commands to specify the data types of Simulink block outputs and parameters. Using aliases to specify signal and parameter data types can greatly simplify global changes to the signal and parameter data types that a model specifies. In particular, changing the data type of all signals and parameters whose data type is specified by an alias requires only changing the base type of the alias. By contrast, changing the data types of signals and parameters whose data types are specified by an actual type name requires respecifying the data type of each signal and parameter individually.

Note Suppose you specify an instance of the `Simulink.AliasType` class as the value of a `Simulink.Parameter` object's **Data type** property. If you enter the parameter object in a subsystem's mask, the subsystem displays the data type's base type instead of its alias name.

Creating a Data Type Alias

You can use either the Model explorer or MATLAB commands (see "MATLAB Commands for Creating Data Type Aliases" on page 5-8) to create a data type alias.

To use the Model explorer to create an alias:

1. Select `Base Workspace` (i.e., the MATLAB workspace) in the Model explorer's **Model Hierarchy** pane.

   You must create data type aliases in the MATLAB workspace. If you attempt to create an alias in a model workspace, Simulink software displays an error.

2. Select **Simulink.AliasType** from the Model explorer's **Add** menu.

   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 in the Model explorer's **Contents** pane.

4. Enter the name of the data type that this alias represents in the **Base type** field in the Model explorer's **Dialog** pane.

   You can specify the name of any existing standard or user-defined data type in this field. Skip this step if the desired base type is `double` (the default).

5. Use the MATLAB `save` command to save the newly created alias in a MAT-file that can be loaded by the models in which it is used.

MATLAB Commands for Creating Data Type Aliases

Use the following syntax to create a data type alias at the MATLAB command line or in a MATLAB program

```
ALIAS = Simulink.AliasType;
```

where `ALIAS` is the name of the variable that you want to serve as the alias. For example, the following line creates an alias names `MyFloat`.

```
MyFloat = Simulink.AliasType;
```

The following notations get and set the properties of a data type alias, respectively,

```
PROPVALUE = ALIAS.PROPNAME;
ALIAS.PROPNAME = PROPVALUE;
```

where `ALIAS` is the name of the alias object, `PROPNAME` is the name of the alias object's properties, and `PROPVALUE` is the property's value. For example, the following code saves the current value of `MyFloat`'s `BaseType` property and assigns it a new value.

```
old = MyFloat.BaseType;
MyFloat.BaseType = 'single';
```

See "Properties" on page 5-13 for information on the names, permitted values, and usage of the properties of data type alias objects.

Data Type Aliases in the Generated Code

If you have a Simulink Coder license, you can cause data type aliases to appear in the code generated for a model using any of the following methods.

- Specifying the signal data type of a block in the model as a `Simulink.AliasType` via the **Block Parameters** dialog box.
- Creating a `Simulink.Signal` object that uses the `Simulink.AliasType` as its data type. Use this signal object as the name of a signal in the model and specify that the signal name must resolve to an object in the MATLAB workspace. See "Signal Objects" in the Simulink Coder User's Guide for more information.
- Creating a `Simulink.Parameter` object that uses the `Simulink.AliasType` as its data type. Use this parameter object as a block parameter in the model. See "Generated Code for Parameter Data Types" in the Simulink Coder User's Guide for more information.

Notes

- If you assign a data type in a block's **Block Parameters** dialog box and by using a `Simulink.Signal` object on the signal feeding into the block, the code is always generated using the data type in the dialog box.
- The Simulink Coder code generator tries to preserve the names of alias types in the generated code. However, in some cases, an alias type name might revert to its underlying equivalent built-in data type. If you have a Embedded Coder license, you can guarantee that the code generator uses the alias type name in the generated code, by using replacement types (see "Data Type Replacement" in the Embedded Coder documentation).
- The `Simulink.AliasType` class does not support multiword data types for code generation.

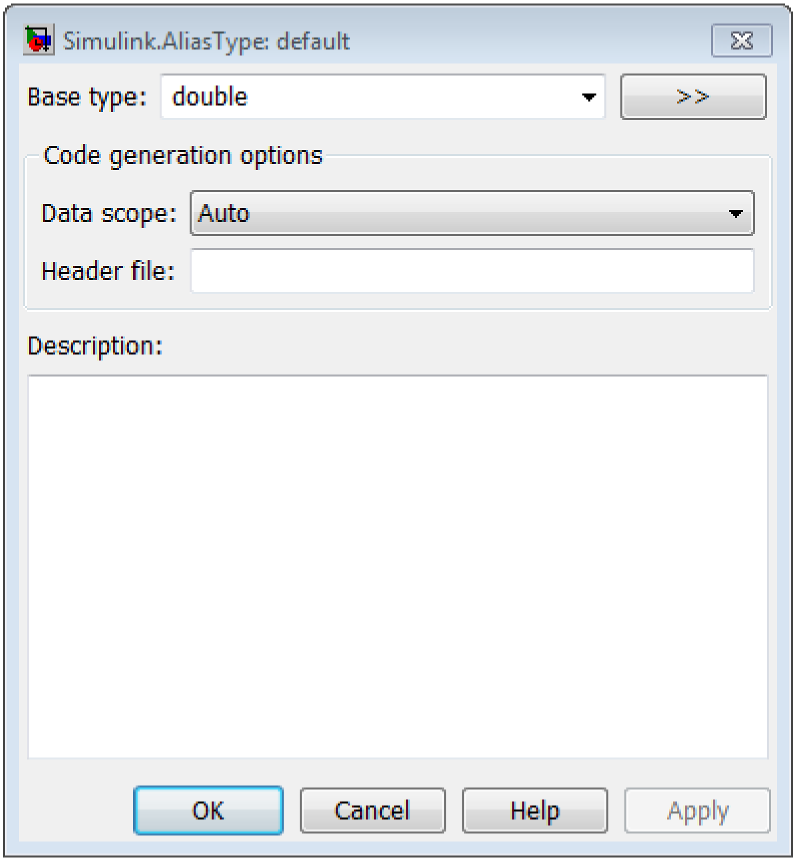
Parent

None

Children

None

Property Dialog Box



Simulink.AliasType: default
Base type:
double
>>
Code generation options
Data scope:
Auto
Header file:
Description:
OK
Cancel
Help
Apply

Base type

The data type to which this alias refers. The default is `double`. To specify another data type, select the data type from the adjacent pull-down list of standard data types or enter the data type's name in the edit field. Note that you can, with one exception, specify a nonstandard data type, e.g., a data type defined by a `Simulink.NumericType` object, by entering the data type's name in the edit field. The exception is a `Simulink.NumericType` whose `Category` is `Fixed-point: unspecified scaling`.

Note `Fixed-point: unspecified scaling` is a partially specified type whose definition is completed by the block that uses the `Simulink.NumericType`. Forbidding its use in alias types avoids creating aliases that have different base types depending on where they are used.

Data scope

Specifies whether the data type definition should be imported from, or exported to, a header file during code generation. The possible values are:

Value	Action
`Auto` (default)	If no value is specified for **Header file**, export the type definition to `*model*_types.h`, where `*model*` is the model name. If a value is specified for **Header file**, import the data type definition from the specified header file.
`Exported`	Export the data type definition to a header file, which can be specified in the **Header file** field. If no value is specified for **Header file**, the header file name defaults to `*type*.h`, where`*type*` is the data type name.
`Imported`	Import the data type definition from a header file, which can be specified in the **Header file** field. If no value is specified for **Header file**, the header file name defaults to `*type*.h`, where`*type*` is the data type name.

Header file

Name of a C header file from which a data type definition is imported, or to which a data type definition is exported, based on the value of **Data scope**. If this field is specified, the specified name is used during code generation for importing or exporting. If this field is empty, the value defaults to `*type*.h` if **Data scope** equals `Imported` or `Exported`, or defaults to `*model*_types.h` if **Data scope** equals `Auto`.

Description

Describes the usage of the data type referenced by this alias.

Properties

Name	Description
`BaseType`	A string specifying the name of a standard or custom data type. (**Base type**)
`DataScope`	A string specifying whether the data type definition should be imported from, or exported to, a header file during code generation. (**Data scope**)
`Description`	A string that describes the usage of the data type. May be a null string. (**Description**)
`HeaderFile`	A string that specifies the name of a C header file from which a data type definition is imported, or to which a data type definition is exported, during code generation. (**Header file**)

See Also

"Generate Code Including User-Defined Data Types" in the Embedded Coder documentation

Simulink.Annotation

Purpose

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, the following 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
`Text`	String specifying text of annotation. Same as `Name`.	string
`ClickFcn`	Specifies MATLAB code to be executed when a user single-clicks this annotation. Simulink software stores the code entered in this field with the model. See "Associate Click Functions with Annotations" for more information.	string
`Description`	String that describes this annotation.	string

Property	Description	Values
`FontAngle`	String specifying the angle of the annotation's font. The default value, `'auto'`, specifies use of the model's preferred font angle.	`'normal' | 'italic' | 'oblique' | {'auto'}`
`FontName`	String specifying name of annotation's font. The default value, `'auto'`, specifies use of the model's preferred font.	string
`FontSize`	Integer specifying size of annotation's font in points. The default value, `-1`, specifies use of the model's preferred font size.	real `{'-1'}`
`FontWeight`	String specifying the weight of the annotation's font. The default value, `'auto'`, specifies use of the model's preferred font weight.	`'light' | 'normal' | 'demi' | 'bold' | {'auto'}`
`Handle`	Annotation handle.	real
`HiliteAncestors`	For internal use.	
`Name`	String specifying text of annotation. Same as `Text`.	string
`Selected`	String specifying whether this annotation is currently selected (`'on`') or not selected (`'off'`).	`{'on'} | 'off'`
`Parent`	String specifying parent name of annotation object.	string
`Path`	Path to the annotation.	string

Property	Description	Values
`Position`	Two-element vector specifying the x-y coordinates of this annotation relative to the top, left corner of the block diagram, e.g., `[236 83]`.	vector `[left bottom]` not enclosed in quotation marks. The maximum value for a coordinate is 32767.
`Horizontal-Alignment`	String specifying the horizontal alignment of this annotation, e.g., `'center'`.	`{'center'} | 'left'|'right'`
`VerticalAlignment`	String specifying the vertical alignment of this annotation, e.g., `'middle'`.	`{'middle'} | 'top'|'cap'|'baseline'|'bottom'`
`ForegroundColor`	String specifying foreground color of this annotation.	RGB value array string | `[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. Block background color can also be `'black'`, `'white'`, `'red'`, `'green'`, `'blue'`, `'cyan'`, `'magenta'`, `'yellow'`, `'gray'`, `'lightBlue'`, `'orange'`, `'darkGreen'`.

Property	Description	Values
`BackgroundColor`	String specifying background color of this annotation.	RGB value array string | `[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. Block background color can also be `'black'`, `'white'`, `'red'`, `'green'`, `'blue'`, `'cyan'`, `'magenta'`, `'yellow'`, `'gray'`, `'lightBlue'`, `'orange'`, `'darkGreen'`.
`DropShadow`	String specifying whether to display a drop shadow. Options are `'on'` or `'off'`.	`'on' | {'off'}`
`TeXMode`	String specifying whether to render TeX markup. Options are `'on'` or `'off'`.	`'on' | {'off'}`
`Type`	Annotation type. This is always `'annotation'`	string
`LoadFcn`	String specifying M-code to be executed when the model containing this annotation is loaded. See "Annotation Callback Functions" in the online Simulink documentation.	string

Property	Description	Values
`DeleteFcn`	String specifying M-code to be executed before deleting this annotation. See “Annotation Callback Functions” in the online Simulink documentation.	string
`RequirementInfo`	For internal use.	string
`Tag`	User-specified text that is assigned to the annotation’s Tag parameter and saved with the annotation.	string
`UseDisplayText-AsClickCallback`	String specifying whether to use the contents of the `Text` property as this annotation’s click function. Options are `'on'` or `'off'`. If set to `'on'`, the text of the annotation is interpreted as a valid MATLAB expression and run. If set to `'off'`, clicking on the annotation runs the click function, if there is one. If there is no click function, clicking the annotation has no effect. See “Associate Click Functions with Annotations” in the Simulink documentation for more information.	`'on' | {'off'}`
`UserData`	Any data that you want to associate with this annotation.	vector

Purpose

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 5-177 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 5-19	Usage type of this DWork vector.
"UsedAsDiscState"	True if this DWork vector is being used to store the values of a block's discrete states.

Properties

`Usage`

Description

Returns a string indicating how this DWork vector is used. Permissible values are:

- DWork
- DState
- Scratch
- Mode

Data Type

string

Access

`RW` for MATLAB S-function blocks, `RO` for other blocks.

`UsedAsDiscState`

Description

True if this DWork vector is being used to store the values of a block's discrete states.

Data Type

Boolean

Access

`RW` for MATLAB S-Function blocks, `RO` for other blocks.

Simulink.BlockCompInputPortData

Purpose

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 5-178 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”	True if this port has direct feedthrough.
“Overwritable”	True if this port is overwritable.

Properties

`DirectFeedthrough`

Description

True if this input port has direct feedthrough.

Data Type

Boolean

Access

`RW` for MATLAB S functions, `RO` for other blocks.

`Overwritable`

Description

True if this input port is overwritable.

Data Type

Boolean

Access

`RW` for MATLAB S functions, `RO` for other blocks.

Simulink.BlockCompOutputPortData

Purpose

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 5-179 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"	Specifies whether an output port's memory is reusable.

Properties

`Reusable`

Description

Specifies whether an output port's memory is reusable. Options are: `NotReusableAndGlobal` and `ReusableAndLocal`.

Data Type

string

Access

`RW` for MATLAB S functions, `RO` for other blocks.

Simulink.BlockData

Purpose

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 5-24	Fundamental base data type.
"AliasedThroughDataTypeID" on page 5-25	Fundamental base data type ID.
"Complexity"	Numeric type (real or complex) of the block data.
"Data"	The block data.
"DataAsDouble"	The block data in `double` form.
"Datatype"	Data type of the block data.
"DatatypeID"	Index of the data type of the block data.
"Dimensions"	Dimensions of the block data.
"Name"	Name of the block data.
"Type"	Type of block data (e.g., a parameter).

Properties

`AliasedThroughDataType`

Description

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.

Data Type

string

Access

`RO`

`AliasedThroughDataTypeID`

Description

Index of the data type alias returned by the `AliasedThroughDataType` property.

Data Type

integer

Access

`RO`

`Complexity`

Description

Numeric type (real or complex) of the block data.

Data Type

string

Access

`RW` for MATLAB S functions, `RO` for other blocks.

`Data`

Description

The block data.

Data Type

The data type specified by the “Datatype” or “DatatypeID” properties of this object.

Access

`RW`

`DataAsDouble`

Description

The block data's in `double` form.

Data Type

`double`

Access

`RO`

`Datatype`

Description

Data type of the values of the block-related object.

Data Type

string

Access

`RO`

`DatatypeID`

Description

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

Data Type	Value
`'int32'`	6
`'uint32'`	7
`'boolean'` or fixed-point data types	8

Data Type

integer

Access

`RW` for MATLAB S functions, `RO` for other blocks

`Dimensions`

Description

Dimensions of the block-related object, e.g., parameter or DWork vector.

Data Type

array

Access

`RW` for MATLAB S functions, `RO` for other blocks

`Name`

Description

Name of block-related object, e.g., a block parameter or DWork vector.

Data Type

string

Access

`RW` for MATLAB S functions, `RO` for other blocks

`Type`

Description

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.
`'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.

Data Type

string

Access

`RO`

Simulink.BlockPath

Purpose

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 “Set 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`	Individual component within the block specified by the block path

Method Summary

Name	Description
`BlockPath`	Create a block path.
`convertToCell`	Convert a block path to a cell array of strings.
`getBlock`	Get a single block path in the model reference hierarchy.
`getLength`	Get the length of the block path.
`validate`	Determine whether the block path represents a valid block hierarchy.

Properties

`SubPath`

Description

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'
```

Data Type

string

Access

`RW`

Methods

`BlockPath`

Purpose

Create block path

Syntax

```
blockpath_object = Simulink.BlockPath()
blockpath_object = Simulink.BlockPath(blockpath)
blockpath_object = Simulink.BlockPath(paths)
blockpath_object = Simulink.BlockPath(paths, subpath)
```

Input Arguments

`blockpath`

Block path object that you want to copy.

`paths`

A string or cell array of strings that Simulink uses to build the block path.

Specify each string in order, from the top model to the specific block for which you are creating a block path.

Each string 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 string, 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 string

When you create a block path for specifying Normal mode visibility:

- The first string must represent a block that is in the top model in the model reference hierarchy.
- Strings must represent Model blocks that are in Normal mode.
- Strings 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`

String that represents an individual component within a block.

Output Arguments

`blockpath_object`

Block path that you create.

Description

`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 strings that you specify with the `paths` argument. Each string represents a path at a level of model hierarchy. Simulink builds the full block path based on the strings.

`blockpath = Simulink.BlockPath(paths, subpath)` creates a block path from the string or cell array of strings that you specify with the `paths` argument and creates a path for the individual component (for example, a signal) of the block.

Example

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 strings 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`

Purpose

Convert block path to cell array of strings

Syntax

```
cellarray = Simulink.BlockPath.convertToCell()
```

Output Arguments

`cellarray`

  Cell array of strings representing elements of block path.

Description

`cellarray = Simulink.BlockPath.convertToCell()` converts a block path to a cell array of strings.

Examples

```
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`

Purpose

Get block path in model reference hierarchy

Syntax

```
block = Simulink.BlockPath.getBlock(index)
```

Input Arguments

`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.

Output Arguments

`block`

The block representing the level in the model reference hierarchy specified by the `index` argument.

Description

`blockpath = Simulink.BlockPath.getBlock(index)` returns the block path of the block specified by the `index` argument.

Example

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

Purpose

Get length of block path

Syntax

`length = Simulink.BlockPath.getLength()`

Output Arguments

`length`

The length of the block path. The length is the number of levels in the model reference hierarchy.

Description

`length = Simulink.BlockPath.getLength()` returns a numeric value that corresponds to the number of levels in the model reference hierarchy for the block path.

Example

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`

Purpose

Determine whether block path represents valid block hierarchy

Syntax

```
Simulink.BlockPath.validate()
Simulink.BlockPath.validate(AllowInactiveVariant)
```

Input Arguments

`AllowInactiveVariant`

Set to `true` to include inactive variants in the validity checking. The default is `false`.

Description

`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.
- All variant elements are active.
- Each element except for the last element:
  - Is a valid Model block
  - References the model of the next element

`Simulink.BlockPath.validate(AllowInactiveVariant)` Specifying `true` causes the validity checking to consider inactive variants as being valid, if they meet the other validity checks described above. Omitting the `AllowInactiveVariant` argument or specifying its default value of `false` causes the method to check only the active variant.

Example

Validate the block paths, checking only the active variant. This validation fails, because the block path actually references model `sldemo_mrv_nonlinear_controller`, while `bp` specifies that the block references model `sldemo_mdlref_second_order_controller`, which is in an inactive variant.

```
sldemo_mdlref_variants
bp = Simulink.BlockPath({'sldemo_mdlref_variants/Controller', ...
'sldemo_mrv_second_order_controller/sensor1'})
bp.validate()
```

Validate by checking all variants. The block path passes the validation when inactive variants are also checked.

```
bp.validate(true)
```

See Also

`Simulink.SimulationData.BlockPath` |
`Simulink.SimulationData.Dataset`

How To

- "Specify the Instance That Has Normal Mode Visibility"

Purpose

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"	True if this port is connected to a bus.
"IsSampleHit"	True if this port produces output or accepts input at the current simulation time step.
"SampleTime"	Sample time of this port.
"SampleTimeIndex"	Sample time index of this port.
"SamplingMode"	Sampling mode of the port.

Properties

`IsBus`

Description

True if this port is connected to a bus.

Data Type

Boolean

Access

`RO`

`IsSampleHit`

Description

True if this port produces output or accepts input at the current simulation time step.

Data Type

Boolean

Access

`RO`

`SampleTime`

Description

Sample time of this port.

Data Type

`[period offset]` where `period` and `offset` are values of type `double`. See “Specify Sample Time” for more information.

Access

`RW` for MATLAB S functions, `RO` for other blocks

`SampleTimeIndex`

Description

Sample time index of this port.

Data Type

integer

Access

`RO`

`SamplingMode`

Description

Sampling mode of the port. Valid values are:

Value	Description
`'frame'`	Port accepts or outputs frame-based signals. The use of frame-based signals requires a DSP System Toolbox license.
`'inherited'`	Sampling mode is inherited from the port to which this port is connected.
`'sample'`	Port accepts or outputs sampled data.

Data Type

string

Access

`RW` for MATLAB S functions, `RO` for other blocks

Simulink.BlockPreCompInputPortData

Purpose

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 5-178 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"	True if this port has direct feedthrough.
"Overwritable"	True if this port is overwritable.

Properties

`DirectFeedthrough`

Description

True if this input port has direct feedthrough.

Data Type

Boolean

Access

`RW` for MATLAB S functions, `RO` for other blocks

`Overwritable`

Description

True if this input port is overwritable.

Data Type

Boolean

Access

`RW` for MATLAB S functions, `RO` for other blocks

Purpose

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 5-179 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”	Specifies whether an output port’s memory is reusable.

Properties

`Reusable`

Description

Specifies whether an output port’s memory is reusable. Options are: `NotReusableAndGlobal` and `ReusableAndLocal`.

Data Type

string

Access

`RW` for MATLAB S functions, `RO` for other blocks

Purpose

Specify properties of signal bus

Description

Objects of this class (in conjunction with objects of the `Simulink.BusElement` class) specify the properties of a signal bus. Use bus objects to enable Simulink software to validate the properties of buses connected to the inputs of blocks in your model. You do this by entering, in the **Data type** parameter of a block parameter dialog box, the name of a bus object that defines a bus. When you update the model diagram or start a simulation of the model, 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.

The blocks that support using a bus object as a data type are:

- Bus Creator
- Constant
- Data Store Memory
- Data Store Read
- Data Store Write
- Inport
- Outport
- Signal Specification

You can use the Model Explorer **Add > Simulink Bus** command (see “Using the Model Explorer to Create Data Objects”), the Simulink Bus editor (see “Manage Bus Objects with the Bus Editor”), or MATLAB commands (see “Data Objects”) to create bus objects in the base MATLAB workspace. You must use the Bus editor or the MATLAB command line to set the properties of a bus object. Simulink also provides a set of utility functions for creating and saving bus objects.

To view bus object properties:

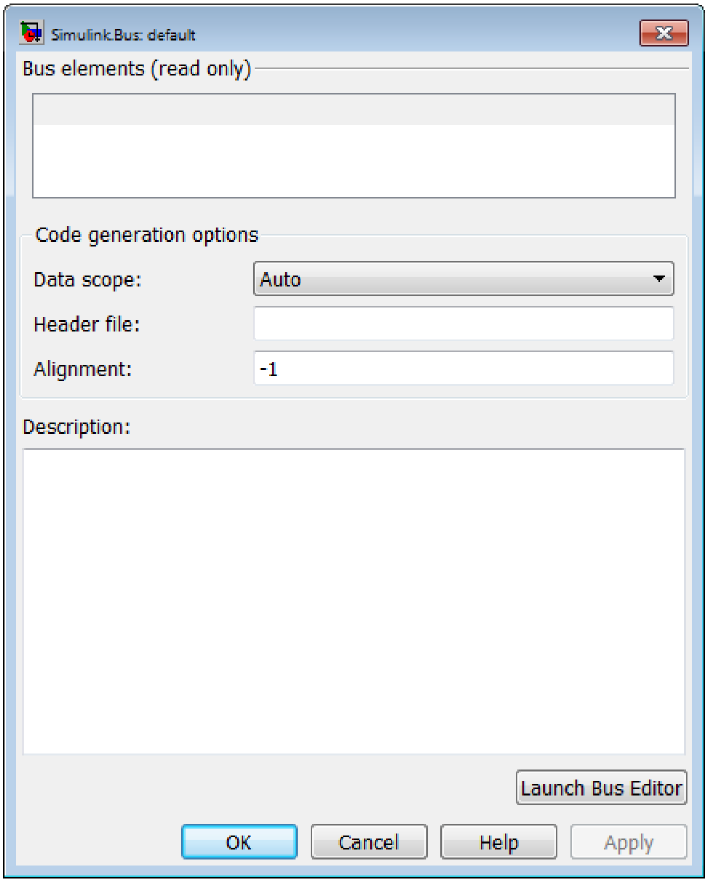
1. Open the Model Explorer.

2 In the **Model Hierarchy** pane, select the Base Workspace node.

3 In the **Contents** pane, select the bus object.

In the **Property** dialog box appears.

Property Dialog Box


Simulink.Bus: default
Bus elements (read only)
Code generation options
Description:
Launch Bus Editor
OK
Cancel
Help
Apply

Data scope: Auto
Header file:
Alignment: -1

Bus elements

Table that displays the properties of the bus elements. You cannot edit this table. You must use either the Simulink **Bus editor** (see "Bus Objects") or MATLAB commands to add or delete bus elements or change the properties of existing bus elements. To launch the bus editor, click the **Launch Bus editor** button at the bottom of this dialog box or in the Simulink Editor, select **Edit > Bus EditorBus editor**.

Data scope

Specifies whether the data type definition should be imported from, or exported to, a header file during code generation. The possible values are:

Value	Action
`Auto` (default)	If no value is specified for **Header file**, export the data type definition to *`model`*`_types.h`, where *`model`* is the model name. If a value is specified for **Header file**, import the data type definition from the specified header file.
`Exported`	Export the data type definition to a header file, which can be specified in the **Header file** field. If no value is specified for **Header file**, the header file name defaults to *`type`*`.h`, where*`type`* is the data type name.
`Imported`	Import the data type definition from a header file, which can be specified in the **Header file** field. If no value is specified for **Header file**, the header file name defaults to *`type`*`.h`, where*`type`* is the data type name.

Header file

Name of a C header file from which a data type definition is imported, or to which a data type definition is exported, based on the value of **Data scope**. This field is intended for use by Simulink Coder software. See "Generate Code Including User-Defined Data Types". Simulink software ignores this field.

Alignment

Data alignment boundary, specified in number of bytes. The starting memory address for the data allocated for the bus will be a multiple of the **Alignment** setting. The default value is `-1`, which specifies that the code generator should determine an optimal alignment based on usage. Otherwise, specify a positive integer that is a power of 2, not exceeding 128. This field is intended for use by Simulink Coder software. See "Configure Data Alignment for Function Implementations". Simulink software ignores this field.

Description

Description of this structure. This field is intended for you to use to document this bus. Simulink software does not use this field.

Properties

Name	Access	Description
`Alignment`	`RW`	Integer value specifying a data alignment boundary, in number of bytes. This property is intended for use by Simulink Coder software. Simulink software does not use it. (Alignment)
`DataScope`	`RW`	A string specifying whether the data type definition should be imported from, or exported to, a header file during code generation. (Data scope)
`Description`	`RW`	A string that describes this bus. This property is intended for user use. Simulink software does not use it. (Description)

Name	Access	Description
`Elements`	`RW`	An array of `Simulink.BusElement` objects that define the names, data types, dimensions, and other properties of the bus's elements. The elements must have unique names (see "Choosing a Signal Name"). (Bus elements)
`HeaderFile`	`RW`	A string that specifies the name of a C header file from which a data type definition is imported, or to which a data type definition is exported, during code generation. This property is intended for use by Simulink Coder software. Simulink software does not use it. (Header file)

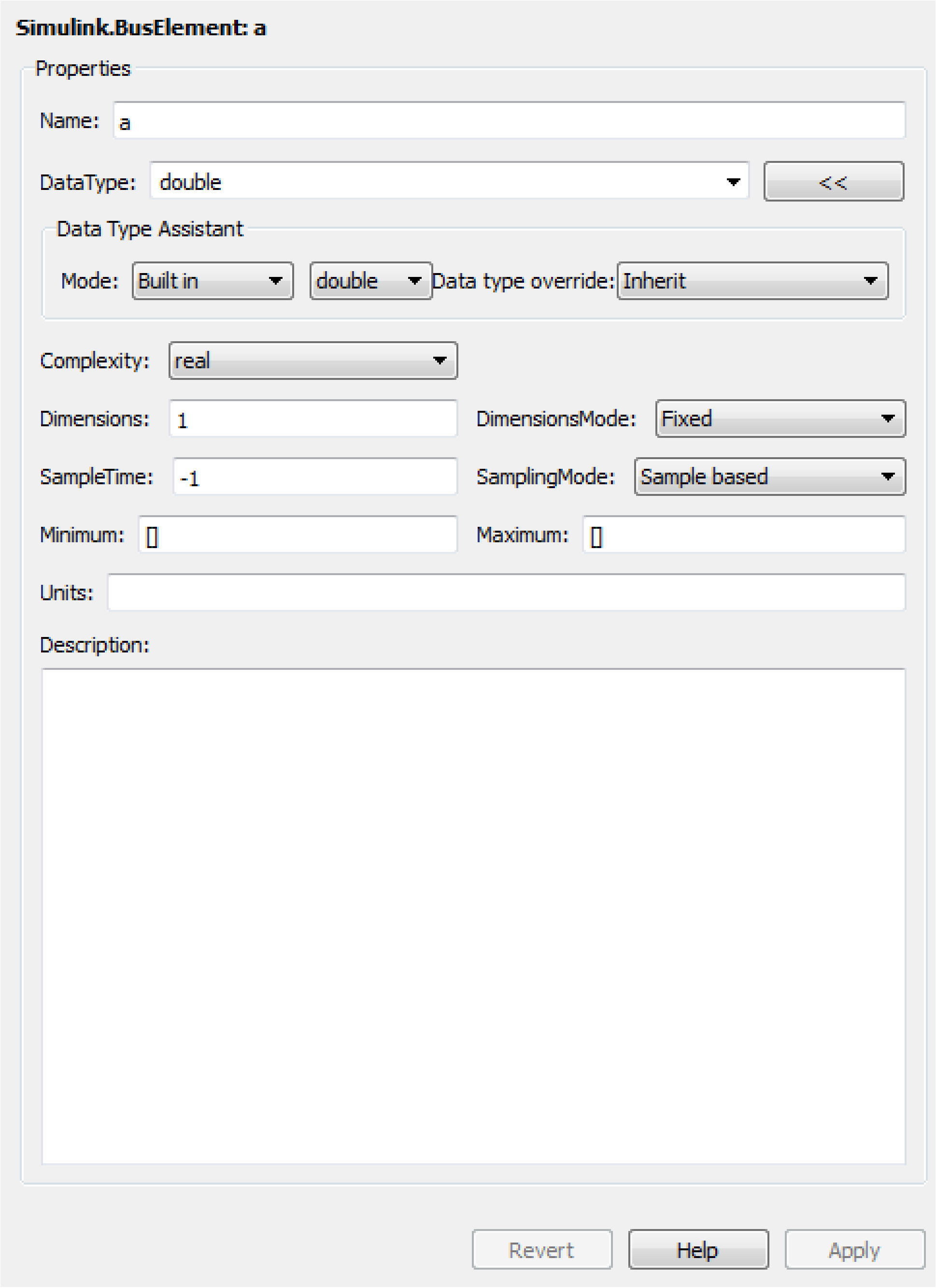
See Also

"Composite Signals", Bus Assignment, Bus Creator, Bus Selector, Bus to Vector, Constant, Inport, Outport, Signal Specification, `Simulink.Bus.cellToObject`, `Simulink.Bus.createMATLABStruct`, `Simulink.Bus.createObject`, `Simulink.BusElement`, `Simulink.Bus.objectToCell`, `Simulink.Bus.save`

Purpose Describe element of signal bus

Description Objects of this class define elements of buses defined by objects of the `Simulink.Bus` class.

Property Dialog Box



Simulink.BusElement: a

Properties

Name: a

DataType: double

Data Type Assistant

Mode: Built in double Data type override: Inherit

Complexity: real

Dimensions: 1

DimensionsMode: Fixed

SampleTime: -1

SamplingMode: Sample based

Minimum: []

Maximum: []

Units:

Description:

`Name`

Name of this element. See “Signal Names” for guidelines for signal names.

`Data Type:` string

`Access:` RW

`DataType`

Name of the data type of this element. The value of this field can be a:

- Built-in Simulink data type (for example, `double` or `uint8`)
- `Simulink.NumericType` object, with one exception. The exception is a `Simulink.NumericType` whose `Category` is `Fixed-point: unspecified scaling`.

> **Note** `Fixed-point:  unspecified scaling` is a partially specified type whose definition is completed by the block that uses the `Simulink.NumericType`. Forbidding its use for bus elements avoids creating bus elements that have different data types depending on where they are used.

- `Simulink.Bus` object, using the `Bus:  <object name>` option. This allows you to create bus objects that specify hierarchical buses (that is, buses that contain other buses).

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Data type** parameter. (See “Specify Data Types Using Data Type Assistant”.)

`Data Type:` string

`Access:` RW

`Dimensions`

A vector specifying the dimensions of this element.

`Data Type:` array.

`Access:` RW

`Complexity`

Numeric type (`real` or `complex`) of this element. Must be `real` if this bus element is itself a bus.

`Data Type:` string

`Access:` RW

`SampleTime`

Size of the interval between times when this signal value must be recomputed. Must be `-1 (inherited)` if this bus element is itself a bus or if the bus that includes this element passes through a block that changes the bus's sample time, such as a Rate Transition block. See "Specify Sample Time" for more information.

`Data Type:` double

`Access:` RW

`Min`

The minimum value of this element. This value must be a finite real double scalar. This value must be empty `[]` if this element is itself a bus.

`Data Type:` double

`Access:` RW

`Max`

The maximum value of this element. This value must be a finite real double scalar. This value must be empty `[]` if this element is itself a bus.

`Data Type:` double

`Access:` RW

`SamplingMode`

Sampling mode of this element. Must be sample-based if this element is itself a bus. This field is intended to be used by applications based on Simulink models.

`Data Type:` string

`Access:` RW

`DimensionsMode`

A field that specifies if the size (the number of elements in a dimension) of this element may vary or remain fixed during simulation. This field can have the following values:

- `Fixed:` The size of the element may not change during simulation.
- `Variable:` The size of the element may change during simulation.

See “Variable-Size Signal Basics” for more information.

`Data Type:` string

`Access:` RW

`Units`

Measurement units in which this value is expressed (for example, inches). This field is intended for use in documenting this parameter. Simulink ignores it.

`Data Type:` string

`Access:` RW

`Description`

Description of the bus element. This field is intended for use in documenting this parameter. Simulink ignores it.

`Data Type:` string

`Access:` RW

See Also

"Composite Signals", Bus Assignment, Bus Creator, Bus Selector, Bus to Vector, `Simulink.Bus`, `Simulink.Bus.cellToObject`, `Simulink.Bus.createObject`, `Simulink.Bus.objectToCell`, `Simulink.Bus.save`

Purpose

Access model configuration set

Description

Instances of this handle class allow you to write programs to create, modify, and attach configuration sets to models. See “Manage a Configuration Set” and “Overview” for more information.

Property Summary

Name	Description
“Components”	Components of the configuration set.
“Description”	Description of the configuration set.
“Name”	Name of the configuration set.

Note You can use the **Model Configuration** dialog box to set the `Name` and `Description` properties of a configuration set. See “Model Configuration Pane” for more information.

Method Summary

Name	Description
“attachComponent”	Attach a component to a configuration set.
“copy”	Create a copy of a configuration set.
“getComponent”	Get a component of a configuration set.
“getFullName”	Get the full path of a configuration set.
“getModel”	Get the handle of the model that owns a configuration set.
“get_param”	Get the value of a configuration set parameter.
“isActive”	Determine whether a configuration set is the active set of the model that owns it.
“isValidParam”	Determine whether a specified parameter is a valid parameter of a configuration set.

Name	Description
“saveAs”	Save a configuration set to a MATLAB file.
“setPropEnabled”	Prevent or allow a user to change a parameter.
“set_param”	Set the value of a configuration set parameter.

Properties

`Components`

Description

Array of `Simulink.ConfigComponent` objects representing the components of the configuration set. For example, solver parameters and data import/export parameters.

Data Type

array

Access

`RW`

`Description`

Description

Description of the configuration set. You can use this property to provide additional information about a configuration set, such as its purpose. This field can remain blank.

Data Type

`string`

Access

`RW`

`Name`

Description

Name of the configuration set. This name represents the configuration set in the Model explorer.

Data Type

`string`

Access

`RW`

Methods

`attachComponent`

Purpose

Attach a component to this configuration set.

Syntax

`attachComponent(`*`component`*`)`

Arguments

`component`

Instance of `Simulink.ConfigComponent` class.

Description

This method replaces a component in this configuration set with a component having the same name.

example

The following example replaces the solver component of the active configuration set of model A with the solver component of the active configuration set of model B.

```
hCs = getActiveConfigSet('B');
hSolverConfig = hCs.getComponent('Solver');
hSolverConfig = hSolverConfig.copy;
hCs = getActiveConfigSet('A');
hCs.attachComponent(hSolverConfig);
```

`copy`

Purpose

Create a copy of this configuration set.

Syntax

`copy`

Description

This method creates a copy of this configuration set.

Note You must use this method to create copies of configuration sets because `Simulink.ConfigSet` is a handle class. See "Handle Versus Value Classes" for more information.

`getComponent`

Purpose

Get a component of this configuration set.

Syntax

`getComponent(`*`componentName`*`)`

Arguments

`componentName`
String specifying the name of the component to be returned.

Description

Returns the specified component. Omit the argument to get a list of the names of the components that this configuration set contains.

Example

The following code gets the solver component of the active configuration set of the currently selected model.

```
hCs = getActiveConfigSet(gcs);
hSolverConfig = hCs.getComponent('Solver');
```

The following code displays the names of the components of the currently selected active configuration set of the model at the MATLAB command line.

```
hCs = getActiveConfigSet(gcs);
hCs.getComponent
```

`getFullName`

Purpose

Get the full path of a configuration set.

Syntax

`getFullName`

Description

This method returns a string specifying the full path of a configuration set, for example, `'vdp/Configuration'`.

`getModel`

Purpose

Get the model that owns this configuration set.

Syntax

`getModel`

Description

Returns a handle to the model that owns this configuration set.

Example

The following command opens the block diagram of the model that owns the configuration set referenced by the MATLAB workspace variable `hCs`.

```
open_system(hCs.getModel);
```

`get_param`

Purpose

Get the value of a configuration set parameter.

Syntax

get_param(*paramName*)

Arguments

`paramName`

String specifying the name of the parameter whose value is to be returned.

Description

This method returns the value of the specified parameter. Specifying *`paramName`* as `'ObjectParameters'` returns the names of the valid parameters in the configuration set.

Example

The following command gets the name of the solver used by the selected active configuration of the model.

```
hAcs = getActiveConfigSet(bdroot);
hAcs.get_param('SolverName');
```

Note You can also use the `get_param` model construction command to get the values of parameters of an active configuration set of a model. For example, `get_param(bdroot, 'SolverName')` gets the solver name of the currently selected model.

`isActive`

Purpose

Determine whether configuration set is the active configuration set for the model.

Syntax

`isActive`

Description

Returns `true` if this configuration set is the active configuration set of the model that owns this configuration set.

`isValidParam`

Purpose

Determine whether a specified parameter is a valid parameter of this configuration set. A parameter is valid if it is compatible with other parameters in the configuration set. For example, if `SolverType` is `'variable-step'`, `FixedStep` is an invalid parameter.

Syntax

`isValidParam(`*`paramName`*`)`

Arguments

`paramName`

String specifying the name of the parameter whose validity is to be determined.

Description

This method returns true if the specified parameter is a valid parameter of this configuration set; otherwise, it returns false.

Example

The following code sets the parameter `StopTime` only if it is a valid parameter of the currently selected active configuration set.

```
hAcs = getActiveConfigSet(gcs);
if hAcs.isValidParam('StopTime')
 set_param('StopTime', '20');
end
```

`saveAs`

Purpose

Save configuration set to MATLAB file

Syntax

`saveAs(`*`fileName`*`)`

`saveAs(`*`fileName`*`, '`*`paramName`*`', `*`paramValue`*`)`

Arguments

`fileName`

String specifying the name of the MATLAB file that the method creates.

`paramName`, *`paramValue`*

Parameter name and value pairs that you optionally use to format the MATLAB file.

Name	Value
`'-format'`	• `'MATLAB function'` (default) — Creates a MATLAB function. • `'MATLAB script'` — Creates a MATLAB script.
`'-comments'`	• `'on'` (default) — The MATLAB file: – Includes the GUI name of the parameter as a comment to help identify the parameters. – Groups the parameters by the pane in the Configuration Parameters dialog box on which they are displayed. • `'off'` — Does not include comments in the MATLAB file, so that the file generates faster.
`'-varname'`	`'`*`variable`*`'` — Any valid variable name. If you do not specify this parameter, the

Name	Value
	MATLAB script uses `cs` for the variable. When you specify `'-format', 'MATLAB script'`, use `'-varname', '`*`variable`*`'` to specify the variable that the script uses for the configuration set object. When you run the script, the script creates the variable in the base workspace.

Description

`saveAs(`*`fileName`*`)` saves a configuration set to a MATLAB file. Before saving, you must get a handle to the configuration set. Use *`fileName`* to specify the file name.

`saveAs(`*`fileName`*`, '`*`paramName`*`', `*`paramValue`*`)` accepts one or more comma-separated parameter name and value pairs. For the valid parameter name and value pairs, see the previous arguments section.

Example

The following code gets the configuration set for `sldemo_counters` and creates a function called `ConfiguredDataFunction`.

```
% Get the active configuration set from sldemo_counters.
hCs = getActiveConfigSet('sldemo_counters');
% Save the configuration set as a function.
hCs.saveAs('ConfiguredDataFunction');
```

The following code gets the configuration set for `sldemo_counters` and creates a script called `ConfiguredDataScript`. The script uses `config_set` for the variable name.

```
% Get the active configuration set from sldemo_counters.
hCs = getActiveConfigSet('sldemo_counters');
```

```
% Save the configuration set as a script.
hCs.saveAs('ConfiguredDataScript', '-format', 'MATLAB script', '-varname', 'config_set');
```

See Also

`getActiveConfigSet`, `getConfigSet`

`setPropEnabled`

Purpose

Enable a configuration set parameter to be changed.

Syntax

`setPropEnabled(`*`paramName`*`, `*`isenabled`*`)`

Arguments

`paramName`

Name of the parameter whose value is to be set.

`isenabled`

Specify as `true` to enable parameter; as `false`, to disable the parameter.

Description

This method sets the enabled status the parameter specified by `paramName` to the value specified by `isenabled`. Disabling a parameter prevents the user from changing it.

Example

The following code prevents the user from setting the simulation stop time of the currently selected model.

```
hAcs = getActiveConfigSet(gcs);
hAcs.setPropEnabled('StopTime', false);
```

`set_param`

Purpose

Set the value of a configuration set parameter.

Syntax

`set_param(paramName, paramValue)`

Arguments

`paramName`
Name of the parameter whose value is to be set.

`paramValue`
Value to assign to the parameter.

Description

This method sets the configuration set parameter specified by `paramName` to the value specified by `paramValue`.

Example

The following command sets the simulation stop time of the selected active configuration set.

```
hAcs = getActiveConfigSet(gcs);
hAcs.set_param('StopTime', '20');
```

Note You can also use the `set_param` model construction command to set the parameters of the active configuration set. For example, `set_param(gcs, 'StopTime', '20')` sets the simulation stop time of the currently selected model.

See Also

"About Configuration Sets"

Purpose

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”	Description of the configuration reference.
“Name”	Name of the configuration reference.
“WSVarName”	Name of the workspace variable that contains the referenced configuration set.

Note You can use the **Configuration Reference** dialog box to set the `Name`, `Description`, and `WSVarName` properties of a configuration reference. See “Create and Attach a Configuration Reference” for details.

Method Summary

Name	Description
“copy”	Create a copy of a configuration reference.
“getFullName”	Get the full pathname of a configuration reference.
“getModel”	Get the handle of the model that owns a configuration reference.
“get_param”	Get the value of a configuration set parameter indirectly through a configuration reference.
“getRefConfigSet”	Get the configuration set specified by a configuration reference.

Name	Description
“isActive”	Determine whether a configuration reference is the active configuration object of the model.
“refresh”	Update configuration reference after any change to properties or configuration set availability.

Properties

`Description`

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.

Data Type

`string`

Access

`RW`

`Name`

Description

Name of the configuration reference. This name represents the configuration reference in the GUI.

Data Type

`string`

Access

`RW`

`WSVarName`

Description

Name of the workspace variable that contains the referenced configuration set.

Data Type

`string`

Access

`RW`

Methods

`copy`

Purpose

Create a copy of this configuration reference.

Syntax

`copy`

Description

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`

Purpose

Get the full pathname of a configuration reference.

Syntax

`getFullName`

Description

This method returns a string specifying the full pathname of a configuration reference, e.g., `'vdp/Configuration'`.

`getModel`

Purpose

Get the model that owns this configuration reference.

Syntax

`getModel`

Description

Returns a handle to the model that owns this configuration reference.

example

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`

Purpose

Get the value of a configuration set parameter indirectly through a configuration reference.

Syntax

`get_param(`*`paramName`*`)`

Arguments

`paramName`

String specifying the name of the parameter whose value is to be returned.

Description

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 `WSVarName` 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.

example

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`

Purpose

Get the configuration set specified by a configuration reference

Syntax

`getRefConfigSet`

Description

Returns a handle to the configuration set specified by the `WSVarName` property of a configuration reference.

`isActive`

Purpose

Determine whether this configuration set is its model's active configuration set.

Syntax

`isActive`

Description

Returns `true` if this configuration set is the active configuration set of the model that owns this configuration set.

`refresh`

Purpose

Update configuration reference after any change to properties or configuration set availability

Syntax

```
refresh
```

Description

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 `WSVarName`. If you omit executing `refresh` after any such change, the configuration reference handle will be stale, and using it will give incorrect results.

Simulink.GlobalDataTransfer

Purpose

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)'`

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)'`

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 the extrapolation method of data transfer to configure continuous-to-continuous task transitions.

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 in the MATLAB documentation.

Examples

Access the properties of this class.

```
dt=get_param(gcs,'DataTransfer');
dt.DefaultTransitionBetweenContTasks

ans =

Ensure deterministic transfer (minimum delay)
```

Alternatives “Customize Concurrent Execution Settings”

See Also `Simulink.SoftwareTarget.TaskConfiguration`

How To

- “Configuring Data Transfer Communications”

Purpose

Extract model file information without loading block diagram into memory

Description

The class Simulink.MDLInfo extracts information from a model file without loading the block diagram into memory.

You can create an `MdlInfo` object containing all the model information properties, or you can use the static methods for convenient access to individual properties without creating the class first. For example, to get the description only:

```
description = Simulink.MDLInfo.getDescription('mymodel')
```

To get the metadata only:

```
metadata = Simulink.MDLInfo.getMetadata('mymodel')
```

All model information properties are read only.

Construction

`info = Simulink.MDLInfo('`*`mymodel`*`')` creates an instance of the `MdlInfo` class `info` and populates the properties with the information from the model file `'`*`mymodel`*`'`.

`mymodel` can be:

- A block diagram name (for example, `vdp`)
- The file name for a file on the MATLAB path (for example, `mymodel.slx`)
- A file name relative to the current folder (for example, `mydir/mymodel.slx`)
- A fully qualified file name (for example, `C:\mydir\mymodel.slx`)

`Simulink.MDLInfo` resolves the supplied name by looking at files on the MATLAB path, and ignores any block diagrams in memory. This may cause unexpected results if you supply the name of a loaded model, but its file is shadowed by another file on the MATLAB path. If a file is shadowed, you see a warning in the command window. To avoid any confusion, supply a fully-qualified file name to `Simulink.MDLInfo`.

Properties

BlockDiagramName

Name of block diagram.

Description

Description of model.

FileName

Name of model file.

Interface

Names and attributes of the block diagram's root inports, outports, model references, etc., describing the graphical interface if you created a Model Reference block from this model.

Structure.

IsLibrary

Whether the block diagram is a library.

Metadata

Names and attributes of arbitrary data associated with the model.

Structure. The structure fields can be strings, numeric matrices of type "double", or more structures. Use the method `getMetadata` to extract this metadata structure without loading the model.

ModelVersion

Model version number.

SimulinkVersion

Version number of Simulink software that was used to save the model file.

Methods

getDescription	Extract file description without loading block diagram into memory
getMetadata	Extract file metadata without loading block diagram into memory

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects 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

Purpose

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

Purpose

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 strings, 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`

Purpose

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 string that identifies 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. See the `Simulink.ModelAdvisor` "Method Summary" for more information.

You can also determine check IDs as follows:

1. In the Model Advisor, select **View > Source Tab**.

2. Navigate to the folder that contains the check.

3. In the right pane, click **Source**. The Model Advisor displays the **Title**, **TitleID**, and **Source** information for each check in the folder.

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”	Close Model Advisor report.
“deselectCheck”	Clear checks.
“deselectCheckAll”	Clear all checks.
“deselectCheckForGroup”	Clear a group of checks.
“deselectCheckForTask”	Clear checks that belong to a specified task or set of tasks.
“deselectTask”	Clear tasks.
“deselectTaskAll”	Clear all tasks.
“displayReport”	Display Model Advisor report.

Name	Description
“exportReport”	Copy report to a specified location.
“filterResultWithExclusion” on page 5-91	Filter objects that have been excluded by user-defined exclusions.
“getBaselineMode”	Get baseline mode setting for the Model Advisor.
“getCheckAll”	Get the IDs of the checks performed by the Model Advisor.
“getCheckForGroup”	Get checks belonging to a check group.
“getCheckForTask”	Get checks belonging to a task.
“getCheckResult”	Get check results.
“getCheckResultData”	Get check result data.
“getCheckResultStatus”	Get pass/fail status of a check or set of checks.
“getGroupAll”	Get the IDs of the groups of tasks performed by the Model Advisor.
“getInputParameters” on page 5-96	Get input parameters of a check.
“getListViewParameters” on page 5-97	Get list view parameters of a check.
“getModelAdvisor”	Get the Model Advisor for a model or subsystem.

Name	**Description**
“getSelectedCheck”	Get selected checks.
“getSelectedSystem” on page 5-99	Get path of system currently targeted by the Model Advisor.
“getSelectedTask”	Get selected tasks.
“getTaskAll”	Get the IDs of the tasks performed by the Model Advisor.
“Simulink.ModelAdvisor.openConfigUI” on page 5-100	Start the Model Advisor Configuration editor.
“Simulink.ModelAdvisor.reportexists”	Determine whether a report exists for a system or subsystem.
“runCheck”	Run selected checks.
“runTask”	Run checks for selected tasks.
“selectCheck”	Select checks.
“selectCheckAll”	Select all checks.
“selectCheckForGroup”	Select a group of checks.
“selectCheckForTask”	Select checks that belong to a specified task.
“selectTask”	Select tasks.
“selectTaskAll”	Select all tasks.
“setActionenable” on page 5-105	Set enable/disable status for a check action.
“setBaselineMode”	Set baseline mode for the Model Advisor.
“setCheckErrorSeverity” on page 5-107	Set severity of a check failure.

Name	Description
“setCheckResult”	Set result for the currently running check.
“setCheckResultData”	Set result data for the currently running check.
“setCheckResultStatus”	Set pass/fail status for the currently running check.
“setListViewParameters” on page 5-110	Set list view parameters for a check.
“verifyCheckRan”	Verify that checks have run.
“verifyCheckResult”	Generate a baseline set of check results or compare the current set of results to the baseline results.
“verifyCheckResultStatus”	Verify that a model has passed or failed a set of checks.
“verifyHTML”	Generate a baseline report or compare the current report to a baseline report.

Methods

closeReport

Purpose

Close Model Advisor report

Syntax

`closeReport`

Description

Closes the report associated with this Model Advisor object, which closes the Model Advisor window.

See Also

“displayReport”

deselectCheck

Purpose

Clear check

Syntax

`success = deselectCheck(ID)`

Arguments

`ID`

String or cell array that specifies the IDs of the checks to be cleared.

`success`

True (1) if the check is cleared.

Description

This method clears the checks specified by `ID`.

Note This method cannot clear disabled checks.

See Also

"getCheckAll", "deselectCheckForGroup", "selectCheck"

deselectCheckAll

Purpose

Clear all checks

Syntax

`success = deselectCheckAll`

Arguments

`success`

True (1) if all checks are cleared.

Description

Clears all checks that are not disabled.

See Also

"selectCheckAll"

deselectCheckForGroup

Purpose

Clear group of checks

Syntax

`success = deselectCheckForGroup(groupName)`

Arguments

`groupName`
: String 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.

Description

Clears a specified group of checks.

See Also

"selectCheckForGroup"

deselectCheckForTask

Purpose

Clear checks that belong to specified task or set of tasks

Syntax

`success = deselectCheckForTask(ID)`

Arguments

`ID`
: String or cell array of strings that specify the IDs of tasks whose checks are to be cleared.

`success`
: True (1) if the specified tasks are cleared.

Description

Clears checks belonging to the tasks specified by the *`ID`* argument.

See Also

“getTaskAll”, “selectCheckForTask”

deselectTask

Purpose

Clear task

Syntax

`success = deselectTask(ID)`

Arguments

`ID`
: String or cell array that specifies the ID of tasks to be cleared

`success`
: True (1) if the method succeeded in clearing the specified tasks.

Description

Clears the tasks specified by *`ID`*.

See Also

“selectTask”, “getTaskAll”

deselectTaskAll

Purpose

Clears all tasks

Syntax

`success = deselectTaskAll`

Arguments

`success`
: True (1) if this method succeeds in clearing all tasks.

Description

Clears all tasks.

See Also

“selectTaskAll”

displayReport

Purpose

Display report in Model Advisor window

Syntax

`displayReport`

Description

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.

See Also

“Simulink.ModelAdvisor.reportexists”

exportReport

Purpose

Create copy of report generated by Model Advisor

Syntax

`[success message] = exportReport(`*`destination`*`)`

Arguments

`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.

Description

This method creates a copy of the last report generated by the Model Advisor and stores the copy at the specified location.

See Also

"Simulink.ModelAdvisor.reportexists"

filterResultWithExclusion

Purpose

Filter objects that have been excluded by user-defined exclusions.

Syntax

```
filteredResultHandles =
obj.filterResultWithExclusion(ResultHandles)
```

Arguments

`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.

Description

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 Verification and Validation™ .

See Also

"getModelAdvisor" on page 5-98

getBaselineMode

Purpose

Determine whether Model Advisor is in baseline data generation mode

Syntax

```
mode = getBaselineMode
```

Arguments

`mode`
: Boolean value indicating baseline mode.

Description

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", to generate baseline data.

See Also

"setBaselineMode", "verifyHTML", "verifyCheckResult", "verifyCheckResultStatus"

getCheckAll

Purpose

Get IDs of all checks

Syntax

```
IDs = getCheckAll
```

Arguments

`IDs`
: Cell array of strings specifying the IDs of all checks performed by the Model Advisor.

Description

Returns a cell array of strings specifying the IDs of all checks performed by the Model Advisor.

See Also

"getTaskAll", "getGroupAll"

getCheckForGroup

Purpose

Get checks that belong to check group

Syntax

`IDs = getCheckForTask(groupName)`

Arguments

`groupName`
String specifying the name of a group.

`IDs`
Cell array of IDs.

Description

Returns a cell array of IDs of the tasks and checks belonging to the group specified by *`groupName`*.

See Also

"getCheckForTask"

getCheckForTask

Purpose

Get checks that belong to task

Syntax

`checkIDs = getCheckForTask(taskID)`

Arguments

`taskID`
ID of a task.

`checkIDs`
Cell array of IDs of checks belonging to the specified task.

Description

Returns a cell array of IDs of the checks belonging to the task specified by *`taskID`*.

See Also

"getCheckForGroup"

getCheckResult

Purpose

Get results of running check or set of checks

Syntax

`result = getCheckResult(ID)`

Arguments

`ID`

ID of a check or cell array of check IDs.

`result`

A check result or cell array of check results.

Description

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 Verification and Validation software. For more information, see the Simulink Verification and Validation documentation.

See Also

"getCheckResultData", "getCheckResultStatus"

getCheckResultData

Purpose

Get data resulting from running check or set of checks

Syntax

`result = getCheckResultData(ID)`

Arguments

`ID`

Check ID or cell array of check IDs.

`result`
: Data from a check result or cell array of data from check results.

Description

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 Verification and Validation software. For more information, see the Simulink Verification and Validation documentation.

See Also

"getCheckResult", "getCheckResultStatus"

getCheckResultStatus

Purpose

Get status of check or set of checks

Syntax

`result = getCheckResultStatus(ID)`

Arguments

`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.

Description

Invoke this method after running a set of checks to determine whether the checks passed or failed.

See Also

"getCheckResult", "getCheckResultData"

getGroupAll

Purpose

Get all groups of checks run by Model Advisor

Syntax

`IDs = getGroupAll`

Arguments

`IDs`
: Cell array of IDs of all groups of checks run by the Model Advisor.

Description

Returns a cell array of IDs of all groups of checks run by the Model Advisor.

See Also

"getCheckAll", "getTaskAll"

getInputParameters

Purpose

Get input parameters of check

Syntax

`params = obj.getInputParameters(check_ID)`

Arguments

`params`

A cell array of `ModelAdvisor.InputParameter` objects.

`obj`

A variable representing the `Simulink.ModelAdvisor` object.

`check_ID`

A string that uniquely identifies the check.

You can omit the *`check_ID`* if you use the method inside a check callback function.

Description

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 Verification and Validation software. For more information, see the Simulink Verification and Validation documentation.

See Also

`ModelAdvisor.InputParameter`

getListViewParameters

Purpose

Get list view parameters of check

Syntax

`params = obj.getListViewParameters(check_ID)`

Arguments

`params`
: A cell array of `ModelAdvisor.ListViewParameter` objects.

`obj`
: A variable representing the `Simulink.ModelAdvisor` object.

`check_ID`
: A string that uniquely identifies the check.

  You can omit the *`check_ID`* if you use the method inside a check callback function.

Description

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 Verification and Validation software. For more information, seethe Simulink Verification and Validation documentation.

See Also

"setListViewParameters" on page 5-110, `ModelAdvisor.ListViewParameter`

getModelAdvisor

Purpose

Get Model Advisor object for system or subsystem

Syntax

```
obj = Simulink.ModelAdvisor.getModelAdvisor(system)
obj = Simulink.ModelAdvisor.getModelAdvisor(system, 'new')
```

Arguments

`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.

Description

This static method (see "Static Methods") creates and returns an instance of `Simulink.ModelAdvisor` class for the model or subsystem specified by `system`.

getSelectedCheck

Purpose

Get currently selected checks

Syntax

```
IDs = getSelectedCheck
```

Arguments

`IDs`

Cell array of IDs of currently selected checks.

Description

Returns the IDs of the currently selected checks in the Model Advisor.

See Also

"getSelectedTask"

getSelectedSystem

Purpose

Get system currently targeted by Model Advisor

Syntax

```
path = getSelectedSystem
```

Arguments

`path`

Path of the selected system.

Description

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`.

See Also

"getModelAdvisor"

getSelectedTask

Purpose

Get selected tasks

Syntax

`IDs = getSelectedTask`

Arguments

`IDs`
: Cell array of IDs of currently selected tasks.

Description

Returns the IDs of the currently selected tasks in the Model Advisor.

See Also

"getSelectedCheck"

getTaskAll

Purpose

Get tasks run by Model Advisor

Syntax

`IDs = getTaskAll`

Arguments

`IDs`
: Cell array of IDs of tasks run by the Model Advisor.

Description

Returns a cell array of IDs of tasks run by the Model Advisor.

See Also

"getCheckAll", "getGroupAll"

Simulink.ModelAdvisor.openConfigUI

Purpose

Starts Model Advisor Configuration editor

Syntax

`Simulink.ModelAdvisor.openConfigUI`

Description

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 Verification and Validation software (see “Organize Checks and Folders Using the Model Advisor Configuration Editor”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 Simulink project (`slprj`) folder (for more information, see “Model Reference Simulation Targets”) in the current folder to store reports and other information. If this folder does not exist in the current folder, the Model Advisor Configuration editor creates it.

Simulink.ModelAdvisor.reportexists

Purpose

Determine whether report exists for model or subsystem

Syntax

`exists = reportexists('`*`system`*`')`

Arguments

`system`

String specifying path of a system or subsystem.

`exists`

True (1) if a report exists for *`system`*.

Description

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.

See Also

“exportReport”

runCheck

Purpose

Run currently selected checks

Syntax

`success = runCheck`

Arguments

`success`
True (1) if the checks were run.

Description

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.

See Also

"selectCheck"

runTask

Purpose

Run currently selected tasks

Syntax

`success = runTask`

Arguments

`success`
True (1) if the tasks were run.

Description

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.

See Also

"selectTask"

selectCheck

Purpose

Select check

Syntax

`success = selectCheck(ID)`

Arguments

`ID`

ID or cell array of IDs of checks to be selected.

`success`

True (1) if this method succeeded in selecting the specified checks.

Description

Select the check specified by *`ID`*. This method cannot select a check that is disabled.

See Also

“selectCheckAll”, “selectCheckForGroup”, “deselectCheck”

selectCheckAll

Purpose

Select all checks

Syntax

`success = selectCheckAll`

Arguments

`success`

True (1) if this method succeeded in selecting all checks.

Description

Selects all checks that are not disabled.

See Also

“selectCheck”, “selectCheckForGroup”, “deselectCheck”

selectCheckForGroup

Purpose

Select group of checks

Syntax

`success = selectCheckForGroup(`*`ID`*`)`

Arguments

`ID`

ID or cell array of group IDs.

`success`

True (1) if this method succeeded in selecting the specified groups

Description

Selects the groups specified by *`ID`*.

See Also

"deselectCheckForGroup"

selectCheckForTask

Purpose

Select checks that belong to specified task or set of tasks

Syntax

`success = selectCheckForTask(`*`ID`*`)`

Arguments

`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

Description

Selects checks belonging to the tasks specified by the *`ID`* argument.

See Also

"deselectCheckForTask"

selectTask

Purpose

Select task

Syntax

`success = selectTask(`*`ID`*`)`

Arguments

`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.

Description

Selects a task.

See Also

"deselectTask"

selectTaskAll

Purpose

Select all tasks

Syntax

`success = selectTaskAll`

Arguments

`success`
: True (1) if this method succeeds in selecting all tasks.

Description

Selects all tasks.

See Also

"deselectTaskAll"

setActionenable

Purpose

Set status for check action

Syntax

`obj``.setActionenable(`*`value`*`)`

Arguments

`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.

Description

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 Verification and Validation software. For more information, seethe Simulink Verification and Validation documentation.

See Also

`ModelAdvisor.Action`

setBaselineMode

Purpose

Set baseline data generation mode for Model Advisor

Syntax

`setBaselineMode(`*`mode`*`)`

Arguments

`mode`

Boolean value indicating setting of Model Advisor's baseline mode, either on (`true`) or off (`false`).

Description

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.

See Also

"getBaselineMode", "verifyCheckResult", "verifyHTML"

setCheckErrorSeverity

Purpose

Set severity of check failure

Syntax

`obj``.setCheckErrorSeverity(`*`value`*`)`

Arguments

`obj`

A variable representing the `Simulink.ModelAdvisor` object.

`value`

Integer indicating severity of failure.

- `0` — Check Result = Warning
- `1` — Check Result = Failed

Description

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 Verification and Validation software. For more information, seethe Simulink Verification and Validation documentation.

See Also

"setCheckResultStatus"

setCheckResult

Purpose

Set result for currently running check

Syntax

`success = setCheckResult(result)`

Arguments

`result`

String or cell array that specifies the result of the currently running task.

`success`

True (1) if this method succeeds in setting the check result.

Description

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 Verification and Validation software. For more information, seethe Simulink Verification and Validation documentation.

See Also

"getCheckResult", "setCheckResultData", "setCheckResultStatus"

setCheckResultData

Purpose

Set result data for currently running check

Syntax

`success = setCheckResultData(data)`

Arguments

`data`

Result data to be set.

`success`
: True (1) if this method succeeds in setting the result data for the current check

Description

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 Verification and Validation software. For more information, seethe Simulink Verification and Validation documentation.

See Also

"getCheckResultData", "setCheckResult", "setCheckResultStatus"

setCheckResultStatus

Purpose

Set status for currently running check

Syntax

`success = setCheckResultStatus(`*`status`*`)`

Arguments

`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.

Description

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 Verification and Validation software. For more information, see the Simulink Verification and Validation documentation.

See Also

"getCheckResultStatus", "setCheckResult", "setCheckResultData", "setCheckErrorSeverity" on page 5-107

setListViewParameters

Purpose

Specify list view parameters for check

Syntax

`obj.setListViewParameters(check_ID, params)`

Arguments

`obj`

A variable representing the `Simulink.ModelAdvisor` object.

`check_ID`

A string 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.

Description

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 Verification and Validation software. For more information, seethe Simulink Verification and Validation documentation.

See Also

"getListViewParameters" on page 5-97, `ModelAdvisor.ListViewParameter`

verifyCheckRan

Purpose

Verify that Model Advisor has run set of checks

Syntax

```
[success, missingChecks, additionalChecks] =
verifyCheckRan(IDs)
```

Arguments

`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 ran.
`additionalChecks`	Cell array of IDs for unspecified checks that ran.

Description

The output variable success returns `true` if all the checks specified by *`IDs`* have run. If not, `success` returns `false`, `missingChecks` lists specified checks that did not run. The `additionalChecks` argument lists unspecified checks that ran.

See Also

"verifyCheckResultStatus"

verifyCheckResult

Purpose

Generate baseline Model Advisor check results file or compare current check results to baseline check results

Syntax

```
[success message] = verifyCheckResult(baseline, checkIDs)
```

Arguments

`baseline`
: Path of the baseline check results MAT-file.

`checkIDs`
: Cell array of check IDs.

`success`
: Boolean value specifying whether the method succeeded.

`message`
: String specifying an error message.

Description

If the Model Advisor is in baseline mode (see "setBaselineMode"), 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") 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". 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"). Then invoke this method with the path of the baseline report as the value of the `baseline` input argument.

See Also

"setBaselineMode", "getBaselineMode", "runCheck", "verifyCheckResultStatus"

verifyCheckResultStatus

Purpose

Verify that model has passed or failed set of checks

Syntax

`[success message] = verifyCheckResultStatus(`*`baseline`*`,` *`checkIDs`*`)`

Arguments

`baseline`
: Array of Boolean variables.

`checkIDs`
: Cell array of check IDs.

`success`
: Boolean value specifying whether the method succeeded.

`message`
: String specifying an error message.

Description

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”) before invoking `verifyCheckResultStatus`.

See Also

“runCheck”

verifyHTML

Purpose

Generate baseline Model Advisor report or compare current report to baseline report

Syntax

```
[success message] = verifyHTML(baseline)
```

Arguments

`baseline`
: Path of a Model Advisor report.

`success`
: Boolean value specifying whether the method succeeded.

`message`
: String specifying an error message.

Description

If the Model Advisor is in baseline mode (see “setBaselineMode”), 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”). The invoke this method with the path of the baseline report as the value of the *`baseline`* input argument.

See Also

“setBaselineMode”, “getBaselineMode”, “verifyCheckResult”

Simulink.ModelDataLogs

Purpose

Container for model's signal data logs

Description

Note The `ModelDataLogs` format is supported for backwards compatibility. The `ModelDataLogs` format will be removed in a future release. For an existing model that uses the `ModelDataLogs` format, you should migrate the model to use `Dataset` format. For details, see "Migrate from ModelDataLogs to Dataset Format".

For new models, use the `Dataset` logging format, which stores logged data in `Simulink.SimulationData.Dataset` objects.

If you set **Configuration Parameters > Data Import/Export > Signal logging format** to `ModelDataLogs`, Simulink software creates instances of the `Simulink.ModelDataLogs` class to contain signal logs that it creates while simulating a model (see "Signal Logging"). In particular, Simulink software creates an instance of this class for a top model and for each model referenced by the top model that contains signals to be logged. Simulink software assigns the `ModelDataLogs` object for the top model to a variable in the MATLAB workspace. The name of the variable is the name specified in the **Signal logging name** field on the **Data Import/export** pane of the model's **Configuration Parameters** dialog box. 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.ScopeDataLogs`

  Container for the data logs of Scope signal viewers

- `Simulink.Timeseries`

  Data log for any signal except a mux or bus signal

- `Simulink.TsArray`

  Data log for a mux or bus signal

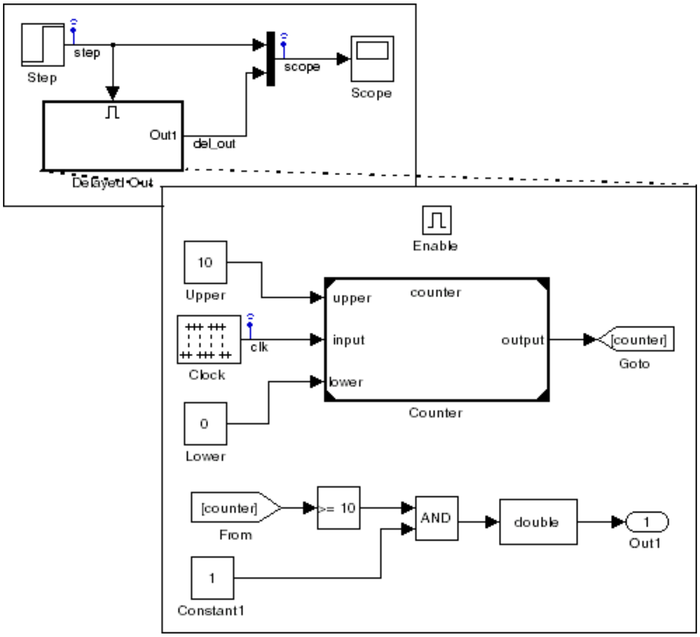
The names of the properties identify the data being logged as follows:

- For signal data logs, the name of the signal
- For a subsystem or model log container, the name of the subsystem or model, respectively
- For a scope viewer data log, the name specified on the viewer's parameter dialog box

---

Note If a name contains spaces, the `ModelDataLogs` objects specifies its name as `('`**`name`**`')` where **`name`** is the actual name, e.g., `('Brake Subsystem')`. See "Handling Spaces and Newlines in Logged Names" for more information.

Consider, for example, the following model.


step
Step
scope
Scope
Out1
del_out
Delayed Out
Enable
10
Upper
upper
counter
clk
Clock
input
output
[counter]
Goto
lower
Counter
0
Lower
[counter]
From
>= 10
AND
double
1
Out1
1
Constant1

As indicated by the testpoint icons, this model specifies that Simulink software should log the signals named `step` and `scope` in the model's root system and the signal named `clk` in the subsystem named Delayed Out. After simulation of this model, the MATLAB workspace contains the following variable:

```
>> logsout

logsout =

Simulink.ModelDataLogs (siglgex):
  Name                      elements  Simulink Class

  scope                        2      TsArray
  step                         1      Timeseries
  ('Delayed Out')              2      SubsysDataLogs
```

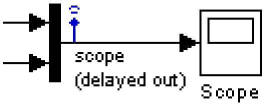
The `logsout` variable contains the signal data logged during the simulation. You can use fully qualified object names or the Simulink `unpack` command to access the signal data stored in `logsout`. For example, to access the amplitudes of the `clk` signal in the Delayed Out subsystem, enter

```
>> data = logsout.('Delayed Out').clk.Data;
```

or

```
>> logsout.unpack('all');
>> data = clk.Data;
```

You can use a custom logging name or signal name when logging a signal. If you use the signal name, and that name occupies more than one line, include an `sprintf('\n')` between the lines of the signal name when accessing the logged data. For example, to access the signal in the following model:



Use the following syntax:

```
logsout.(['scope' sprintf('\n') '(delayed out)'])
```

See “Handling Spaces and Newlines in Logged Names” for more information.

See Also

“Signal Logging”, `Simulink.SubsysDataLogs`, `Simulink.ScopeDataLogs`, `Simulink.Timeseries`, `Simulink.TsArray`, `who`, `whos`, `unpack`

Simulink.SimState.ModelSimState

Purpose

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 software generates a string 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.

Attributes:

`dependent`

snapshotTime

Time at which Simulink software 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 in the MATLAB Programming Fundamentals documentation.

Simulink.ModelManagement.Project.CurrentProject

Purpose

(To be removed) Manipulate the currently open Simulink project from the command line

---

Note `Simulink.ModelManagement.Project.CurrentProject` will be removed in a future release. Use `simulinkproject` and related functions instead. See "Simulink Projects" for links to project functions.

Description

Use an instance of `Simulink.ModelManagement.Project.CurrentProject` class to perform the following operations.

- Open and close Simulink projects
- Investigate the properties of the currently open Simulink project
- Change some properties of the currently open Simulink project

Construction

Create an instance of the class as follows.

```
p = Simulink.ModelManagement.Project.CurrentProject();
```

Methods

addFileToProject	(To be removed) Add a file to the currently open Simulink project
attachLabelToFile	(To be removed) Add a label to a file in the currently open Simulink project
close	(To be removed) Close the currently open Simulink project
createCategory	(To be removed) Create a category in the currently open Simulink project

createLabel	(To be removed) Create a label in a category in the currently open Simulink project
detachLabelFromFile	(To be removed) Remove a label from a file in the currently open Simulink project
export	(To be removed) Export Simulink project to Zip file
getAttachedLabels	(To be removed) Get a list of labels attached to a file in the currently open Simulink project
getCategories	(To be removed) Get a list of the categories of file labels in the currently open Simulink project
getFilesInProject	(To be removed) Get a list of files in the currently open Simulink project
getLabels	(To be removed) Get a list of labels in a category in the currently open Simulink project
getProjectName	(To be removed) Get the name of the currently open Simulink project
getRootDirectory	(To be removed) Get the name of the root folder for the currently open Simulink project
getRootFolder	(To be removed) Get the name of the root folder for the currently open Simulink project

isProjectLoaded	(To be removed) Return true when a Simulink project is loaded and false when no project is loaded
loadProject	(To be removed) Load an existing Simulink project and close any currently open project
removeCategory	(To be removed) Remove a category from the currently open Simulink project
removeFileFromProject	(To be removed) Remove a file from the currently open Simulink project
removeLabel	(To be removed) Remove a label from a category in the currently open Simulink project
setProjectName	(To be removed) Change the name of the currently open Simulink project

Examples

This example creates a new instance of the `Simulink.ModelManagement.Project.CurrentProject` class.

```
p = Simulink.ModelManagement.Project.CurrentProject();
```

How To

- “Simulink Projects”

Purpose

Describe model workspace

Description

Instances of this class describe model workspaces. Simulink software creates an instance of this class for each model that you open during a Simulink session. See "Model Workspaces" for more information.

Property Summary

Name	Access	Description
`DataSource`	`RW`	Specifies the source used to initialize this workspace. Valid values are • `'Model File'` • `'MAT-File'` • `'MATLAB Code'` • `'MATLAB File'`
`FileName`	`RW`	Specifies the name of the MAT-file used to initialize this workspace. Simulink software ignores this property if `DataSource` is neither `'MAT-File'` nor `'MATLAB Code'`.
`MATLABCode`	`RW`	A string specifying MATLAB code used to initialize this workspace. Simulink software ignores this property if `DataSource` is not `'MATLAB Code'`.

Method Summary

Name	Description
"assignin"	Assign a value to a variable in the model's workspace.
"clear"	Clear the model's workspace.

Name	Description
“evalin”	Evaluate an expression in the model’s workspace.
“reload”	Reload the model workspace from the workspace’s data source.
“save”	Save the model’s workspace to a specified MAT-file.
“saveToSource”	Save the workspace to the MAT-file that the workspace designates as its data source.
“whos”	List the variables in the model workspace.

Methods

`assignin`

Purpose

Assign a value to a variable in the model’s workspace.

Syntax

```
assignin('varname', varvalue)
```

Arguments

`varname`
Name of the variable to be assigned a value.

`varvalue`
Value to be assigned the variable.

Description

This method assigns the value specified by `varvalue` to the variable whose name is `varname`.

See also

“evalin”

`clear`

Purpose

Clear the model's workspace.

Syntax

`clear`

Description

This method empties the workspace of its variables.

`evalin`

Purpose

Evaluate an expression in the model's workspace.

Syntax

`evalin('expression')`

Arguments

`expression`
A MATLAB expression to be evaluated.

Description

This method evaluates expression in the model workspace.

See also

"assignin"

`reload`

Purpose

Reload the model workspace from the workspace's data source.

Syntax

`reload`

Description

This method reloads the model workspace from the data source specified by its DataSource parameter. The data source must be `'MAT-File'`, `'MATLAB Code'`, or `'MATLAB File'` .

See also

"saveToSource"

`save`

Purpose

Save the model's workspace to a specified MAT-file.

Syntax

`save('filename')`

Arguments

`filename`
: Name of a MAT-file.

Description

This method saves the model's workspace to the MAT-file specified by `filename`.

Note This method allows you to save the workspace to a file other than the file specified by the workspace's `FileName` property. If you want to save the model workspace to the file specified by the file's `FileName` property, it is simpler to use the workspace's `saveToSource` method.

example

```
hws = get_param('mymodel','modelworkspace')
hws.DataSource = 'MAT-File';
hws.FileName = 'workspace';
hws.assignin('roll', 30);
hws.saveToSource;
hws.assignin('roll', 40);
hws.save('workspace_test.mat');
```

See also

"reload", "saveToSource"

`saveToSource`

Purpose

Save the workspace to the MAT-file that it designates as its data source.

Syntax

```
saveToSource
```

Description

This method saves the model workspace designated by its `FileName` property.

example

```
hws = get_param('mymodel','modelworkspace')
hws.DataSource = 'MAT-File';
hws.FileName = 'params';
hws.assignin('roll', 30);
hws.saveToSource;
```

See also

“save”, “reload”

`whos`

Purpose

List the variables in the model workspace.

Syntax

```
whos
```

Description

This method lists the variables in the model’s workspace. The listing includes the size and class of the variables.

example

```
>> hws = get_param('mymodel','modelworkspace');
>> hws.assignin('k', 2);
>> hws.whos

  Name      Size                      Bytes  Class

  k         1x1                           8  double array
```

Purpose

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"	enable Level-2 MATLAB S-function to use multidimensional signals.
"DialogPrmsTunable"	Specifies which of the S-function's dialog parameters are tunable.
"NextTimeHit"	Time of the next sample hit for variable sample time S-functions.

Method Summary

Name	Description
"AutoRegRuntimePrms"	Register this block's dialog parameters as run-time parameters.
"AutoUpdateRuntimePrms"	Update this block's run-time parameters.
"IsDoingConstantOutput"	Determine whether the current simulation stage is the constant sample time stage.
"IsMajorTimeStep"	Determine whether the current simulation time step is a major time step.
"IsSampleHit"	Determine whether the current simulation time is one at which a task handled by this block is active.

Name	Description
“IsSpecialSampleHit”	Determine whether the current simulation time is one at which multiple tasks handled by this block are active.
“RegBlockMethod”	Register a callback method for this block.
“RegisterDataTypeFxpBinaryPoint”	Register fixed-point data type with binary point-only scaling.
“RegisterDataTypeFxpFSlopeFixexpBias”	Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias.
“RegisterDataTypeFxpSlopeBias”	Register data type with [Slope Bias] scaling.
“SetAccelRunOnTLC”	Specify whether to use this block’s TLC file to generate the simulation target for the model that uses it.
“SetPreCompInpPortInfoToDynamic”	Set precompiled attributes of this block’s input ports to be inherited.
“SetPreCompOutPortInfoToDynamic”	Set precompiled attributes of this block’s output ports to be inherited.

Name	Description
"SetPreCompPortInfoToDefaults"	Set precompiled attributes of this block's ports to the default values.
"SetSimViewingDevice"	Specify whether block is a viewer.
"SupportsMultipleExecInstances"	
"WriteRTWParam"	Write custom parameter information to Simulink Coder file.

Properties

`AllowSignalsWithMoreThan2D`

Description

Allow Level-2 MATLAB S-functions to use multidimensional signals. You must set the `AllowSignalsWithMoreThan2D` property in the `setup` method.

Data Type

Boolean

Access

`RW`

`DialogPrmsTunable`

Description

Specifies whether a dialog parameter of the S-function is tunable. Tunable parameters are registered as run-time parameters when you call the "AutoRegRuntimePrms" 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'};
```

Data Type

array

Access

`RW`

`NextTimeHit`

Description

Time of the next sample hit for variable sample-time S-functions.

Data Type

double

Access

`RW`

Methods

`AutoRegRuntimePrms`

Purpose

Register a block's tunable dialog parameters as run-time parameters.

Syntax

`AutoRegRuntimePrms;`

Description

Use in the `PostPropagationSetup` method to register this block's tunable dialog parameters as run-time parameters.

`AutoUpdateRuntimePrms`

Purpose

Update a block's run-time parameters.

Syntax

`AutoRegRuntimePrms;`

Description

Automatically update the values of the run-time parameters during a call to `ProcessParameters`.

See the S-function *`matlabroot`*`adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

`IsDoingConstantOutput`

Purpose

Determine whether this is in the constant sample time stage of a simulation.

Syntax

`bVal = IsDoingConstantOutput;`

Description

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`

Purpose.

Determine whether current time step is a major or a minor time step.

Syntax

```
bVal = IsMajorTimeStep;
```

Description

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`

Purpose

Determine whether the current simulation time is one at which a task handled by this block is active.

Syntax

```
bVal = IsSampleHit(stIdx);
```

Arguments

`stIdx`
: Global index of the sample time to be queried.

Description

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`

Purpose

Determine whether the current simulation time is one at which multiple tasks implemented by this block are active.

Syntax

`bVal = IsSpecialSampleHit(stIdx1,stIdx1);`

Arguments

`stIdx1`
: Index of sample time of first task to be queried.

`stIdx2`
: Index of sample time of second task to be queried.

Description

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).

`RegBlockMethod`

Purpose

Register a block callback method.

Syntax

`RegBlockMethod(methName, methHandle);`

Arguments

`methName`
: Name of method to be registered.

`methHandle`

MATLAB function handle of the callback method to be registered.

Description

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`

Purpose

Register fixed-point data type with binary point-only scaling.

Syntax

```
dtID = RegisterDataTypeFxpBinaryPoint(isSigned, wordLength,
fractionalLength, obeyDataTypeOverride);
```

Arguments

`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 `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.

Description

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”.

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” to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use “RegisterDataTypeFxpSlopeBias” 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`

Purpose

Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias

Syntax

```
dtID = RegisterDataTypeFxpFSlopeFixexpBias(isSigned,
wordLength, fractionalSlope, fixedexponent, bias,
obeyDataTypeOverride);
```

Arguments

`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.

Description

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”.

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” to register a data type with binary point-only scaling.
- Use “RegisterDataTypeFxpSlopeBias” 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`

Purpose

Register data type with [Slope Bias] scaling.

Syntax

```
dtID = RegisterDataTypeFxpSlopeBias(isSigned, wordLength,
totalSlope, bias, obeyDataTypeOverride);
```

Arguments

`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.

Description

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 5-176.

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" to register a data type with binary point-only scaling.
- Use "RegisterDataTypeFxpFSlopeFixexpBias" 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`

Purpose

Specify whether to use block's TLC file to generate code for the Accelerator mode of Simulink software.

Syntax

```
SetAccelRunOnTLC(bVal);
```

Arguments

`bVal`

May be `'true'` (use TLC file) or `'false'` (run block in interpreted mode).

Description

Specify if the block should use its TLC file to generate code that runs with the accelerator. If this option is `'false'`, the block runs in interpreted mode. See the S-function *`matlabroot`*`/toolbox/simulink/blocks/msfcn_times_two.m` in the Simulink model `msfcndemo_timestwo` for an example.

`SetPreCompInpPortInfoToDynamic`

Purpose

Set precompiled attributes of this block's input ports to be inherited.

Syntax

`SetPreCompInpPortInfoToDynamic;`

Description

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/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

`SetPreCompOutPortInfoToDynamic`

Purpose

Set precompiled attributes of this block's output ports to be inherited.

Syntax

`SetPreCompOutPortInfoToDynamic;`

Description

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/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

`SetPreCompPortInfoToDefaults`

Purpose

Set precompiled attributes of this block's ports to the default values.

Syntax

`SetPreCompPortInfoToDefaults;`

Description

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`

Purpose

Specify whether this block is a viewer.

Syntax

`SetSimViewingDevice(bVal);`

Arguments

`bVal`

May be `'true'` (is a viewer) or `'false'` (is not a viewer).

Description

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`

Purpose

Specify whether or not a For Each Subsystem supports an S-function inside of it.

Syntax

`SupportsMultipleExecInstances(bVal);`

Arguments

`bVal`

May be `'true'` (S-function is supported) or `'false'` (S-function is not supported).

Description

Specify if an S-function can operate within a For Each Subsystem.

`WriteRTWParam`

Purpose

Write a custom parameter to the Simulink Coder information file used for code generation.

Syntax

`WriteRTWParam(pType, pName, pVal)`

Arguments

`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.

Description

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/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

Simulink.NumericType

Purpose

Specify data type

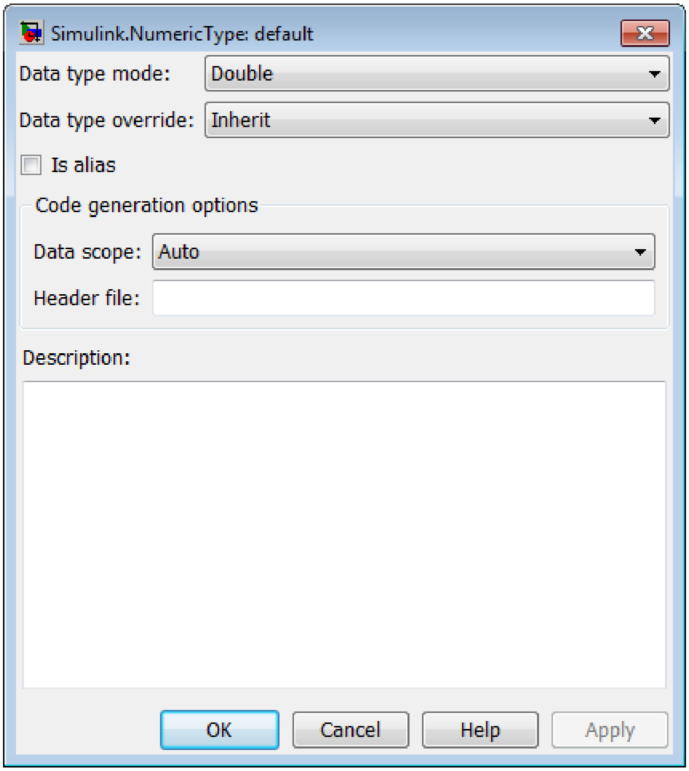
Description

This class lets you specify a data type as follows:

1. Create an instance of this class in the MATLAB base workspace or a model workspace. To create a numeric type in a model workspace, you must disable the **Is alias** option.
2. Set object properties to the properties of the custom 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.

Property Dialog Box



Simulink.NumericType: default

Data type mode: Double

Data type override: Inherit

☐ Is alias

Code generation options

Data scope: Auto

Header file:

Description:

Data type mode

Data type of this numeric type. The options are

Option	Description
`Double`	Same as the MATLAB `double` type.
`Single`	Same as the MATLAB `single` type.
`Boolean`	Same as the MATLAB `boolean` type.

Option	Description
`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 data type mode causes Simulink software to enable controls on the dialog box (see below) that apply to the mode and to disable other controls that do not apply. Selecting a fixed-point data type mode might, depending on the other dialog box options that you select, cause the model to run only on systems that have a Fixed-Point Designer option installed.

Data type override

Data type override setting for this numeric type. The options are

Option	Description
Inherit (default)	Data type override setting for the context in which this numeric type is used (block, signal, Stateflow chart in Simulink) applies to this numeric type.
Off	Data type override setting does not affect this numeric type.

Is alias

If this option is selected for a workspace object of this type, Simulink software uses the name of the object as the data type for all objects that specify the object as its data type. Otherwise, Simulink software uses the data type mode of the data type as its name, or, if the data type mode is a fixed-point mode, Simulink software generates a name that encodes the type's properties, using the encoding specified by the Fixed-Point Designer product.

Data scope

Specifies whether the data type definition should be imported from, or exported to, a header file during code generation. The possible values are:

Value	Action
`Auto` (default)	If no value is specified for **Header file**, export the type definition to `*model*_types.h`, where `*model*` is the model name. If a value is specified for **Header file**, import the data type definition from the specified header file.
`Exported`	Export the data type definition to a header file, which can be specified in the **Header file** field. If no value is specified for **Header file**, the header file name defaults to *`type`*`.h`, where*`type`* is the data type name.
`Imported`	Import the data type definition from a header file, which can be specified in the **Header file** field. If no value is specified for **Header file**, the header file name defaults to *`type`*`.h`, where*`type`* is the data type name.

Header file

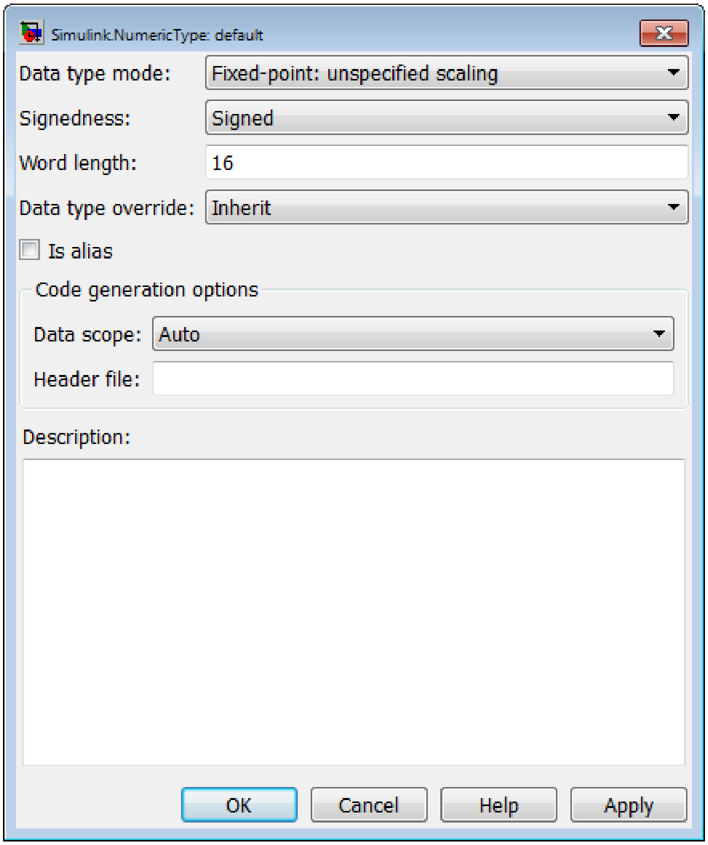
Name of a C header file from which a data type definition is imported, or to which a data type definition is exported, based on the value of **Data scope**. If this field is specified, the specified name is used during code generation for importing or exporting. If this field is empty, the value defaults to *`type`*`.h` if **Data scope** equals `Imported` or `Exported`, or defaults to `*model*_types.h` if **Data scope** equals `Auto`.

Description

Description of this data type. This field is intended for use in documenting this data type. Simulink software ignores it.

Signedness

Specifies whether the data type is signed or unsigned, or inherits its signedness. Set the option to `Signed`, `Unsigned`, or `Auto`. This option is enabled only for fixed-point data type modes.



Data type mode: Fixed-point: unspecified scaling

Signedness: Signed

Word length: 16

Data type override: Inherit

☐ Is alias

Code generation options

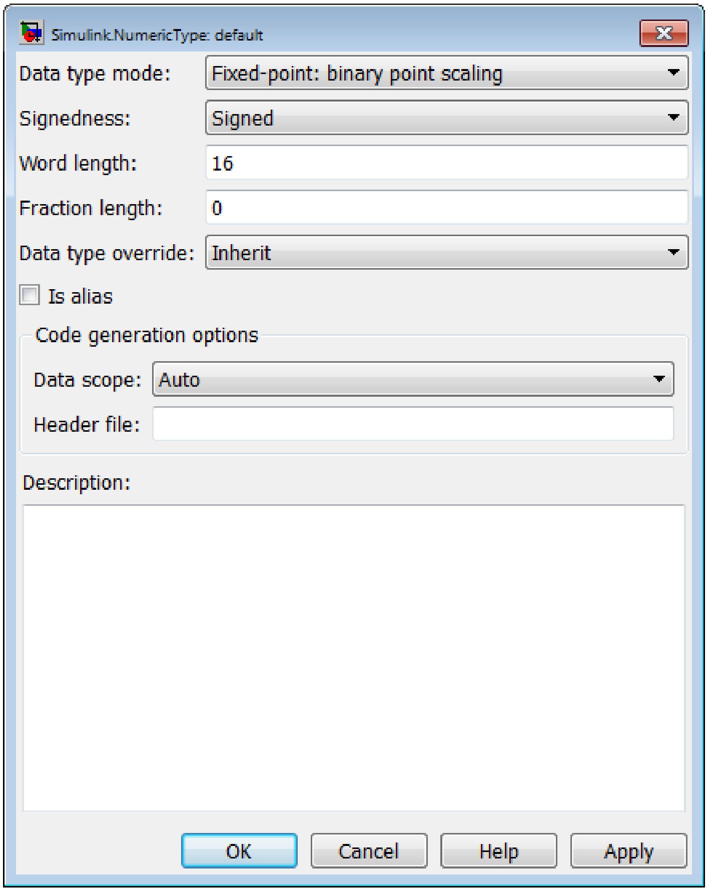
Data scope: Auto

Header file:

Description:

Word-Length
Word length in bits of the fixed-point data type. This option is enabled only for fixed-point data type modes.

Fraction length
Number of bits to the right of the binary point. This option is enabled only if the data type mode is `Fixed-point: binary point scaling`.


Simulink.NumericType: default

Data type mode: Fixed-point: binary point scaling

Signedness: Signed

Word length: 16

Fraction length: 0

Data type override: Inherit

☐ Is alias

Code generation options

Data scope: Auto

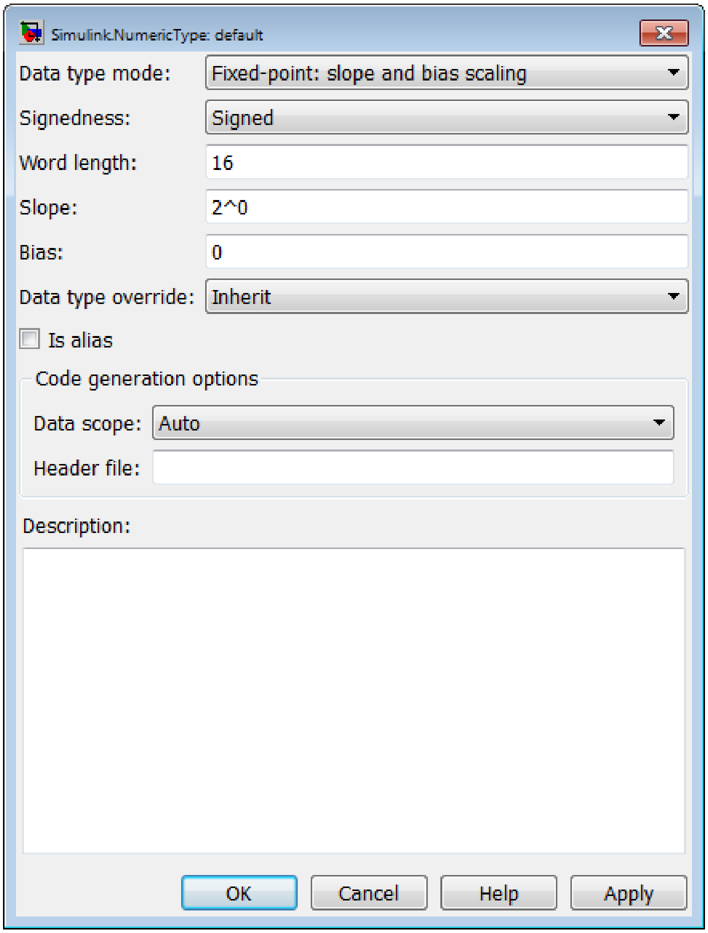
Header file:

Description:

OK
Cancel
Help
Apply

Slope

Slope for slope and bias scaling. This option is enabled only if the data type mode is `Fixed-point: slope and bias scaling`.



Simulink.NumericType: default

Data type mode: Fixed-point: slope and bias scaling

Signedness: Signed

Word length: 16

Slope: 2^0

Bias: 0

Data type override: Inherit

☐ Is alias

Code generation options

Data scope: Auto

Header file:

Description:

Bias

Bias for slope and bias scaling. This option is enabled only if the data type mode is `Fixed-point:  slope and bias scaling`. See the preceding figure.

Properties

Name	Access	Description
`Bias`	`RW`	Bias used for slope and bias scaling of a fixed-point data type. This field is intended for use by the Fixed-Point Designer product. (Bias)
`DataScope`	`RW`	A string specifying whether the data type definition should be imported from, or exported to, a header file during code generation. (Data scope)
`DataTypeMode`	`RW`	String that specifies the data type mode of this numeric type. Valid values are `'Double'`, `'Boolean'`, `'Single'`, `'Fixed-point:  unspecified scaling'`, `'Fixed-point:  binary point scaling'`, and `'Fixed-point: slope and bias scaling'`. (Data type mode)
`DataTypeOverride`	`RW`	String that specifies the data type override mode. Valid values are `Inherit` and `Off`. (Data type override)
`Description`	`RW`	Description of this data type. (Description)

Name	Access	Description
`Fixedexponent`	`RW`	Exponent used for binary point scaling. This property equals `-FractionLength`. Setting this property causes Simulink software to set the `FractionLength` and `Slope` properties accordingly, and vice versa. This property applies only if the `DataTypeMode` is `Fixed-point:  binary point scaling` or `Fixed-point:  slope and bias scaling`. It does not appear on the object's Property dialog box, but can be accessed from the MATLAB command prompt.
`FractionLength`	`RW`	Integer that specifies the size in bits of the fractional portion of the fixed-point number. This property equals `-Fixedexponent`. Setting this property causes Simulink software to set the `Fixedexponent` property accordingly, and vice versa. This field is intended for use by the Fixed-Point Designer product. (Fraction length)
`HeaderFile`	`RW`	A string that specifies the name of a C header file from which a data type definition is imported, or to which a data type definition is exported, during code generation. (Header file)
`IsAlias`	`RW`	Integer that specifies whether to use the name of this object as the name of the data type that it specifies. Valid values are `1` (yes) or `0` (no). (Is alias)

Name	Access	Description
`Signedness`	`RW`	Boolean that specifies whether this data type is signed, unsigned, or inherits its signedness. Valid values are `1` (signed), `0` (unsigned), or `Auto` (inherit signedness). (Signedness)
`Slope`	`RW`	Slope for slope and bias scaling of fixed-point numbers. 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. This property appears only if `DataTypeMode` is `Fixed-point:  slope and bias scaling`. (Slope)
`SlopeAdjustmentFactor`	`RW`	Slope for slope and bias scaling of fixed-point numbers. Setting this property causes Simulink software to adjust the Slope property accordingly, and vice versa. This property applies only if `DataTypeMode` is `Fixed-point:  slope and bias scaling`. It does not appear on the object's Property dialog box, but can be accessed from the MATLAB command prompt.
`WordLength`	`RW`	Integer that specifies the word size of this data type. This field is intended for use by the Fixed-Point Designer product. This

Name	Access	Description
		property appears only if `DataTypeMode` is `Fixed-point`. (Word Length)

Methods

Name	Description
`isboolean`	Determine whether data type is Boolean Returns `1` when the `DataTypeMode` is `'Boolean'`, `0` otherwise.
`isdouble`	Determine whether data type is double precision Returns `1` when the `DataTypeMode` is `'Double'`, `0` otherwise.
`isfixed`	Determine whether data type is fixed point Returns `1` when the `DataTypeMode` is any of the fixed-point options, `0` otherwise. The fixed-point options are: • `'Fixed-point: unspecified scaling'` • `'Fixed-point: binary point scaling'` • `'Fixed-point: slope and bias scaling'`
`isfloat`	Determine whether data type is floating point Returns `1` when the `DataTypeMode` is `'Double'` or `'Single'`, `0` otherwise.

Name	Description
`isscalingbinarypoint`	Determine whether data type has binary point scaling Returns `1` when the data type has binary point scaling or trivial slope and bias scaling, `0` otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero. `DataTypeMode` is `'Fixed-point:  binary point scaling'`
`isscalingslopebias`	Determine whether data type has nontrivial slope and bias scaling Returns `1` when the data type has nontrivial slope and bias scaling, `0` otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero. `DataTypeMode` is `'Fixed-point:  slope and bias scaling'`
`isscalingunspecified`	Determine whether data type has unspecified scaling Returns `1` when the data type is fixed point and its scaling has not been specified, `0` otherwise. `DataTypeMode` is `'Fixed-point: unspecified scaling'`
`issingle`	Determine whether data type is single-precision Returns `1` when the `DataTypeMode` is `'Single'`, `0` otherwise.

See Also

"Generate Code Including User-Defined Data Types" in the Embedded Coder documentation

Purpose

Specify value, value range, data type, and other properties of block parameter

Description

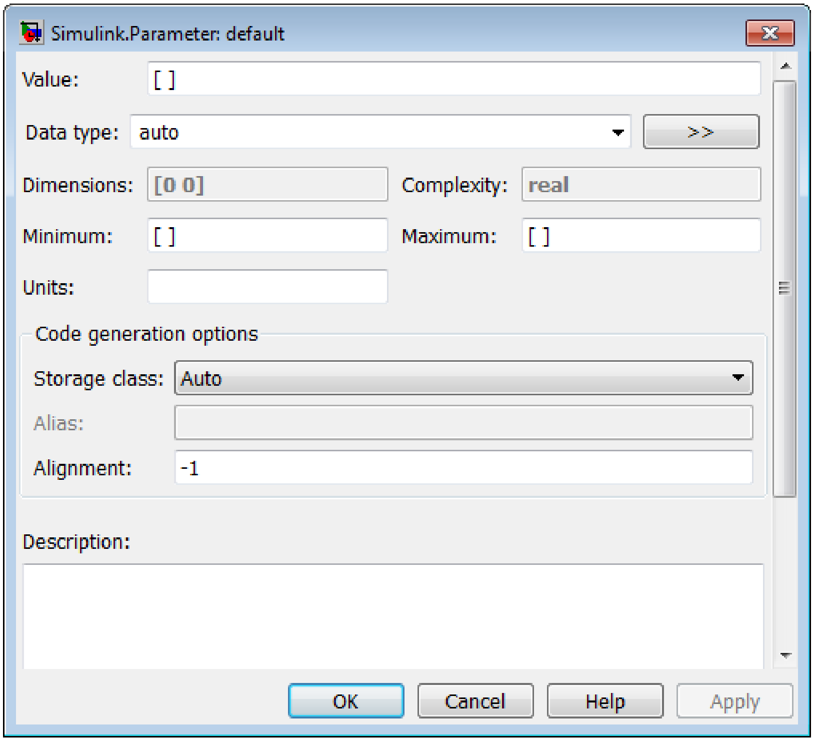
This class enables you to create workspace objects that you can then use as the values of block parameters, (for example, the value of the `Gain` parameter of a Gain block. You can create a `Simulink.Parameter` object in the base MATLAB workspace or a model workspace. However, to create the object in a model workspace, you must set the object storage class to `Auto`.

Parameter objects let you specify not only the value of a parameter but also other information about the parameter, such as the parameter's purpose, its dimensions, or its minimum and maximum values. Some Simulink products use this information. For example, Simulink and Simulink Coder products use information specified by `Simulink.Parameter` objects to determine whether the parameter is tunable (see "Tunable Parameters" in *Simulink User's Guide*).

Simulink performs range checking of parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum and maximum values and data type.

You can use the **Simulink.Parameter** dialog box to define a `Simulink.Parameter` object. To open the dialog box, in the Model Explorer, select the base workspace or a model workspace and select **Add > Simulink Parameter**.

Property Dialog Box



Simulink.Parameter: default

Value: [ ]

Data type: auto

Dimensions: [0 0]

Complexity: real

Minimum: [ ]

Maximum: [ ]

Units:

Code generation options

Storage class: Auto

Alias:

Alignment: -1

Description:

Value

Value of the parameter. You can use MATLAB expressions to specify the numeric type, dimensions, and data type of the parameter (see “Data Types Supported by Simulink”). You can also specify fixed-point values for block parameters (see “Specifying Fixed-Point Values Directly” in the Fixed-Point Designer documentation). The following examples illustrate this syntax.

Expression	Description
`single(1.0)`	Specifies a single-precision value of 1.0
`int8(2)`	Specifies an 8-bit integer of value 2

Expression	Description
`int32(3+2i)`	Specifies a complex value whose real and imaginary parts are 32-bit integers
`fi(2.3,true,16,3)`	Specifies a signed fixed-point numeric object having a value of 2.3, a word length of `16` bits, and a fraction length of `3`. (Requires Fixed-Point Designer)
`struct('A',20,'B',5)`	Specifies a structure for a bus with two signals, `A` and `B`, with double-precision values of 20 and 5, respectively. The shape and attributes of the structure must match the shape and attributes of the elements in the bus. For information about specifying an initial condition structure, see "Specify Initial Conditions for Bus Signals". (Requires Fixed-Point Designer)

Note Specifying a typed expression as the **Value** property of the parameter object overrides the current setting of the **Data type** property.

If **Value** evaluates to a MATLAB structure, then in the **Configuration Parameters > Diagnostics > Data Validity** pane, set "Underspecified initialization detection" to `simplified`.

Data type

Data type of the parameter. You can either select a data type from the adjacent pulldown menu or enter a string. If you select `auto` (the default), the block that references the parameter object determines the data type of the variable used to represent this parameter in code generated from the model. If you enter a string, it must evaluate to one of the following:

- A built-in data type that Simulink supports (see “Data Types Supported by Simulink”).
- A `Simulink.NumericType` object
- A `Simulink.AliasType` object
- A `Simulink.Bus` object (using the `Bus: <object name>` option)

  You can use the Bus Editor to define or edit a `Simulink.Parameter` object with a bus object for its data type. Use one of these approaches:

  - Select the **File > Create/Edit a Simulink.Parameter object** menu item.
  - Click the **Create/Edit a Simulink.Parameter object** icon (▦) from the toolbar.

  You can then edit the `Simulink.Parameter` object in the MATLAB Editor and evaluate the code to create or update the values for this `Simulink.Parameter` object.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Data type** parameter. (See “Specify Data Types Using Data Type Assistant” in *Simulink User’s Guide*.)

Note If you specify a parameter object data type using the **Data type** property, it overrides any typed expression in the **Value** property and changes the value to be untyped.

Units
Measurement units in which this value is expressed (for example, inches). This field is intended for use in documenting this parameter. Simulink ignores it.

Dimensions
Dimensions of the parameter. Simulink determines the dimensions from the entry in the **Value** field of this parameter. You cannot set this field yourself.

Complexity
Numeric type (i.e., real or complex) of the parameter. Simulink determines the numeric type of this parameter from the entry in the **Value** field of this parameter. You cannot set this field yourself.

Minimum
Minimum value that the parameter 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 parameter, do not set the minimum value for bus data on the parameter 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 generates a warning if the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type. When updating the diagram or starting a simulation, Simulink generates an error in these cases.

Maximum
Maximum value that the parameter 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 parameter, do not set the maximum value for bus data on the parameter 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 generates a warning if the parameter value is greater than the maximum value or if the maximum value is outside the range of the parameter data type. When updating the diagram or starting a simulation, Simulink generates an error in these cases.

Storage class

Storage class of this parameter. Simulink code generation products use this property to allocate memory for this parameter in generate code. For more information, see "Tunable Parameter Storage Classes".

Alias

Alternate name for this parameter. Simulink ignores this setting.

Alignment

Data alignment boundary, specified in number of bytes. The starting memory address for the data allocated for the parameter will be a multiple of the **Alignment** setting. The default value is `-1`, which specifies that the code generator should determine an optimal alignment based on usage. Otherwise, specify a positive integer that is a power of 2, not exceeding 128. This field is intended for use by Simulink Coder software (see "Configure Data Alignment for Function Implementations"). Simulink software ignores this setting.

Description

Description of this parameter. This field is intended for use in documenting this parameter. Simulink ignores it.

Properties

Name	Access	Description
`Value`	RW	Value of this parameter. (Value)
`CoderInfo`	RW	Information used by Simulink Coder software for generating code for this parameter. The value of this property is an object of `Simulink.ParamCoderInfo` class.
`Description`	RW	String that describes this parameter. This property is intended for user use. Simulink itself does not use it. (Description)
`DataType`	RW	String specifying the data type of this parameter. (Data type)
`Min`	RW	Minimum value that this parameter can have. (Minimum)
`Max`	RW	Maximum value that this parameter can have. (Maximum)
`DocUnits`	RW	Measurement units in which this parameter's value is expressed. (Units)
`Complexity`	RO	String specifying the numeric type of this parameter. Valid values are `'real'` or `'complex'`. (Complexity)
`Dimensions`	RO	Vector specifying the dimensions of this parameter. (Dimensions)

Simulink.ParamCoderInfo

Purpose

Specify information needed to generate code for parameter

Description

Simulink software creates an instance of this class for each instance of a `Simulink.Parameter` object that it creates. Simulink software uses the `Simulink.ParamCoderInfo` object to store information needed to generate code for the parameter specified by the `Simulink.Parameter` object.

You can set the properties of an instance of this class via the `CoderInfo` property or the property dialog box of the `Simulink.Parameter` object that uses it. For example, the following MATLAB expression sets the `StorageClass` property of a `Simulink.ParamCoderInfo` object used by a parameter object name `myparam`.

```
myparam.CoderInfo.StorageClass = 'exportedGlobal';
```

Property Dialog Box

Use the Code Generation Options section of the `Simulink.Parameter` property dialog box to set the `StorageClass`, `Alias`, and `Alignment` properties of objects of this class.

Properties

Name	Description
`Alias`	Alternate name for this parameter.
`Alignment`	Data alignment boundary for this parameter. See “Configure Data Alignment for Function Implementations” in the Embedded Coder documentation for more information.
`CustomAttributes`	Custom storage class attributes of this parameter. See “Introduction to Custom Storage Classes” in the Embedded Coder documentation for more information.

Name	Description
`CustomStorageClass`	Custom storage class of this parameter.
`StorageClass`	Storage class of this parameter. For more information, see “Parameters” in the Simulink Coder documentation.

Simulink.RunTimeBlock

Purpose

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"	Block's handle.
"CurrentTime"	Current simulation time.

Name	Description
"NumDworks"	Number of discrete work vectors used by the block.
"NumOutputPorts"	Number of block output ports.
"NumContStates"	Number of block's continuous states.
"NumDworkDiscStates"	Number of block's discrete states
"NumDialogPrms"	Number of parameters that can be entered on S-function block's dialog box.
"NumInputPorts"	Number of block's input ports.
"NumRuntimePrms"	Number of run-time parameters used by block.
"SampleTimes"	Sample times at which block produces outputs.

Method Summary

Name	Description
"ContStates"	Get a block's continuous states.
"DataTypeIsFixedPoint"	Determine whether a data type is fixed point.
"DatatypeName"	Get name of a data type supported by this block.
"DatatypeSize"	Get size of a data type supported by this block.
"Derivatives"	Get a block's continuous state derivatives.
"DialogPrm"	Get a parameter entered on an S-function block's dialog box.
"Dwork"	Get one of a block's DWork vectors.

Name	Description
"FixedPointNumericType"	Determine the properties of a fixed-point data type.
"InputPort"	Get one of a block's input ports.
"OutputPort"	Get one of a block's output ports.
"RuntimePrm"	Get one of the run-time parameters used by a block.

Properties

`BlockHandle`

Description

Block's handle.

Access

`RO`

`CurrentTime`

Description

Current simulation time.

Access

`RO`

`NumDworks`

Description

Number of data work vectors.

Access

`RW`

See Also

`ssGetNumDWork`

`NumOutputPorts`

Description

Number of output ports.

Access

`RW`

See Also

`ssGetNumOutputPorts`

`NumContStates`

Description

Number of continuous states.

Access

`RW`

See Also

`ssGetNumContStates`

`NumDworkDiscStates`

Description

Number of discrete states. In a MATLAB S-function, you need to use DWorks to set up discrete states.

Access

`RW`

See Also

`ssGetNumDiscStates`

`NumDialogPrms`

Description

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.

Access

`RW`

See Also

`ssGetNumSFcnParams`

`NumInputPorts`

Description

Number of input ports.

Access

`RW`

See Also

`ssGetNumInputPorts`

`NumRuntimePrms`

Description

Number of run-time parameters used by this block. See "Run-Time Parameters" for more information.

Access

`RW`

See Also

`ssGetNumSFcnParams`

`SampleTimes`

Description

Block's sample times.

Access

`RW` for MATLAB S-functions, `RO` for all other blocks.

Methods

`ContStates`

Purpose

Get a block's continuous states.

Syntax

`states = ContStates();`

Description

Get vector of continuous states.

See Also

`ssGetContStates`

`DataTypeIsFixedPoint`

Purpose

Determine whether a data type is fixed point.

Syntax

`bVal = DataTypeIsFixedPoint(dtID);`

Arguments

`dtID`

Integer value specifying the ID of a data type.

Description

Returns `true` if the specified data type is a fixed-point data type.

`DatatypeName`

Purpose

Get the name of a data type.

Syntax

`name = DatatypeName(dtID);`

Arguments

`dtID`

Integer value specifying ID of a data type.

Description

Returns the name of the data type specified by `dtID`.

See Also

"DatatypeSize"

`DatatypeSize`

Purpose

Get the size of a data type.

Syntax

```
size = DatatypeSize(dtID);
```

Arguments

`dtID`

Integer value specifying the ID of a data type.

Description

Returns the size of the data type specified by `dtID`.

See Also

"DatatypeName"

`Derivatives`

Purpose

Get derivatives of a block's continuous states.

Syntax

```
derivs = Derivatives();
```

Description

Get vector of state derivatives.

See Also

`ssGetdX`

`DialogPrm`

Purpose

Get an S-function's dialog parameters.

Syntax

```
param = DialogPrm(pIdx);
```

Arguments

`pIdx`

Integer value specifying the index of the parameter to be returned.

Description

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.

See Also

`ssGetSFcnParam`, "RuntimePrm"

`Dwork`

Purpose

Get one of a block's DWork vectors.

Syntax

```
dworkObj = Dwork(dwIdx);
```

Arguments

`dwIdx`

Integer value specifying the index of a work vector.

Description

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`.

See Also

`ssGetDWork`

`FixedPointNumericType`

Purpose

Get the properties of a fixed-point data type.

Syntax

```
eno = FixedPointNumericType(dtID);
```

Arguments

`dtID`

Integer value specifying the ID of a fixed-point data type.

Description

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.

`InputPort`

Purpose

Get an input port of a block.

Syntax

```
port = InputPort(pIdx);
```

Arguments

`pIdx`

Integer value specifying the index of an input port.

Description

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.

See Also

`ssGetInputPortSignalPtrs`, `Simulink.BlockPreCompInputPortData`, `Simulink.BlockCompInputPortData`, "OutputPort"

`OutputPort`

Purpose

Get an output port of a block.

Syntax

```
port = OutputPort(pIdx);
```

Arguments

`pIdx`

> Integer value specifying the index of an output port.

Description

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.

See Also

`ssGetInputPortSignalPtrs`,
`Simulink.BlockPreCompOutputPortData`,
`Simulink.BlockCompOutputPortData`

`RuntimePrm`

Purpose

Get an S-function's run-time parameters.

Syntax

```
param = RuntimePrm(pIdx);
```

Arguments

`pIdx`

Integer value specifying the index of a run-time parameter.

Description

Get the run-time parameter whose index is `pIdx`.

See Also

`ssGetRunTimeParamInfo`

Simulink.SampleTime

Purpose

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 character 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 string that represents the annotation of a specific sample time (for example, 'D1')

OwnerBlock

For asynchronous and variable sample times, `OwnerBlock` is a string containing the full path to the block that controls the sample time. For all other types of sample times, it is an empty string.

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 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.ScopeDataLogs

Purpose

Store data logged by Scope signal viewer

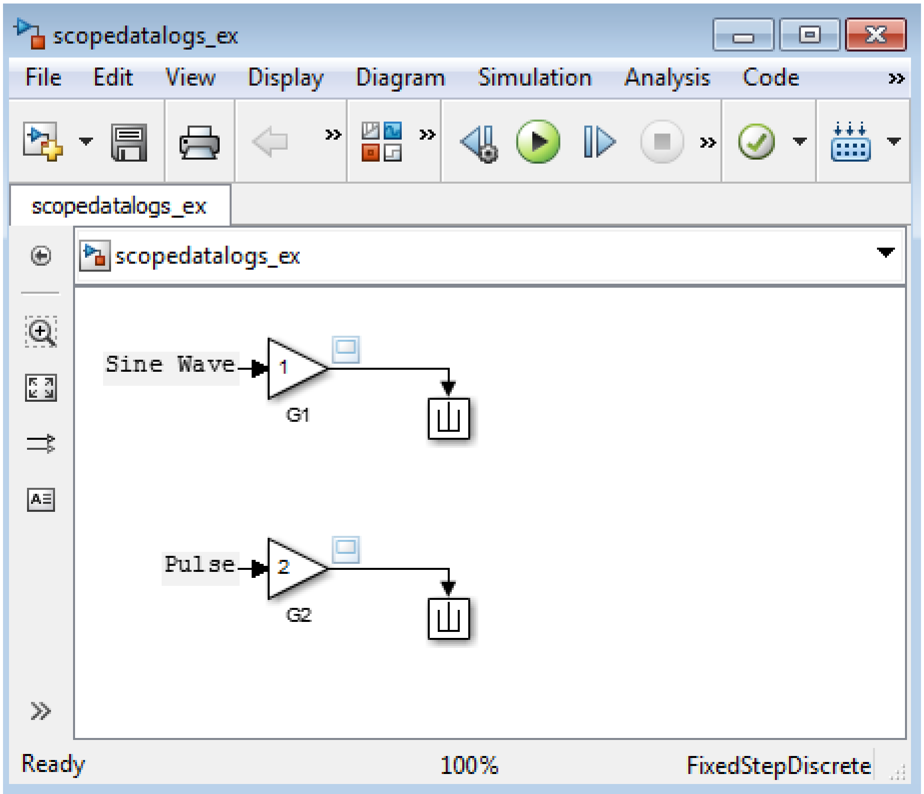
Description

Note The `Simulink.ScopeDataLogs` class is used in conjunction with the `ModelDataLogs` logging data format. The `ModelDataLogs` format is supported for backwards compatibility. The `ModelDataLogs` format will be removed in a future release.

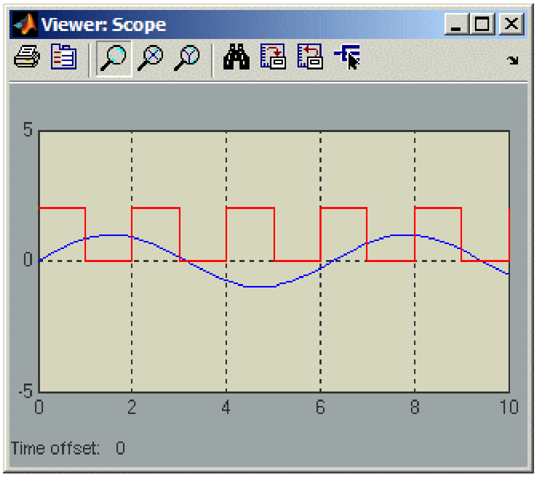
For new models, use the `Dataset` logging format. If you use the `Dataset` format for signal logging, then Simulink does not log the signals configured to be logged in the Signal Viewer. Explicitly mark the signal for signal logging by using the Signal Properties dialog box. To access the Signal Properties dialog box, right-click a signal and from the context menu, select **Properties**.

Simulink software creates instances of this class to log data displayed on Scope viewers (see “Visualize Results”). In particular, if you have enabled data logging for a model, Simulink software creates an instance of this class for each scope viewer enabled for logging in the model and assigns it to a property of the model’s `Simulink.ModelDataLogs` object. The instance created for each viewer has a Name property whose value is the name specified on the History pane of the viewer’s parameter dialog box (see Scope for more information). The instance also has an axes property for each of the scope’s axes labeled `Axes1`, `Axes2`, etc. The value of each axes property is itself a `Simulink.ScopeDataLogs` object that contains `Simulink.Timeseries` objects, one for each signal displayed on the axes. The time series objects contain the signal data displayed on the axes.

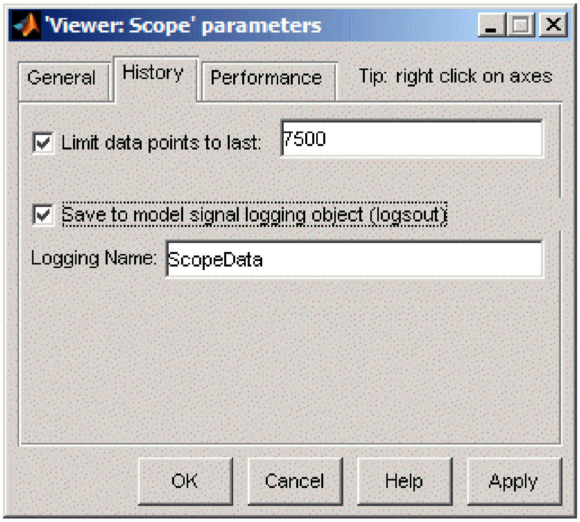
Consider, for example, the following model:


scopedatalogs_ex
File Edit View Display Diagram Simulation Analysis Code
scopedatalogs_ex
scopedatalogs_ex
Sine Wave
1
G1
Pulse
2
G2
Ready
100%
FixedStepDiscrete

This model displays signals `out1` and `out2` on a scope viewer that has only one set of axes.


Viewer: Scope
5
0
-5
0
2
4
6
8
10
Time offset: 0

The model enables data logging for the scope viewer under the variable name `ScopeData` and for the model as a whole under the default variable name `logsout`.


'Viewer: Scope' parameters
General
History
Performance
Tip: right click on axes
OK
Cancel
Help
Apply

☑ Limit data points to last: 7500

☑ Save to model signal logging object (logsout)

Logging Name: ScopeData

After simulation of the model, the MATLAB workspace contains a `Simulink.ModelDataLogs` object named `logsout` containing a `Simulink.ScopeDataLogs` object that in turn contains a `Simulink.ScopeDataLogs` object that contains `Simulink.Timeseries` objects that contain the times series data for signals `out1` and `out 2`.

You can use Simulink data object dot notation to access the data, e.g.,

```
>> logsout.ScopeData.axes1

ans =

Simulink.ScopeDataLogs (axes1):
  Name                   elements  Simulink Class

  out1                       1      Timeseries
  out2                       1      Timeseries
```

See Also

"Signal Logging", `Simulink.ModelDataLogs`, `Simulink.SubsysDataLogs`, `Simulink.Timeseries`, `Simulink.TsArray`, `Simulink.SimulationData.Dataset`, `who`, `whos`, `unpack`

Simulink.scopes.TimeScopeConfiguration

Purpose

Configure Time Scope for programmatic access

Description

The `Simulink.scopes.TimeScopeConfiguration` object contains the scope configuration information for the Time Scope block.

Construction

Call the `get_param` function, specifying a Time Scope block.

`htsc = get_param(gcbh,'ScopeConfiguration')` constructs a new Time Scope Configuration object.

Properties

ActiveDisplay

Active display for display-specific properties

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed. This property is Tunable.

Setting this property controls which display is used for `ShowGrid`, `ShowLegend`, `Title`, `PlotAsMagnitudePhase`, `YLabel`, and `YLimits`.

Default: `1`

AxesScaling

Specify how axes should be scaled when plotting data

Specify when the scope should automatically scale the axes. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools > Scale axes limits**.
  - Press the **Scale Axes Limits** toolbar button.

  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.

- `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
- `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. Selecting this option shows the **Number of updates** edit box.

This parameter is Tunable.

BufferLength

Specify number of data points in buffer

Specify the number of data points for each input signal that the scope holds in its memory cache. The number of data points equals the product of the number of channels in the signal and the number of time samples.

Default: `50000`

LayoutDimensions

Layout grid dimensions

Specify the layout grid dimensions as a 2-element vector: `[numberOfRows, numberOfColumns]`. You can use no more than four rows or four columns. This property is Tunable.

Default: `[1,1]`

MaximizeAxes

Maximize axes control

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display.

Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable.

Default: `'Auto'`

Name

Caption to display on the Time Scope window

Specify as a string the caption to display on the scope window. This property is Tunable.

Default: `'Time Scope'`

NumInputPorts

Number of input signals

Specify the number of input signals to display on the scope as a positive integer. You must invoke the `step` method with the same number of inputs as the value of this property.

Default: `1`

OpenAtSimulationStart

Open the scope when starting the simulation

Set this property to `true` to open the scope when the simulation starts. Set this property to `false` to prevent the scope from opening at the start of simulation.

Default: `true`

PlotAsMagnitudePhase

Plot signal magnitude and phase

When you set this property to `true`, the scope plots the magnitude and phase of the input signal on two separate axes within the same active display. When you set this property to `false`, the scope plots the real and imaginary parts of the input signal on two separate axes within the same active display. This property is particularly useful for complex-valued input signals. Selecting this check box affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees. This property is Tunable.

When set, `ActiveDisplay` controls which displays are updated. The active display shows the magnitude of the input signal on the top axes and its phase, in degrees, on the bottom axes.

Default: `false`

Position

Time Scope window position in pixels

Specify, in pixels, the size and location of the scope window as a 4-element double vector of the form, `[left bottom width height]`. You can place the scope window in a specific position on your screen by modifying the values to this property. This property is Tunable.

Default: The default depends on your screen resolution. By default, the Time Scope window appears in the center of your screen with a width of `410` pixels and height of `300` pixels.

ShowGrid

Option to enable or disable grid display

When you set this property to `true`, the grid appears. When you set this property to `false`, the grid is hidden. This property is Tunable.

When set, `ActiveDisplay` controls which display is updated.

Default: `false`

ShowLegend

When you set this property to `true`, the scope displays a legend with automatic string labels for each input channel. When you set this property to `false`, the scope does not display a legend. See FrameBasedProcessing for information on input channels. This property is Tunable.

When set, `ActiveDisplay` controls which display is updated.

Default: `false`

TimeAxisLabels

Time-axis labels

Specify how *time*-axis labels should appear in the scope displays as one of `'All'`, `'Bottom`, or `'None'`.

- When you set this property to `'All'`, *time*-axis labels appear in all displays.
- When you set this property to `'Bottom'`, *time*-axis labels appear in the bottom display of each column.

- When you set this property to `'None'`, there are no labels in any displays.

This property is Tunable.

Default: `'All'`

TimeDisplayOffset

Time display offset

Specify the offset, in seconds, to apply to the *time*-axis. This property can be either a numeric scalar or a vector of length equal to the number of input channels. If you specify this property as a scalar, then that value is the time display offset for all channels. If you specify a vector, each vector element is the time offset for the corresponding channel. For vectors with length less than the number of input channels, the time display offsets for the remaining channels are set to 0. If a vector has a length greater than the number of input channels, the extra vector elements are ignored. This property is Tunable.

See FrameBasedProcessing for information on input channels. See `TimeSpan` and `TimeSpanSource` for information on the *x*-axis limits and time span settings.

Default: `0`

TimeSpan

Time span

Specify the time span, in seconds, as a positive, numeric scalar value. This property applies when `FrameBasedProcessing` is `false`. This property also applies when `FrameBasedProcessing` is `true` and `TimeSpanSource` is `Property`. The *time*-axis limits are calculated as follows.

- Minimum *time*-axis limit = min(`TimeDisplayOffset`)

- Maximum *time*-axis limit = max(`TimeDisplayOffset`) + `TimeSpan`

where `TimeDisplayOffset` and `TimeSpan` are the values of their respective properties. This property is Tunable.

Default: `10`

TimeSpanOverrunAction

Wrap or scroll when the `TimeSpan` value is overrun

Specify how the scope displays new data beyond the visible time span. You can select one of the following options:

- `Wrap` — In this mode, the scope displays new data until the data reaches the maximum *time*-axis limit. When the data reaches the maximum *time*-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum *time*-axis limit.
- `Scroll` — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

This parameter is Tunable.

Default: `'Wrap'`

TimeUnits

Units of the *time*-axis

Specify the units used to describe the *time*-axis. You can select one of the following options:

- `Metric` — In this mode, the scope converts the times on the *time*-axis to the most appropriate measurement units. These can include milliseconds, microseconds, nanoseconds, minutes, days, etc. The scope chooses the appropriate measurement units based on the minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.
- `Seconds` — In this mode, the scope always displays the units on the *time*-axis as seconds.
- `None` — In this mode, the scope does not display any units on the *time*-axis. The scope only shows the word `Time` on the *time*-axis.

This property is Tunable.

Default: `'Metric'`

Title

Display title

Specify the display title as a string. This property is Tunable.

When set, `ActiveDisplay` controls which display is updated.

Default: `''`

YLabel

The label for the *y*-axis

Specify as a string the text for the scope to display to the left of the *y*-axis. This property is Tunable.

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two *y*-axis labels are read-only values. The *y*-axis labels are set to `'Magnitude'` and `'Phase'` for the magnitude plot and the phase plot, respectively. When set, `ActiveDisplay` controls which display is updated.

Default: `'Amplitude'` if `PlotAsMagnitudePhase` is `false`

YLimits

The limits for the *y*-axis

Specify the *y*-axis limits as a 2-element numeric vector, `[ymin ymax]`. This property is Tunable.

When `PlotAsMagnitudePhase` is `true`, this property specifies the *y*-axis limits of only the magnitude plot. The *y*-axis limits of the phase plot are always `[-180, 180]`. When set, `ActiveDisplay` controls which display is updated.

Default: `[-10, 10]`, if `PlotAsMagnitudePhase` is `false`, or `[0, 10]`, if `PlotAsMagnitudePhase` is `true`.

Examples

Example: Construct a Time Scope Configuration Object

Create a new Simulink model with a randomly-generated name.

```
sysname=char(randi(26,1,7)+96);
new_system(sysname);
```

Add a new Time Scope block to the model.

```
add_block('built-in/TimeScope',[sysname,'/TimeScope'])
```

Call the `get_param` function to retrieve the default Time Scope block properties.

```
htsc = get_param([sysname,'/TimeScope'],'ScopeConfiguration')
```

```
ans =

  TimeScopeConfiguration with properties:

                   Name: 'TimeScope'
          NumInputPorts: '1'
```

```
           BufferLength: '50000'
OpenAtSimulationStart: 0
               Position: [755 450 410 300]
           MaximizeAxes: 'Auto'
            AxesScaling: 'Manual'
               TimeSpan: 'Auto'
  TimeSpanOverrunAction: 'Wrap'
              TimeUnits: 'none'
      TimeDisplayOffset: '0'
         TimeAxisLabels: 'Bottom'
       LayoutDimensions: [1 1]
          ActiveDisplay: 1
                  Title: ''
             ShowLegend: 0
               ShowGrid: 1
   PlotAsMagnitudePhase: 0
                YLimits: [-10 10]
                 YLabel: 'Amplitude'
```

Related Examples

- “Control Time Scope Programmatically”

Simulink.sdi.DiffRunResult

Purpose

Results from comparing two simulation runs

Description

The `Simulink.sdi.DiffRunResult` class manages the results from comparing two simulation runs. A `Simulink.sdi.DiffRunResult` object contains a `Simulink.sdi.DiffSignalResult` object for each signal compared.

Construction

The function `Simulink.sdi.compareRuns` returns a handle to a `Simulink.sdi.DiffRunResult` object.

Properties

runID1

Run ID, a unique number identifying the first run compared.

runID2

Run ID, a unique number identifying the second run compared.

count

Number of compared signal results.

matlabVersion

Version of MATLAB used to create an instance of `Simulink.sdi.DiffRunResult`.

dateCreated

Date of object creation, stored in serial date number format.

Methods

getResultByIndex	Return signal comparison result, object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

The function `Simulink.sdi.compareRuns` returns a `Simulink.sdi.DiffRunResult` object containing the results of the comparison. The `Simulink.sdi.DiffRunResult` object contains a `Simulink.sdi.DiffSignalResult` object for each signal comparison between the two simulation runs.

```
% Configure model "slexAircraftExample" for logging and simulate
set_param('slexAircraftExample/Pilot','WaveForm','square');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
                  'SaveFormat', 'StructureWithTime', ...
                  'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run, Simulink.sdi.Run, from simOu
runID1 = Simulink.sdi.createRun('First Run','namevalue',{'simOut'},{si

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create another Simulation Data Inspector run
runID2 = Simulink.sdi.createRun('Second Run','namevalue',{'simOut'},{s

% Compare two runs and get an instance of Simulink.sdi.DiffRunResult
diff = Simulink.sdi.compareRuns(runID1, runID2);

% Get the number of signal comparison results
count = diff.count;

% Iterate over results and find out if signals match
for i=1:count
   % Get the Simulink.sdi.DiffSignalResult, diffSignal
  diffSignal = diff.getResultByIndex(i);
  signalID1  = diffSignal.signalID1;
   signalID2  = diffSignal.signalID2;
   match      = diffSignal.match;

   if match
```

```
      disp([num2str(signalID1) ' and ' num2str(signalID2)...
           ' match']);
   else
      disp([num2str(signalID1) ' and ' num2str(signalID2)...
           ' don''t match']);
   end
end
```

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.compareRuns` | `Simulink.sdi.DiffSignalResult`

How To

- “Record and Inspect Signal Data Programmatically”

Purpose

Return signal comparison result, `Simulink.sdi.DiffSignalResult` object

Syntax

```
diffSignalObj = diffRunObj.getResultByIndex(index)
```

Description

`diffSignalObj = diffRunObj.getResultByIndex(index)` returns the `Simulink.sdi.DiffSignalResult` object, `diffSignalObj`, which contains the comparison results for a signal. `diffRunObj` is an instance of a `Simulink.sdi.DiffRunResult` class, which contains an array of signal comparison results, where each element is an instance of a `Simulink.sdi.DiffSignalResult` class.

Input Arguments

index

An index to the array of `Simulink.sdi.DiffSignalResult` objects contained in a `Simulink.sdi.DiffRunResult` object.

Output Arguments

diffSignalObj

Handle to a `Simulink.sdi.DiffSignalResult` object representing the results of comparing two signals between simulation runs.

Examples

The function `Simulink.sdi.compareRuns` returns a `Simulink.sdi.DiffRunResult` object containing the results of the comparison. The `Simulink.sdi.DiffRunResult` object contains a `Simulink.sdi.DiffSignalResult` object for each signal comparison between the two simulation runs. `diff.getResultByIndex` returns the `Simulink.sdi.DiffSignalResult` object for each signal comparison.

```
% Configure model "slexAircraftExample" for logging and simulate
set_param('slexAircraftExample/Pilot','WaveForm','square');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run, Simulink.sdi.Run, from simO
```

```
runID1 = Simulink.sdi.createRun('First Run','namevalue',{'simOut'},{simOu

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create another Data Inspector run and get signal IDs
runID2 = Simulink.sdi.createRun('Second Run','namevalue',{'simOut'},{simO

% Compare two runs and get an instance of Simulink.sdi.DiffRunResult
diff = Simulink.sdi.compareRuns(runID1, runID2);

% Get the number of signal comparison results
count = diff.count;

% Iterate over results and display the comparison results
for i=1:count
  diffSignal = diff.getResultByIndex(i);
  signalID1  = diffSignal.signalID1;
   signalID2  = diffSignal.signalID2;
   match      = diffSignal.match;

   if match
      disp([num2str(signalID1) ' and ' num2str(signalID2)...
           ' match']);
   else
      disp([num2str(signalID1) ' and ' num2str(signalID2)...
           ' don''t match']);
   end
end
```

See Also

Simulink.sdi.Run | Simulink.sdi.Signal | Simulink.sdi.DiffRunResult | Simulink.sdi.DiffSignalResult | Simulink.sdi.compareRuns

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.DiffSignalResult

Purpose

Results from comparing two signals

Description

The `Simulink.sdi.DiffSignalResult` object manages the results from comparing two signals. A `Simulink.sdi.DiffSignalResult` object contains the value differences of the signals, the tolerance data, and the data after any specified synchronization methods are performed.

Construction

The function `Simulink.sdi.compareSignals` returns a handle to a `Simulink.sdi.DiffSignalResult` object, which contains the comparison results.

A `Simulink.sdi.DiffSignalResult` object is also created when the function `Simulink.sdi.compareRuns` creates a `Simulink.sdi.DiffRunResult` object, which in turn creates `Simulink.sdi.DiffSignalResult` objects.

Properties

signalID1

Signal ID, a unique number identifying the first signal compared.

signalID2

Signal ID, a unique number identifying the second signal compared.

match

A boolean indicating if the two `timeseries` objects match according to the specified tolerance and time synchronization options.

diff

A MATLAB `timeseries` object specifying the value differences after synchronizing the two time series data.

tol

A MATLAB `timeseries` object specifying the actual absolute tolerance value at each synchronized time point.

sync1

A MATLAB `timeseries` object specifying time series 1 after synchronization has been applied.

sync2

A MATLAB `timeseries` object specifying time series 2 after synchronization has been applied.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

In this example, a `Simulink.sdi.DiffSignalResult` object is created after comparing two signals in the Simulation Data Inspector.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run and get signal IDs
[~, ~, signalIDs] = Simulink.sdi.createRun('My Run','namevalue',{'MyDa

sig1 = signalIDs(1);
sig2 = signalIDs(2);

% Compare two signals, which returns the results in an instance of Si
diff = Simulink.sdi.compareSignals(sig1, sig2);

% Find if the signals match
match = diff.match;

% Get the tolerance used in Simulink.sdi.compareSignals
tolerance = diff.tol;
```

See Also

`Simulink.sdi.DiffRunResult` | `Simulink.sdi.createRun` | `Simulink.sdi.compareRuns`

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.Run

Purpose

Manages signal data and metadata of simulation run

Description

The `Simulink.sdi.Run` object contains the signal logging information for one simulation run, which includes the logged signal data, a run ID, and the total number of signals in the run.

Construction

The function `Simulink.sdi.createRun` creates a `Simulink.sdi.Run` object.

Properties

dateCreated

Date and time of the run, stored as a `double`.

description

User-specified description of the run. The default value is an empty string.

id

Read-only unique number to identify a run.

name

User-specified name of the run. The default value is an empty string.

signalCount

Number of signals in the run.

tag

User-specified string for categorization, identification, or attaching other information to this run. The default value is an empty string.

Methods

getSignal	Return object by signal ID
getSignalByIndex	Return object by index

getSignalIDByIndex	Return signal ID at array index
isValidSignalID	Determine if signal ID is valid within run

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

The `Simulink.sdi.Run` object contains a `Simulink.sdi.Signal` object for each logged signal. This example creates a run from simulation data in the base workspace. It demonstrates how to access the `Simulink.sdi.Run` object from the Simulation Data Inspector . You can select which signals to view and then open the Simulation Data Inspector tool to inspect those signals.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
                  'SaveFormat', 'StructureWithTime', ...
                  'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run, Simulink.sdi.Run, from simOut i
runID = Simulink.sdi.createRun('My Run', 'base', {'simOut'});

% Get the handle to the run object, Simulink.sdi.Run, corresponding to th
runObj = Simulink.sdi.getRun(runID);

% Get the name of the run
runName = runObj.name;

% Get number of signals
numSignals = runObj.signalCount;

% To view the all of the signals in the run, select each signal in the ru
for i=1:numSignals
  signalObj = runObj.getSignalByIndex(i);
  signalObj.checked = true;
end
```

```
% Open the Simulation Data Inspector to inspect the selected signals.
Simulink.sdi.view;
```

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRun` | `Simulink.sdi.view`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.sdi.Run.getSignal

Purpose

Return `Simulink.sdi.Signal` object by signal ID

Syntax

```
signalObj = runObj.getSignal(signalID)
```

Description

`signalObj = runObj.getSignal(signalID)` returns the `Simulink.sdi.Signal` object, `signalObj`, corresponding to the signal ID, `signalID`, stored in the `Simulink.sdi.Run` object, `runObj`.

Input Arguments

signalID

Signal ID, a unique number identifying a signal in a run in the Simulation Data Inspector.

Output Arguments

signalObj

Handle to a `Simulink.sdi.Signal` object representing a signal in a run object, `Simulink.sdi.Run`.

Examples

The `Simulink.sdi.Run` method, `getSignal`, returns a signal object representing the signal data and metadata in a run. You can modify the signal object properties to configure the signal for plotting or comparing to other signals in the Simulation Data Inspector.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
               'SaveFormat', 'StructureWithTime', ...
               'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run
[runID,runIndex,signalIDs] = Simulink.sdi.createRun('My Run','base',{'sim

% Get the Simulink.sdi.Run object corresponding to the new run ID
runObj = Simulink.sdi.getRun(runID);

% Get the number of signals in the run
numSignals = runObj.signalCount;
```

```
% Get the Simulink.sdi.Signal objects for each signal in the run and s
for i = 1:numSignals
   signalObjs(i) = runObj.getSignal(signalIDs(i));
   signalObjs(i).checked = true;
end
```

See Also

`Simulink.sdi.Run` | `Simulink.sdi.Signal` | `Simulink.sdi.createRun`

How To

- “Record and Inspect Signal Data Programmatically”

Simulink.sdi.Run.getSignalByIndex

Purpose

Return `Simulink.sdi.Signal` object by index

Syntax

```
signalObj = runObj(index)
```

Description

`signalObj = runObj(index)` returns the `Simulink.sdi.Signal` object, `signalObj`, at the index into the array of signals contained in the `Simulink.sdi.Run` object, `runObj`.

Input Arguments

index

Number specifying an index to the array of signals contained in a `Simulink.sdi.Run` object. The first index in the array is 1.

Output Arguments

signalObj

Handle to a `Simulink.sdi.Signal` object representing a signal in a run object, `Simulink.sdi.Run`.

Examples

The `Simulink.sdi.Run` method, `getSignalByIndex`, returns a signal object representing the signal data and information in a run. You can modify the signal object properties to configure the signal for plotting or comparing to other signals in the Simulation Data Inspector.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat', 'Struc
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Get the Simulink.sdi.Run object corresponding to the new run ID
runObj = Simulink.sdi.getRun(runID);

% Get the number of signals in the run
numSignals = runObj.signalCount;

% Get the Simulink.sdi.Signal object for the first signal in the run
```

```
if numSignals > 0
   signalObj = runObj.getSignalByIndex(1);
end
```

See Also

`Simulink.sdi.Signal` | `Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.getRun`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.sdi.Run.getSignalIDByIndex

Purpose

Return signal ID at array index

Syntax

```
signalID = runObj.getSignalIDByIndex(index)
```

Description

`signalID = runObj.getSignalIDByIndex(index)` returns the signal ID for the signal at the specified index to the array of signals contained in the `Simulink.sdi.Run` object, `runObj`.

Input Arguments

index

Number specifying the index to the array of signals contained in a `Simulink.sdi.Run` object. The first index in the array is 1.

Output Arguments

signalID

Signal ID, a unique number identifying a signal in a run.

Examples

The `Simulink.sdi.Run` method, `getSignalIDByIndex`, returns the signal ID corresponding to a signal in a run. With the signal ID you can get the signal object representing the signal data and metadata. You can compare two signals by passing their signal IDs to `Simulink.sdi.compareSignals`.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Get the Simulink.sdi.Run object corresponding to the new run ID
runObj = Simulink.sdi.getRun(runID);

% Get the number of signals in the run
numSignals = runObj.signalCount;
```

```
% Get the signal ID for the first signal in the run
if numSignals > 0
   signalID = runObj.getSignalIDByIndex(1);
end
```

See Also

`Simulink.sdi.Signal` | `Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.getRun` | `Simulink.sdi.compareSignals`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.sdi.Run.isValidSignalID

Purpose

Determine if signal ID is valid within run

Syntax

`isValid = runObj.isValidSignalID(signalID)`

Description

`isValid = runObj.isValidSignalID(signalID)` returns `true` if the signal ID, `signalID`, corresponds to a signal in the run object, `runObj`. Otherwise, it returns `false`.

Input Arguments

signalID

Signal ID, a unique number identifying a signal stored in the run object.

Output Arguments

isValid

A Boolean value: `true`, if the signal exists; `false`, otherwise.

Examples

Before calling a function that takes a signal ID as input, verify that the signal ID is valid.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Get the Simulink.sdi.Run object corresponding to the new run ID
runObj = Simulink.sdi.getRun(runID);

% Get the number of signals in the run
numSignals = runObj.signalCount;

% Get the signal ID for the first signal in the run
if numSignals > 0
   signalID = runObj.getSignalIDByIndex(1);
```

```
end

% Before calling getSignal, check that the signalID is valid
if runObj.isValidSignalID(signalID)
    signalObj = runObj.getSignal(signalID)
end
```

See Also

`Simulink.sdi.Signal` | `Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.getRun` | `Simulink.sdi.Run.getSignalByIndex` | `Simulink.sdi.Run.isValidSignalID` | `Simulink.sdi.Run.getSignal`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.sdi.Signal

Purpose

Manages signal time series data and metadata

Description

The `Simulink.sdi.Signal` object contains the signal logging information for one simulation run. It also contains properties for visualizing and comparing signals.

Construction

The function `Simulink.sdi.createRun` creates a `Simulink.sdi.Run` object, which creates a `Simulink.sdi.Signal` object for each signal in the simulation output.

Properties

Signal Properties (read only)

id

Signal ID, a unique number identifying the signal.

runID

Run ID, a unique number identifying the signal's parent run.

rootSource

String to access the high-level logging structure this signal was derived from.

timeSource

String to access the logged signal's time vector.

dataSource

String to access the logged signal's data values.

dataValues

Time series data for this signal.

blockSource

String specifying the path to the block that defines the signal.

modelSource

String specifying the name of the model that defines the signal.

signalLabel

String specifying the name of the signal.

timeDim

For any given data sample, the time dimension.

sampleDims

Dimensions of a data sample.

portIndex

Index of the port that defines the signal.

channel

Channel index. This property applies to matrix data only. Matrix data is flattened into a scalar time series by using channels.

SID

"Simulink Identifier" on page 7-2 of the block that defines the signal.

Comparison Properties (read and write)

Each signal has properties that the Simulation Data Inspector uses for comparing two signals. The Simulation Data Inspector uses the comparison properties from the first signal passed in, also called the reference signal.

absTol

Absolute tolerance of the signal. Must be a positive number.

relTol

Relative tolerance of the signal. Must be a positive number.

syncMethod

Time synchronization method to align time vector when comparing signals. Possible values are: `'intersection'`, `'uniform'`, and `'union'`.

interpMethod

Interpolation method to align data. Possible values are: `'zoh'` and `'linear'`.

Visualization Properties (read and write)

lineColor

Signal line color in the plot in the form of a vector `[r g b]`. `r` is the red component, `g` the green component, and `b` the blue component.

lineDashed

Signal line format in the plot

checked

Boolean specifying if the signal is selected for plotting: `true` for selected and `false` for cleared.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a run and call the `Simulink.sdi.getSignal` function to get a `Simulink.sdi.Signal` object.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
                   'SaveFormat', 'StructureWithTime', ...
                   'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run which returns a list of signal I
[~,~,signalIDs] = Simulink.sdi.createRun('My Run', 'base', {'simOut'});

% Get the signal object corresponding to the first signal ID
```

```
signalObj = Simulink.sdi.getSignal(signalIDs(1));

% signalObj is an instance of Simulink.sdi.Signal. Get the run ID for
runID = signalObj.runID;

% Modify or define comparison and visualization properties for this si
signalObj.syncMethod = 'intersection';
signalObj.lineColor = [1,0.4,0.6];
signalObj.lineDashed = '-';
signalObj.checked = true;

% View signals in the Simulation Data Inspector tool
Simulink.sdi.view;
```

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.getSignal`

How To

- "Record and Inspect Signal Data Programmatically"

Simulink.Signal

Purpose

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.

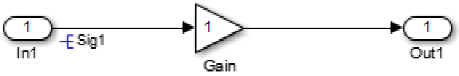
Signal Specification Block: An Alternative to Simulink.Signal

You can use a `Signal Specification` block rather than a `Simulink.Signal` object to assign properties left unspecified by a signal source. Each technique has advantages and disadvantages:

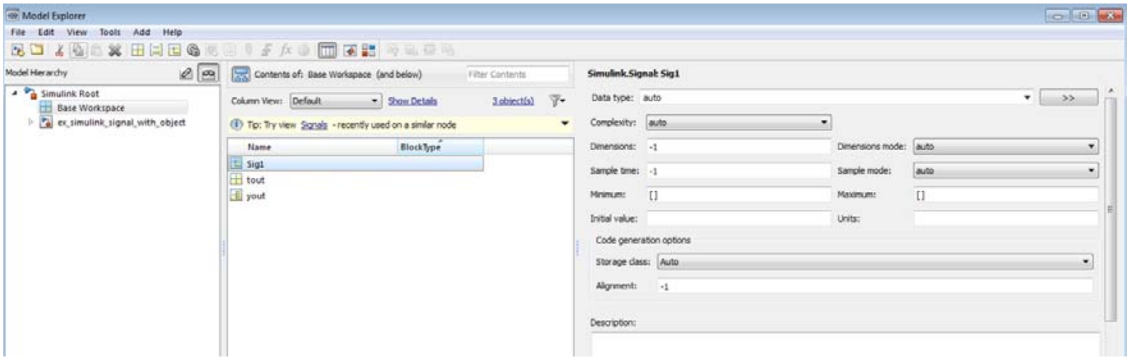
- Using a signal object simplifies the model and allows you to change signal property values without editing the model, but does not show signal property values directly in the block diagram.
- Using a Signal Specification block displays signal property values directly in the block diagram, but complicates the model and requires editing it to change signal property values.
- You can use a Signal Specification block with virtual and nonvirtual buses; you can use only nonvirtual buses with a `Simulink.Signal` object.

The following two models illustrate the respective advantages of the two ways of assigning attributes to a signal.

In the first example, the signal object named `Sig1` specifies the sample time and data type of the signal emitted by input port `In1`.



To determine the properties of the `Sig1` signal, you can view the signal object in the Model Explorer. In this model, the sample time is `-1` and the data type is `auto`.



- Simulink Root
  - Base Workspace
  - ex_simulink_signal_with_object

Contents of: Base Workspace (and below)

Column View: Default

Tip: Try view Signals - recently used on a similar node

Name	BlockType
Sig1	
tout	
yout	

Simulink.Signal: Sig1

Data type: auto

Complexity: auto

Dimensions: -1

Dimensions mode: auto

Sample time: -1

Sample mode: auto

Minimum: []

Maximum: []

Initial value:

Units:

Code generation options

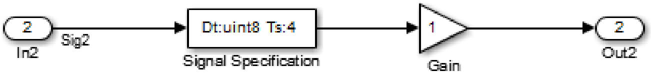
Storage class: Auto

Alignment: -1

Description:

Using a signal object to specify the sample time and data type properties of signal `Sig1` allows you to change the sample time or data type without having to edit the model. For example, you could use the Model Explorer, the MATLAB command line, or a MATLAB program to change these properties.

The second example uses a Signal Specification block specifies the sample time and data type of the signal emitted by input port `In2`. The Signal Specification block displays the data type and signal sample time properties right in the diagram, which in this case are `uint8` and `4`, respectively.



Using Signal Objects to Assign or Validate Signal Attributes

This section describes how you can use signal objects to assign or validate signal attributes. The same techniques work with discrete states also. To use a signal object to assign or validate signal attribute values:

1. Create a `Simulink.Signal` object that has the same name as the signal to which you want to assign attributes or whose attributes you want to validate.
    1. Open the Model Explorer.
    2. In the Model Hierarchy pane, select either the Base workspace or Model workspace node, depending on the context you want for the signal object. If you create the signal object in a model workspace, you must set the **Storage class** parameter to `Auto`.
    3. Select **Add > Simulink Signal**.
2. Set the properties of the object that correspond to the attributes left unspecified by the signal source, or that correspond to the attributes you want to validate. See “Property Dialog Box” on page 5-227 for details.
3. Enable explicit or implicit signal resolution:
    - **Explicit resolution:** In the Signal Properties dialog box for the signal, enable **Signal name must resolve to Simulink signal object**. This is the preferred technique. See “Explicit and Implicit Symbol Resolution” for more information.
    - **Implicit resolution:** Set the **Configuration Parameters > Diagnostics > Data Validity > Signal** resolution option for

the model to `Explicit and implicit` or `Explicit and warn implicit`. Explicit resolution is the preferred technique.

4. Assign the signal object to a workspace variable.

5. Associate the signal object with the source signal.

   - Give the signal the same name as the workspace variable that references the signal object.

   - You can use a variety of techniques to associate a signal object with a signal. For examples, see “Using Signal Objects to Initialize Signals and Discrete States”, “Using Signal Objects to Tune Initial Values”, and “Apply Custom Storage Classes”.

Validation

The result when a signal does not match a signal object can depend on several factors. Simulink software can validate a signal property when you update the diagram, while you run a simulation, or both. When and how validation occurs can depend on internal rules that are subject to change, and sometimes on configuration parameter settings.

Not all signal validation compares signal source attributes with signal object properties. For example, if you specify **Minimum** and **Maximum** signal values using a signal object, the signal source must specify the same values as the signal object (or inherit the values from the object) but such validation relates only to agreement between the source and the object, not to enforcement of the minimum and maximum values during simulation.

If the value of **Configuration Parameters > Diagnostics > Data Validity > Simulation range checking** is `none` (the default), Simulink does not enforce any minimum and maximum signal values during simulation, even though a signal object provided or validated them. To enforce minimum and maximum signal values during simulation, set **Simulation range checking** to `warning` or `error`. See “Signal Ranges” and “Diagnostics Pane: Data Validity” for more information.

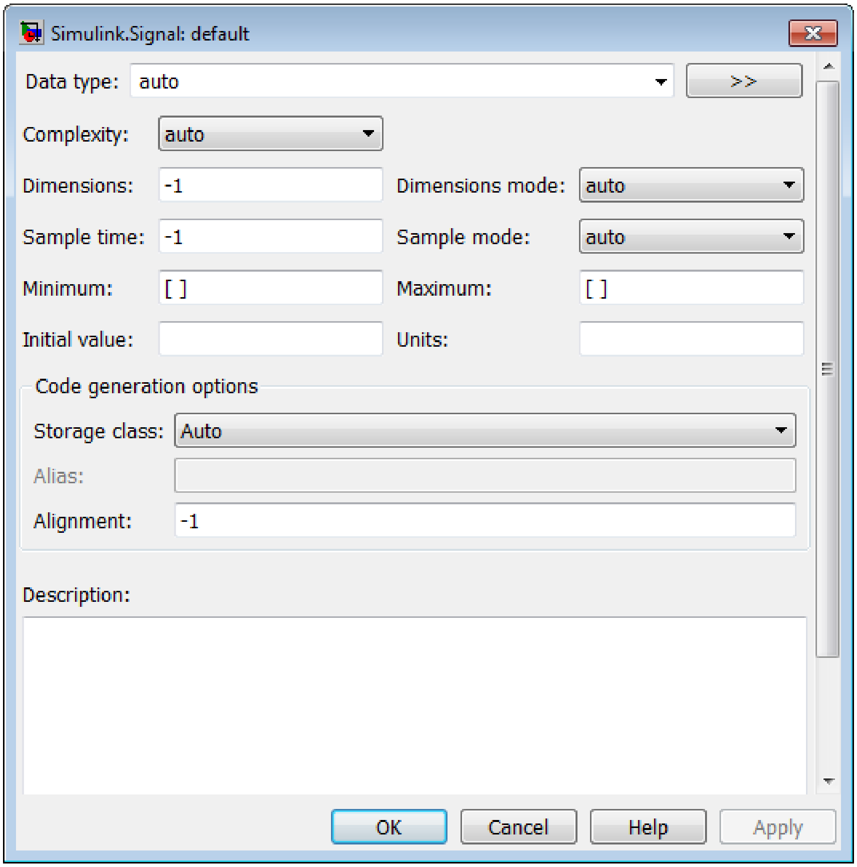
Multiple Signal Objects

You can associate a given *signal object* with more than one signal if the storage class of the signal object is `Auto`. If the storage class of the object is other than `Auto`, you can associate the signal object with no more than one signal.

You can associate a given *signal* with no more than one signal object. The signal can refer to the signal object more than once, but every reference must resolve to exactly the same signal object. Referencing two different signal objects that have exactly the same properties causes a compile-time error.

A compile-time error occurs if a model associates more than one signal object with any signal. To prevent the error, decide which object you want the signal to use, then delete or reconfigure all references to any other signal objects, so that all remaining references resolve to the chosen signal object. See "Display Signal Sources and Destinations" for a description of techniques that you can use to trace the full extent of a signal.

Property Dialog Box



Simulink.Signal: default

Data type: auto

Complexity: auto

Dimensions: -1

Dimensions mode: auto

Sample time: -1

Sample mode: auto

Minimum: [ ]

Maximum: [ ]

Initial value:

Units:

Code generation options

Storage class: Auto

Alias:

Alignment: -1

Description:

Data type

Data type of the signal. The default entry, `auto`, specifies that Simulink should determine the data type. Use the adjacent pulldown list to specify built-in data types (for example, `uint8`). To specify a custom data type, enter a MATLAB expression that specifies the type, (for example, a base workspace variable that references a `Simulink.NumericType` object).

To specify a bus object as the data type for the signal object, use the `Bus: <object_name>` option. See “Bus Support” on page 5-232 for details about what you need to do if you specify a bus object as the data type.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Data type** parameter. (See “Specify Data Types Using Data Type Assistant” in *Simulink User’s Guide*.)

Complexity

Numeric type of the signal. Valid values are `auto` (determined by Simulink), `real`, or `complex`.

Dimensions

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.

Dimensions mode

Dimensions mode of this signal. From the drop-down list, select

- 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.

Sample time

Rate at which the value of this signal should be computed. See “Specify Sample Time” for details.

Sample mode

Sample mode of this signal. Simulink ignores the setting of this field.

Minimum

Minimum value that the signal should 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").

Maximum

Maximum value that the signal should 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").

Initial value

Signal or state value before a simulation takes its first time step. You can specify any MATLAB string expression that evaluates to a `double` numeric scalar value or array.

Valid:

```
1.5
[1 2 3]
1+0.5

foo = 1.5;
s1.InitialValue = 'foo';
```

Invalid:

```
uint(1)
foo = '1.5';
s1.InitialValue = 'foo';
```

If you use a bus object as the data type for the signal object, set **Initial value** to a string containing either `0` or a MATLAB structure that matches the bus object. See "Bus Support" on page 5-232 for details.

If the initial value evaluates to a MATLAB structure, then in the **Configuration Parameters > Diagnostics > Data Validity** pane, 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 (which cannot be specified as `[]`) are used instead.

- Outport blocks of conditionally executed subsystems
- Merge blocks

Units

Measurement units in which the value of this signal is expressed, (for example, inches). You can use this parameter to document this signal. Simulink ignores it during simulation.

Storage class

Storage class of this signal. For more information, see “Simulink Package Custom Storage Classes”.

If you create the signal object in a model workspace, you must set the object storage class to `Auto`.

Alias

Alternate name for this signal. Simulink ignores this setting. This property is used for code generation.

Alignment

Data alignment boundary, specified in number of bytes. The starting memory address for the data allocated for the signal will be a multiple of the **Alignment** setting. The default value is `-1`, which specifies that the code generator should determine an optimal alignment based on usage. Otherwise, specify a positive integer that is a power of 2, not exceeding 128. This field is intended for use by Simulink Coder software. See "Configure Data Alignment for Function Implementations". Simulink software ignores this setting.

Description

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.

Bus Support

Using Bus Objects as the Data Type

`Simulink.Signal` supports nonvirtual buses as the output data type.

If you set the **Data type** of the signal object to be a bus object, then you cannot associate the signal object with a non-bus signal.

Using Structures for the Initial Value

If you use a bus object as the data type, set **Initial value** to `0` or a MATLAB structure that matches the bus object.

The structure you specify must contain a value for every element of the bus represented by the bus object.

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.

Setting Configuration Parameters to Support Using a Bus Object Data Type

To enable the use of a bus object as the signal object data type, before you start a simulation, in the **Configuration Parameters >**

Diagnostics > Connectivity pane, set “Mux blocks used to create bus signals” to `error`. The documentation for that diagnostic explains how convert your model to handle error messages the diagnostic generates.

Properties

Name	Access	Description
`CoderInfo`	`RW`	Information used by Simulink Coder for generating code for this signal. The value of this property is an object of `Simulink.SignalCoderInfo` class.
`Description`	`RW`	Description of this signal. This field is intended for use in documenting this signal. (Description)
`DataType`	`RW`	String specifying the data type of this signal. (Data type)
`Min`	`RW`	Minimum value that this signal can have. (Minimum)
`Max`	`RW`	Maximum value that this signal can have. (Maximum)
`DocUnits`	`RW`	Measurement units used for expressing this signal value. (Units)
`Dimensions`	`RW`	Scalar or vector specifying the dimensions of this signal. (Dimensions)
`Complexity`	`RW`	String specifying the numeric type of this signal. Valid values are `'auto'`, `'real'`, or `'complex'`. (Complexity)
`SampleTime`	`RW`	Rate at which this signal should be updated. (Sample time)
`Sampling Mode`	`RW`	Sampling mode of this signal. (Sample mode)
`InitialValue`	`RW`	Signal or state value before a simulation takes its first time step. (Initial Value)

Purpose

Specify information needed to generate code for signal

Description

Simulink software creates an instance of this class for each instance of a `Simulink.Signal` object that it creates. Simulink software uses the `Simulink.SignalCoderInfo` object to store information needed to generate code for the signal specified by the `Simulink.Signal` object.

You can set the properties of an instance of this class via the `CoderInfo` property or the property dialog box of the `Simulink.Signal` object that uses it. For example, the following MATLAB expression sets the `StorageClass` property of a `Simulink.SignalCoderInfo` object named `mysignal`.

```
mysignal.CoderInfo.StorageClass = 'exportedGlobal';
```

Property Dialog Box

Use the Code Generation Options section of the `Simulink.Signal` property dialog box to set the `StorageClass`, `Alias`, and `Alignment` properties of objects of this class.

Properties

Name	Description
`Alias`	Alternate name for this signal.
`Alignment`	Data alignment boundary for this signal. See “Configure Data Alignment for Function Implementations” in the Embedded Coder documentation for more information.
`CustomAttributes`	Custom storage class attributes of this signal. See “Introduction to Custom Storage Classes” in the Embedded Coder documentation for more information.
`CustomStorageClass`	Custom storage class of this signal.
`StorageClass`	Storage class of this signal. For more information, see “Signals” in the Simulink Coder documentation.

Purpose

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`	Individual component within the block specified by the block path

Method Summary

Name	Description
`BlockPath`	Create a block path.
`convertToCell`	Convert a block path to a cell array of strings.

Name	Description
`getBlock`	Get a single block path in the model reference hierarchy.
`getLength`	Get the length of the block path.

Properties

`SubPath`

Description

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'
```

Data Type

string

Access

`RW`

Methods

`BlockPath`

Purpose

Create block path

Syntax

```
blockpath_object = Simulink.SimulationData.BlockPath()
blockpath_object = Simulink.SimulationData.BlockPath(blockpath)
blockpath_object = Simulink.SimulationData.BlockPath(paths)
blockpath_object = Simulink.SimulationData.BlockPath(paths, subpath)
```

Input Arguments

`blockpath`

The block path object that you want to copy.

`paths`

A string or cell array of strings that Simulink uses to build the block path.

Specify each string in order, from the top model to the specific block for which you are creating a block path.

Each string 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 string, 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 string

`subpath`

A string that represents an individual component within a block.

Output Arguments

`blockpath_object`

The block path that you create.

Description

`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 string or cell array of strings that you specify with the `paths` argument. Each string represents a path at a level of model hierarchy.

`blockpath = Simulink.SimulationData.BlockPath(paths, subpath)` creates a block path from the string or cell array of strings that you specify with the `paths` argument and creates a path for the individual component (for example, a signal) of the block.

Example

Create a block path object called `bp1`, using a cell array of strings 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`

Purpose

Convert block path to cell array of strings

Syntax

```
cellarray = Simulink.SimulationData.BlockPath.convertToCell()
```

Output Arguments

`cellarray`

The cell array of strings representing the elements of the block path.

Description

`cellarray = Simulink.SimulationData.BlockPath.convertToCell()` converts a block path to a cell array of strings.

Examples

```
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`

Purpose

Get single block path in model reference hierarchy

Syntax

`block = Simulink.SimulationData.BlockPath.getBlock(index)`

Input Arguments

`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.

Output Arguments

`block`

The block representing the level in the model reference hierarchy specified by the `index` argument.

Description

`blockpath = Simulink.SimulationData.BlockPath.getBlock(index)` returns the block path of the block specified by the `index` argument.

Example

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`

Purpose

Get length of block path

Syntax

`length = Simulink.SimulationData.BlockPath.getLength()`

Output Arguments

`length`

The length of the block path. The length is the number of levels in the model reference hierarchy.

Description

`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.

Example

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`

How To

- "Specify the Instance That Has Normal Mode Visibility"

Simulink.SimulationData.Dataset

Purpose

Container for logged data using MATLAB `timeseries` objects

Description

Simulink creates `Simulink.SimulationData.Dataset` objects to store data elements when:

- Performing signal logging, if you use the default signal logging format of `Dataset`
- Logging a data store

Using the `Dataset` format for signal logging offers several advantages, compared to the `ModelDataLogs` format. For a discussion of those advantages, see "Signal logging format".

`Simulink.SimulationData.Dataset` provides a `getElement` method for accessing individual elements in the dataset. You can specify an element by index, name, or block path.

To select the `Dataset` format for signal logging, use *one* of these approaches:

- Set the **Configuration Parameters > Data Import/Export > Signal logging format** parameter to `Dataset`. For details, see "Signal logging format".
- From the MATLAB command line, set the `SignalLoggingSaveFormat` parameter to `Dataset`.

Unlike the handle-based `ModelDataLogs` class, `Simulink.SimulationData.Dataset` is a value class. For details, see "Handle Versus Value Classes".

If you have logged signal data in the `ModelDataLogs` format, you can use the `Simulink.ModelDataLogs.convertToDataset` function to update the `ModelDataLogs` data to `Dataset` format.

Simulink checks signal logging data format consistency for certain model referencing configurations. For details, see "Model Reference Signal Logging Format Consistency". You can use the Upgrade Advisor (with the `upgradeadvisor` function) to upgrade a model to use `Dataset` format.

An alternative approach for handling reported inconsistencies is to use the `Simulink.SimulationData.updateDatasetFormatLogging` function to update the models to use `Dataset` format. This approach sets the **Model Configuration Parameters > Data Import/Export > Signal logging format** parameter to `Dataset` for each referenced model and each variant.

If you have logged signal data in the `ModelDataLogs` format, you can use the `Simulink.ModelDataLogs.convertToDataset` function to convert the `ModelDataLogs` data to `Dataset` format.

For details about migrating an existing model that uses `ModelDataLogs` format to use `Dataset` format, see "Migrate from ModelDataLogs to Dataset Format".

Property Summary

Name	Description
`Name`	Logged dataset name.

Method Summary

Name	Description
`addElement`	Add an element to the end of a dataset.
`get` The `getElement` method uses the same syntax and behavior as the get method	Get element or collection of elements from the dataset, based on index, name, or block path.
`getElementNames`	Get the names of dataset elements.
`numElements`	Get number of elements in the dataset.
`setElement`	Change the element stored at the specified index

Properties

`Name`

Description

The name of the dataset. The default is an empty string.

Specify a name when you want to easily distinguish one dataset from another. For example, if you have signal logging in model reference hierarchy, name the signal logging dataset for each model to make the output easier to analyze.

The following example shows how to give a dataset the name of `Dataset1`.

```
ds = Simulink.SimulationData.Dataset
ds.Name = 'Dataset1'
```

Data Type

string

Access

`RW`

Methods

`addElement`

Purpose

Add element to end of dataset.

Syntax

```
dataset = dataset.addElement(element)
dataset = dataset.addElement(element,name)
```

Input Arguments

`element`

The element to add to the dataset. The `element` must be one of the following:

- `Simulink.SimulationData.Signal` object
- `Simulink.SimulationData.DataStoreMemory` object

`name`

A string specifying a name for the element.

Output Arguments

`dataset`

The dataset to which you add the element. The new element is added to the end of the dataset.

Description

`dataset = Simulink.SimulationData.Dataset.addElement(element)` adds an element to the `Simulink.SimulationData.Dataset` dataset.

`dataset = Simulink.SimulationData.Dataset.addElement(element, name)` adds an element to the `Simulink.SimulationData.Dataset` dataset. This element receives the name that you specify with the `name` argument. You can use `name` to identify an element that does not have a name. If an object that already has a name (for example, a `Simulink.SimulationData.Signal` object to which you assigned a name), the element uses that name unless you specify a different name by using with the `name` argument.

Examples

Create a dataset and add three elements to that dataset.

```
time = 0.1*(0:100)';
ds = Simulink.SimulationData.Dataset;
element1 = Simulink.SimulationData.Signal;
element1.Name = 'A';
element1.Values = timeseries(sin(time), time);
ds = ds.addElement(element1);
element2 = Simulink.SimulationData.Signal;
element2.Name = 'B';
element2.Values = timeseries(2*sin(time), time);
ds = ds.addElement(element2);
element3 = Simulink.SimulationData.Signal;
element3.Name = 'C';
```

```
element3.Values = timeseries(3*sin(time), time);
ds = ds.addElement(element3)
```

The output is:

```
ds =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'topOut'
    Total Elements: 3

  Elements:
    1: 'A'
    2: 'B'
    3: 'C'
```

`get`

Purpose

Get element or collection of elements from dataset, based on index, name, or block path

The `getElement` method shares the same syntax and behavior as the `get` method.

Syntax

```
element = dataset.get(search_arg)
element = dataset.get(search_arg,'-blockpath')
```

Input Arguments

`search_arg`

The index, name, or block path that identifies the element to access. To access an element by:

- Index — Set `search_arg` to a scalar numeric value.

- Name — Set `search_arg` to a character array.
- Block path — Specify a `Simulink.SimulationData.BlockPath` object. For signal logging, specify the blockpath for the source block for the element.

When specifying a name or block path for `search_arg` in cases where there could be duplicate names or block paths, use a cell array to consistently return a `Simulink.SimulationData.Dataset` object. (Such duplication could occur, for example, in a model reference hierarchy.)

`'-blockpath'`

Specify a block path directly. Set `search_arg` to a character array representing the full block path to the element, followed by `'-blockpath'`.

Output Arguments

`element`

The element that the `getElement` method adds to the dataset. The returned value depends on the `search_arg` format.

search_arg Argument Value	Returned Value
Single-character array or `Simulink.SimulationData.BlockPath` object	Single element if only one element is found; dataset if more than one element with the specified name exists
Cell array containing one string or `Simulink.SimulationData.BlockPath` object	Dataset

Description

`element = Simulink.SimulationData.Dataset.get(search_arg)` returns the element corresponding the index, name, or block path that you specify with `search_arg`.

`element = Simulink.SimulationData.Dataset.get(search_arg, '-blockpath')` returns the element corresponding the `search_arg` character array that identifies the block path for the element that you want to access.

Examples

Access dataset elements in the top model of the `ex_bus_logging` model. The signal logging dataset is `topOut`.

```
ex_bus_logging
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 the first element of the dataset, by specifying an index for `search_arg`.

```
element1 = topOut.get(1)
```

Access the second element of the dataset, by using a cell array to specify the name for `search_arg`.

```
element2 = topOut.get({'OUTPUTBUS'})
```

Access the third element of the dataset by block path. In the Simulink Editor, select the IncrementBusCreator block. At the MATLAB command line, use `gcb` for `search_arg` and use the `'-blockpath'` argument.

```
element3 = topOut.get(gcb,'-blockpath')
close_system('ex_bus_logging');
close_system('ex_mdlref_counter_bus');
```

`getElementNames`

Purpose

Return the names of all of the elements in the dataset.

Syntax

```
element_list = dataset.getElementNames
```

Input Arguments

None

Output Arguments

`element_list`

A cell array of the strings containing names of all of the elements of the dataset.

Description

`element_list = Simulink.SimulationData.Dataset.getElementNames(element)` returns the names of all of the elements in the `Simulink.SimulationData.Dataset` object.

Examples

Return the names of the elements for the `topOut` dataset (the signal logging data).

```
ex_bus_logging
sim('ex_bus_logging')
el_names = topOut.getElementNames

el_names =

    'COUNTERBUS'
    'OUTPUTBUS'
    'INCREMENTBUS'
    'inner_bus'

close_system('ex_bus_logging');
close_system('ex_mdlref_counter_bus');
```

`numElements`

Purpose

Get number of elements in dataset.

Syntax

```
length = Simulink.SimulationData.Dataset.numElements()
```

Description

`length = Simulink.SimulationData.Dataset.numElements()` Get the number of elements in the top-level dataset. To get the number of elements of a nested dataset, use `numElements` with the nested dataset.

Example

Get the number of elements in the signal logging dataset for the `ex_bus_logging`.

```
length = topOut.numElements()
```

`setElement`

Purpose

Change element stored at specified index.

Syntax

```
dataset = Simulink.SimulationData.Dataset.setElement(index,...
element)
dataset = Simulink.SimulationData.Dataset.setElement(index,...
element,name)
```

Input Arguments

`index`

A scalar numeric value specifying the index for the added element. The value must be between 1 and the number of elements plus 1.

`element`

The element to replace an existing element or to add to the dataset. The `element` must be one of the following:

- `Simulink.SimulationData.Signal` object
- `Simulink.SimulationData.DataStoreMemory` object

`name`

A string specifying a name for the element.

Output Arguments

`dataset`

The dataset in which you change or add an element.

Description

`dataset = Simulink.SimulationData.Dataset.setElement(index, element)` Change the element stored at the specified index, for an existing index. If index is one greater than the number of elements in the dataset, the element is added at the end of the dataset.

`dataset = Simulink.SimulationData.Dataset.setElement(index, element, name)` changes the element stored at the specified index and gives it the name that you specify with the `name` argument. You can use `name` to identify an element that does not have a name. If an object that already has a name (for example, a `Simulink.SimulationData.Signal` object to which you assigned a name), the element uses that name unless you specify a different name by using with the `name` argument.

Example

```
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
```

See Also

`Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.ModelDataLogs`

How To

- “Specify the Signal Logging Data Format”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”

Simulink.SimulationData.DataStoreMemory

Purpose

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`	Location of Data Store Memory block for the logged data store
`DSMWriterBlockPaths`	Location of Data Store Write blocks that write to the data store
`DSMWriters`	Data Store Write blocks for each signal value
`Name`	Name of the data store dataset
`Scope`	Scope of the data store: `'local'` or `'global'`
`Values`	Time and data that were logged

Properties

`BlockPath`

Description

Location of Data Store Memory block for the logged data store.

Data Type

string

Access

`RW`

`DSMWriterBlockPaths`

Description

Location of blocks that write to the data store. Each element of the array contains the full block path of one writer block.

Data Type

Vector of `Simulink.SimulationData.BlockPath` objects

Access

`RO`

`DSMWriters`

Description

The number of writes in the data store.

The `nth` element of `DSMWriters` contains the index of the element in `DSMWriterBLockPaths` that contains the block path of the writer that performed the `nth` write to `Values`.

Data Type

Integer vector

Access

`RO`

`Name`

Description

Name of the data store dataset

Data Type

string

Access

`RO`

`Scope`

Description

Scope of the data store: `'local'` or `'global'`

Data Type

string

Access

`RW`

`Values`

Description

Time and data that were logged

Data Type

MATLAB `timeseries`

Access

`RW`

See Also

`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.BlockPath` | Data Store Memory | Data Store Write

How To

- "Log Data Stores"

Simulink.SimulationData.LoggingInfo

Purpose

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`	Signal logging mode.
`NameMode`	Source of signal logging name.
`LoggingName`	Custom signal logging name.
`DecimateData`	Use interval for logging data points.
`Decimation`	Decimation value (`M`): Simulink logs every `M`th data point.
`LimitDataPoints`	Limit number of data points to log.
`MaxPoints`	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`	Create a set of signal logging override settings for a signal.

Properties

`DataLogging`

Description

Signal logging mode.

Indicates whether logging is enabled for this signal.

Data Type

logical value — `{true} | false`

Access

`RW`

`NameMode`

Description

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'`).

Data Type

logical value — `true | {false}`

Access

`RW`

`LoggingName`

Description

Custom signal logging name

The custom signal logging name to use for this signal, if the `NameMode` property is `true`.

Data Type

string

Access

`RW`

`DecimateData`

Description

Log a subset of sample points, selecting data points at a specified interval.

Data Type

logical value — `true | {false}`

Access

`RW`

`Decimation`

Description

Decimation value (`M`). If the `DecimateData` property is `true`, then Simulink logs every `M`th data point.

Data Type

nonzero integer

Access

`RW`

`LimitDataPoints`

Description

Limit the number of data points to log.

Data Type

logical value — `true | {false}`

Access

`RW`

`MaxPoints`

Description

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.

Data Type

nonzero integer

Access

`RW`

Methods

`LoggingInfo`

Purpose

Create a `Simulink.SimulationData.LoggingInfo` object.

Syntax

```
logging_info_object = Simulink.SimulationData.LoggingInfo()
logging_info_object = Simulink.SimulationData.LoggingInfo(object)
```

Input Arguments

`object`

A signal logging override settings object whose property values the constructor uses for the new `Simulink.SimlationData.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

Output Arguments

`logging_info_object`

A `Simulink.SimulationData.LoggingInfo` object.

Description

`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.

Examples

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`.

```
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.SimulationData.ModelLoggingInfo` | `Simulink.SimulationData.SignalLoggingInfo` | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.ModelDataLogs`

How To

- “Command-Line Interface for Overriding Signal Logging Settings”
- “Specify the Signal Logging Data Format”
- “Export Signal Data Using Signal Logging”

- “Log Data Stores”

Simulink.SimulationData.ModelLoggingInfo

Purpose

Signal logging override settings for a model

Description

This class is a collection of `Simulink.SimulationData.SignalLoggingInfo` objects that specify all signal logging override settings for a model.

Use methods and properties of this class to:

- Turn off logging for a signal or a Model block.
- Change logging settings for any signals that are already marked for logging within a model.

You can control whether a top-level model and referenced models use override signal logging settings or use the signal logging settings specified by the model. Use the `LoggingMode` and `LogAsSpecifiedByModels` properties to control which logging settings to apply.

Logging Mode for Models	Property Settings
For top-level model and all referenced models, use logging settings specified in the model.	Set `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`. In the `LogAsSpecifiedByModels` cell array, include the models that you do not want to use the override signal logging settings.

For more information and examples, see "Command-Line Interface for Overriding Signal Logging Settings".

Property Summary

Name	Description
`LoggingMode`	Signal logging override status
`LogAsSpecifiedByModels`	Source of signal logging settings for the top-level model or a top-level Model block
`Signals`	All signals that have signal override settings

Method Summary

Name	Description
`findSignal`	Find signals within the `Signals` vector, using block path and output port index.
`verifySignalAndModelPaths`	Verify signal and model paths for the model signal logging override object.
`getLogAsSpecifiedInModel`	Determine whether the model logs signals as specified in the model or uses override settings.
`setLogAsSpecifiedInModel`	Set the logging mode for the top-level model or a top-level Model block.
`createFromModel`	Create and populate a model signal logging override object with all logged signals in the model reference hierarchy.
`ModelLoggingInfo`	Set signals to log or override logging settings.

Properties

`LoggingMode`

Description

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.

Data Type

logical value — `true` or `false`

Access

`RW`

`LogAsSpecifiedByModels`

Description

When `LoggingMode` is set to `'OverrideSignals'`, this cell array specifies whether the top-level model or a top-level Model block logs all signals based on the signal logging settings defined in the model.

- For the top-level model and top-level Model blocks that the cell array includes, Simulink ignores the `Signals` property overrides.
- For a model or Model block that the cell array does *not* include, Simulink uses the `Signals` property to determine which signals to log.

When `LoggingMode` is set to `'LogAllAsSpecifiedInModel'`, Simulink ignores the `LogAsSpecifiedByModels` property.

Use the `getLogAsSpecifiedInModel` method to determine whether the top-level model or top-level Model block logs signals as specified in the model (default logging), and use `setLogAsSpecifiedInModel` to turn default logging on and off.

Data Type

cell array — For the top-level model, specify the model name. For Model blocks, specify the block path.

Access

`RW`

`Signals`

Description

Vector of `Simulink.SimulationData.SignalLoggingInfo` objects for all signals with signal logging override settings.

Data Type

vector of `Simulink.SimulationData.SignalLoggingInfo` objects

Access

`RW`

Methods

`createFromModel`

Purpose

Create a `Simulink.SimulationData.ModelLoggingInfo` object for a top-level model, with override settings for each logged signal in the model.

Syntax

```
model_logging_info_object = ...
Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
model, options)
```

Input Arguments

`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.SimlationData.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.

Output Arguments

`model_logging_override_object`

`Simulink.SimulationData.ModelLoggingInfo` object for the top-level model.

Description

`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

Examples

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);
```

The following example explicitly specifies the kinds of systems from which to include signals, rather than use the default settings for each kind of system. This example specifies to include signals from all model reference variants (instead of using the default of including only active variant).

The `sldemo_mdlref_variants` model has two variants: `sldemo_mrv_nonlinear_controller` and `sldemo_controller`. In this example, in each variant, you configure a signal for signal logging, and then create a `Simulink.SimulationData.ModelLoggingInfo` object. The resulting object includes, in the `Signals` property, two signals (one from each variant).

```
sldemo_mrv_nonlinear_controller;
sldemo_mrv_second_order_controller;
ph = get_param('sldemo_mrv_nonlinear_controller/Add','PortHandles');
set_param(ph.Outport(1),'DataLogging','on');
```

```
ph1 = get_param('sldemo_mrv_second_order_controller/Add','PortHandles');
set_param(ph1.Outport(1),'DataLogging','on');
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
'sldemo_mdlref_variants', 'Variants', 'AllVariants')
```

Results:

```
V_NONLINEAR_CONTROLLER =

Simulink.Variant
    Condition: 'CTRL==1'


V_SECOND_ORDER_CONTROLLER =

Simulink.Variant
    Condition: 'CTRL==2'


CTRL =

     1

mi =

  Simulink.SimulationData.ModelLoggingInfo
  Package: Simulink.SimulationData

  Properties:
                     Model: 'sldemo_mdlref_variants'
               LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
                   Signals: [1x2 Simulink.SimulationData.SignalLoggingInfo]

   Methods
```

`ModelLoggingInfo`

Purpose

Specify signals to log or override logging settings.

Syntax

```
model_logging_override_object = ....
Simulink.SimulationData.ModelLoggingInfo(model)
```

Input Arguments

`model`

Name of the top-level model for which to create a `Simulink.SimulationData.ModelLoggingInfo` object

Output Arguments

`model_logging_override_object`

`Simulink.SimulationData.ModelLoggingInfo` object created for the specified top-level model.

Description

`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.

Examples

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`

Purpose

Find signals within the `Signals` vector, using a block path and optionally an output port index.

Syntax

```
signal_indices = ...
      model_logging_override_object.findSignal(block_path)
signal_indices = ...
      model_logging_override_object.findSignal(...
      block_path, port_index)
```

Input Arguments

`block_path`

Source block to search. The `block_path` must be one of the following:

- String
- Cell array of strings
- `Simulink.BlockPath` object

`port_index`

Index of the output port to search. Specify a scalar greater than, or equal to, `1`.

Output Arguments

`signal_indices`

Vector of numeric indices into the signals vector of the `Simulink.SimulationData.ModelLoggingInfo` object.

Description

`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 string 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.

Examples

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)
```

The output is:

```
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({'sldemo_mdlref_bus/CounterA', ...
'sldemo_mdlref_counter_bus/Bus Creator'})
```

For an example that uses the `findSignal` method with a Stateflow chart, see "Override Logging Properties with the Command-Line API" in the Stateflow documentation.

`getLogAsSpecifiedInModel`

Purpose

Determine whether the model logs as specified in the model or uses override settings.

Syntax

```
logging_mode = ...
```

```
getLogAsSpecifiedInModel(model_logging_override_object, path)
```

Input Arguments

`model_logging_override_object`

A `Simulink.SimulationData.ModelLoggingInfo` object.

`path`

The `path` is a string that specifies one of the following:

- Name of the top-level model
- Block path of a Model block in the top-level model

Output Arguments

`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.

Description

`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.

Examples

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`

Purpose

Set logging mode for top-level model or top-level Model block

Syntax

`setLogAsSpecifiedInModel(override_object, path)`

Input Arguments

`override_object`

`Simulink.SimulationData.ModelLoggingInfo` object.

`path`

String 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.

Description

`setLogAsSpecifiedInModel(override_object, path, value)` sets the LoggingMode property for a top-level model or a Model block in the top-level model.

Example

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';
setLogAsSpecifiedInModel(mi, 'sldemo_mdlref_bus', true);
set_param('sldemo_mdlref_bus', 'DataLoggingOverride', mi);
```

---

verifySignalAndModelPaths

Purpose

Verify paths in `Simulink.SimulationData.ModelLoggingInfo` object.

Syntax

```
verified_object = verifySignalAndModelPaths...
     (model_logging_override_object, action)
```

Input Arguments

`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.

Output Arguments

`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.

Description

`verified_object = verifySignalAndModelPaths(model_logging_override_object, action)`

For a `Simulink.SimulationData.ModelLoggingInfo` object, verify that:

- All strings 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.

Example

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.SimulationData.LoggingInfo` | `Simulink.SimulationData.SignalLoggingInfo` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.ModelDataLogs`

How To

- "Command-Line Interface for Overriding Signal Logging Settings"
- "Specify the Signal Logging Data Format"
- "Export Signal Data Using Signal Logging"
- "Log Data Stores"

Simulink.SimulationData.SignalLoggingInfo

Purpose

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`	`Simulink.BlockPath` of source block of a signal to log.
`OutputPortIndex`	Index of an ouput port to log.
`LoggingInfo`	`Simulink.SimulationData.LoggingInfo` object containing all logging override settings for a signal.

Method Summary

Name	Description
`SignalLoggingInfo`	Create a signal logging override object for a signal.

Properties

`BlockPath`

Description

`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'})
```

Data Type

`Simulink.BlockPath`

Access

RW

`OuputPortIndex`

Description

Index of the output port to log. The index is a 1-based numeric value.

Data Type

nonzero integer

Access

RW

`LoggingInfo`

Description

`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.

Data Type

cell array

Access

RW

Methods

`SignalLoggingInfo`

Purpose

Construct a `Simulink.SimulationData.SignalLoggingInfo` object.

Syntax

```
signal_logging_info_object = ...
   Simulink.SimulationData.SignalLoggingInfo()
signal_loggingInfo_object = ...
   Simulink.SimulationData.SignalLoggingInfo(path)
```

```
signalLoggingInfo_object = ...
   Simulink.SimulationData.SignalLoggingInfo(path,index)
```

Input Arguments

`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.

Output Arguments

`signal_logging_object`

`Simulink.SimulationData.SignalLoggingInfo` object that represents the override settings of a signal.

Description

`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.

Examples

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.SimulationData.ModelLoggingInfo` | `Simulink.SimulationData.LoggingInfo` | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.ModelDataLogs`

How To

- “Command-Line Interface for Overriding Signal Logging Settings”
- “Specify the Signal Logging Data Format”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”

Simulink.SimulationData.Signal

Purpose

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`	Block path for the source block for the signal
`Name`	Name of signal element to use for name-based access
`PortIndex`	Numeric index of port that was logged
`PortType`	Type of port that was logged: for signal logging, the port type is `'outport'`
`Values`	Time and data that were logged

Properties

`BlockPath`

Description

Block path for the source block for the signal

Data Type

`Simulink.SimulationData.BlockPath`

Access

`RW`

`Name`

Description

Name of signal element to use for name-based access

Data Type
string

Access
`RW`

`PortIndex`

Description
Numeric index of port that was logged

Data Type
string

Access
`RW`

`PortType`

Description
Type of port that was logged: for signal logging, the port type is `'outport'`

Data Type
string

Access
`RW`

`Values`

Description
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.”

Data Type
A single MATLAB `timeseries` object or a MATLAB structure of `timeseries` objects (for bus signals)

Access

`RW`

See Also

`Simulink.BlockPath` | `Simulink.SimulationData.Dataset` | `timeseries`

How To

- “Access Signal Logging Data”
- “Import MATLAB timeseries Data”
- “Import Structures of timeseries Objects for Buses”

Purpose

Access object values of simulation results

Description

The `SimulationOutput` class contains all simulation outputs, including workspace variables.

Use `Simulink.SimulationOutput.who` and either `Simulink.SimulationOutput.get` or `Simulink.SimulationOutput.find` methods to access the output variable names and their respective values.

Methods

`find`	Access and display values of simulation results
`get`	Access and display values of simulation results
`who`	Access and display output variable names of simulation

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Simulate the `vdp` model and place the results of the `Simulink.SimulationOutput` object in `simOut`.

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...
                    'SaveState','on','StateSaveName','xoutNew',...
                    'SaveOutput','on','OutputSaveName','youtNew');
```

Store the variable names of the outputs in `simOutVars`, using the `who` method.

```
simOutVars = simOut.who
```

Simulink returns and displays:

```
simOutVars =
  'xoutNew'
  'youtNew'
```

Get the values of the variable *youtNew*.

```
yout = simOut.get('youtNew')
```

Simulink returns and displays the values.

See Also

`Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Dataset`

How To

- “Specify the Signal Logging Data Format”
- “Export Simulation Data”

Superclasses

Trigger

Purpose

Describe aperiodic trigger

Description

The `Simulink.SoftwareTarget.AperiodicTrigger` class describes an aperiodic trigger object. This object inherits its properties and methods from the `Simulink.SoftwareTarget.Trigger` class.

Properties

TargetObject

Target specific custom properties for the selected `EventHandlerType` (see the table in `EventHandlerType`).

Data Type: Object

Access: Read only

EventHandlerType

Describes the aperiodic trigger source for the trigger. Valid values depend on the value of the system target file for the active configuration set:

System Target File	EventHandlerType	TargetObject Class
`ert.tlc`, `grt.tlc`, `idelink_ert.tlc`, `idelink_grt.tlc`	Event (Windows)	`Simulink.SoftwareTarget.WindowsEventHandler`
`Ert.tlc`, `grt.tlc`, `idelink_ert.tlc`, `idelink_grt.tlc`	POSIX® signal (Linux®/VxWorks® 6.x)	`Simulink.SoftwareTarget.PosixSignalHandler`
`xpctarget.tlc`, `xpctargetert.tlc`	xPC interrupt	`xpcHardwareInterrupt`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Add an aperiodic trigger to the `sldemo_concurrent_execution model`.

```
% Load example model
sldemo_concurrent_execution
% Access its configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
% Access task configuration data
taskConfig = configSet.concurrentExecutionComponents;
% Add an aperiodic trigger
myTrigger = taskConfig.addAperiodicTrigger('MyTrigger')
% Set the name of the newly added aperiodic trigger
myTrigger.Name = 'MyTrigger';
```

Alternatives

“Customize Concurrent Execution Settings”

See Also

`Simulink.SoftwareTarget.Trigger` | `Simulink.SoftwareTarget.PeriodicTrigger`

How To

- “Customize Concurrent Execution Settings”

Simulink.SoftwareTarget.PeriodicTrigger

Superclasses

Trigger

Purpose

Describe periodic trigger

Description

The `Simulink.SoftwareTarget.PeriodicTrigger` class describes a periodic trigger object. This object inherits its properties and methods from the `Simulink.SoftwareTarget.Trigger` class.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Add an aperiodic trigger for the `sldemo_concurrent_execution model`.

```
% Load example model
sldemo_concurrent_execution
% Access its configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
% Access task configuration data
taskConfig = configSet.concurrentExecutionComponents;
% Access and change the name of the periodic trigger
taskConfig.Triggers(1).Name = 'MyTrigger';
```

Alternatives

"Customize Concurrent Execution Settings"

See Also

`Simulink.SoftwareTarget.Trigger` |
`Simulink.SoftwareTarget.AperiodicTrigger`

How To

- "Customize Concurrent Execution Settings"

Simulink.SoftwareTarget.PosixSignalHandler

Superclasses

Purpose

Describe aperiodic trigger for POSIX targets

Description

The `Simulink.SoftwareTarget.PosixSignalHandler` class describes an aperiodic trigger for one or more POSIX signals.

Construction

Create a `Simulink.SoftwareTarget.PosixSignalHandler` object by setting the EventHandlerType type of an aperiodic trigger object to `Posix Signal (Linux/VxWorks 6.x)`. This object is created as a side effect and stored in the `TargetObject` property of the aperiodic trigger object.

Properties

SignalNumber

Signal number in the range from 2 to *SIGRTMAX*-*SIGRTMIN*-1, where *SIGRTMAX* is the maximum value and *SIGRTMIN* is the minimum value of the signals supported by the target Linux or VxWorks operating system.

Data Type: Integer

Access: Read write

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `Simulink.SoftwareTarget.PosixSignalHandler` object.

```
% Open demo model
sldemo_concurrent_execution
% Access configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
% Access concurrent execution
taskConfig = configSet.concurrentExecutionComponents;
% Add an aperiodic trigger
trigger = taskConfig.addAperiodicTrigger('MyTrigger');
% Specify custom target
```

```
trigger.EventHandlerType = 'Posix Signal (Linux/VxWorks 6.x)';
% Access the custom target data
trigger.TargetObject
```

Alternatives

"Customize Concurrent Execution Settings"

See Also

`Simulink.SoftwareTarget.TaskConfiguration` |
`Simulink.SoftwareTarget.WindowsEventHandler` |
`Simulink.SoftwareTarget.Trigger.addTask`

How To

- "Customize Concurrent Execution Settings"

Simulink.SoftwareTarget.TaskConfiguration

Purpose

Configure model for concurrent execution

Description

The `Simulink.SoftwareTarget.TaskConfiguration` object contains the task configuration information for the concurrent execution of a model.

Construction

Create `Simulink.SoftwareTarget.TaskConfiguration` object by configuring a model for concurrent execution.

Properties

Name

Name of `Simulink.SoftwareTarget.TaskConfiguration` object.

Data Type: String

Access: Read/write

Triggers

Vector of periodic and aperiodic triggers, each of which describes trigger conditions for a set of tasks.

Data Type: Vector

Access: Read only

TargetCustomizationObject

Set of valid triggers for a deployment target.

Data Type: Object. For example, `Simulink.SoftwareTarget.GRTCustomization`.

Access: Read only

BlockToTaskMapping

Specification of block to task mapping.

Data Type: Object

Access: Read only

Methods

addAperiodicTrigger	Add aperiodic trigger to task configuration
deleteTrigger	Delete aperiodic trigger from task configuration
findTrigger	Return trigger object
map	Map block to object
unmap	Remove mapping from block to specified object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `Simulink.SoftwareTarget.TaskConfiguration` object.

```
% Create a new model
new_system('MyModel');
% Open the new model
open_system('MyModel');
configSet = Simulink.SoftwareTarget.concurrentExecution('MyModel',...
            'AddConfigurationForConcurrentExecution');
% Access the task configuration object
taskConfig = configSet.concurrentExecutionComponents;
```

Alternatives

“Customize Concurrent Execution Settings”

See Also

`Simulink.SoftwareTarget.TaskConfiguration.addAperiodicTrigger | Simulink.SoftwareTarget.TaskConfiguration.deleteTrigger | Simulink.SoftwareTarget.TaskConfiguration.map | Simulink.SoftwareTarget.TaskConfiguration.unmap`

How To

- “Customize Concurrent Execution Settings”

Superclasses

Purpose

Describe task that models unit of concurrent execution

Description

The `Simulink.SoftwareTarget.Task` class describes a task that models a unit of concurrent execution.

Properties

Color

Color of the task icon.

Data Type: Vector of three doubles for an RGB triplet (`[R,G,B]`)

Access: Read only

Name

Task name.

Data Type: string

Access: Read/write

Period

Period of task. This value must be a positive real number or a positive ratio.

Data Type: real

Access: Read/write

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `Simulink.SoftwareTarget.Task` object for the `sldemo_concurrent_execution` model.

```
% Load example model
sldemo_concurrent_execution
% Access its configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
% Access task configuration data
```

```
taskConfig = configSet.concurrentExecutionComponents;
% Add a new task object
taskObject = taskConfig.Triggers(1).addTask('MyTask');
% Set the color of the task object
taskObject.Color = [1 0 0];
% Delete the task object
taskConfig.Triggers.deleteTask(taskObject);
```

Alternatives

"Customize Concurrent Execution Settings"

See Also

`Simulink.SoftwareTarget.TaskConfiguration`

How To

- "Customize Concurrent Execution Settings"

Simulink.SoftwareTarget.Trigger

Purpose

Base class for PeriodicTrigger and AperiodicTrigger

Description

`Simulink.SoftwareTarget.Trigger` is the base class for the `Simulink.SoftwareTarget.PeriodicTrigger` and `Simulink.SoftwareTarget.AperiodicTrigger` objects.

Construction

This is an abstract class. See `Simulink.SoftwareTarget.AperiodicTrigger` and `Simulink.SoftwareTarget.PeriodicTrigger` for concrete classes.

Properties

Name

Name of trigger.

Data Type: String

Access: Read/write

TriggerType

Type of trigger.

Data Type: Enumeration. Can be one of:

- `'Periodic'`
- `'Aperiodic'`

Access: Read only

Tasks

Vector of tasks defined for this trigger.

Data Type: Vector

Access: Read only

Color

Color for rendering the trigger.

Data Type: RGB vector of doubles ([R,G,B])

Access: Read/write

Methods

addTask	Add task to specified trigger
deleteTask	Delete specified task from trigger
findTask	Return task object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Alternatives

"Customize Concurrent Execution Settings"

See Also

`Simulink.SoftwareTarget.Trigger.addTask` | `Simulink.SoftwareTarget.AperiodicTrigger` | `Simulink.SoftwareTarget.PeriodicTrigger` | `Simulink.SoftwareTarget.Trigger.deleteTask` | `Simulink.SoftwareTarget.Trigger.findTask`

How To

- "Customize Concurrent Execution Settings"

Simulink.SoftwareTarget.WindowsEventHandler

Superclasses

Purpose

Describe aperiodic trigger for Windows targets

Description

The `Simulink.SoftwareTarget.WindowsEventHandler` class describes an aperiodic trigger for one or more signals on Windows targets.

Construction

Create a `Simulink.SoftwareTarget.WindowsEventHandler` object by setting the EventHandlerType type of an aperiodic trigger object to `Event (Windows)`. This object is created as a side effect and stored in the `TargetObject` property of the aperiodic trigger object.

Properties

EventName

Name associated with the aperiodic trigger. Specify a valid Windows path name component, excluding backslash (`` `\' ``), limited to `MAX_PATH` characters.

Data Type: String

Access: Read/write

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `Simulink.SoftwareTarget.WindowsEventHandler` object.

```
% Open demo model
sldemo_concurrent_execution
% Access configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
% Access concurrent execution
taskConfig = configSet.concurrentExecutionComponents;
% Add an aperiodic trigger
trigger = taskConfig.addAperiodicTrigger('MyTrigger');
trigger.EventHandlerType = 'Event (Windows)';
% Specify custom target
% Access the custom target data
```

```
trigger.TargetObject
```

See Also

`Simulink.SoftwareTarget.TaskConfiguration` |
`Simulink.SoftwareTarget.PosixSignalHandler` |
`Simulink.SoftwareTarget.Trigger.addTask`

Purpose

Add aperiodic trigger to task configuration

Syntax

`aperiodic_trigger = addAperiodicTrigger(trigger_name)`

Description

`aperiodic_trigger = addAperiodicTrigger(trigger_name)` adds the aperiodic trigger, `aperiodic_trigger`, to the task configuration.

Input Arguments

trigger_name

Name of trigger object.

Default: none

Output Arguments

aperiodic_trigger

Aperiodic trigger object. If `trigger_name` is the name of another trigger object within the parent task configuration, the software makes `trigger_name` unique by appending the smallest integer, for example, `aperiodic_trigger1`.

Examples

Add an aperiodic trigger to the `sldemo_concurrent_execution model`.

```
% Load example model
sldemo_concurrent_execution
% Access its configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
% Access task configuration data
taskConfig = configSet.concurrentExecutionComponents;
% Add an aperiodic trigger
myTrigger = taskConfig.addAperiodicTrigger('MyTrigger')
% Set the name of the newly added aperiodic trigger
myTrigger.Name = 'MyTrigger';
```

Alternatives

"Customize Concurrent Execution Settings"

See Also

`Simulink.SoftwareTarget.TaskConfiguration` |
`Simulink.SoftwareTarget.TaskConfiguration.deleteTrigger`

| `Simulink.SoftwareTarget.TaskConfiguration.map` | `Simulink.SoftwareTarget.TaskConfiguration.unmap`

How To

- "Customize Concurrent Execution Settings"

Simulink.SoftwareTarget.TaskConfiguration.deleteTrigger

Purpose

Delete aperiodic trigger from task configuration

Syntax

```
deleteTrigger(trigger_name)
deleteTrigger(trigger_number)
```

Description

`deleteTrigger(trigger_name)` deletes the specified aperiodic trigger, `trigger_name`, from the task configuration. This method returns an error if `trigger_name` is the first periodic trigger in the task configuration, or if `trigger_name` does not exist in the task configuration.

`deleteTrigger(trigger_number)` deletes the specified aperiodic trigger, `trigger_number`, from the task configuration. This method returns an error if `trigger_number` is the first periodic trigger in the task configuration, or if `trigger_number` does not exist in the task configuration.

Input Arguments

trigger_name

Name of trigger object.

Default: none

trigger_number

Index number of trigger object.

Examples

Delete trigger by trigger number.

```
triggers = taskConfiguration.Triggers

for i=2:length(triggers)

  taskConfiguration.deleteTrigger(triggers(i))

end
```

See Also

`Simulink.SoftwareTarget.TaskConfiguration` | `Simulink.SoftwareTarget.TaskConfiguration.addAperiodicTrigger` | `Simulink.SoftwareTarget.TaskConfiguration.map` | `Simulink.SoftwareTarget.TaskConfiguration.unmap`

Simulink.SoftwareTarget.TaskConfiguration.findTrigger

Purpose

Return trigger object

Syntax

`aperiodic_trigger = findTrigger(trigger_name)`

Description

`aperiodic_trigger = findTrigger(trigger_name)` returns the specified trigger object.

Input Arguments

trigger_name

Name of trigger object.

Default: none

Output Arguments

aperiodic_trigger

Aperiodic trigger object.

See Also

`Simulink.SoftwareTarget.TaskConfiguration` | `Simulink.SoftwareTarget.TaskConfiguration.deleteTrigger` | `Simulink.SoftwareTarget.TaskConfiguration.map` | `Simulink.SoftwareTarget.TaskConfiguration.unmap`

Simulink.SoftwareTarget.TaskConfiguration.map

Purpose

Map block to object

Syntax

```
map(block_handle,task_object)
map(block_handle,aperiodic_object)
map(block_handle,trigger_number, task_name)
map(block_handle,trigger_name, task_name)
map(block_handle,trigger_name)
map(block_handle,trigger_number)
```

Description

`map(block_handle,task_object)` maps the given block to the task object.

`map(block_handle,aperiodic_object)` maps the given block to the aperiodic trigger object.

`map(block_handle,trigger_number, task_name)` maps the given block to the task object, as identified by the trigger number and task name.

`map(block_handle,trigger_name, task_name)` maps the given block to the task object, as identified by the trigger name and task name.

`map(block_handle,trigger_name)` maps the given block to the aperiodic trigger object, as identify by the trigger name, in the task configuration.

`map(block_handle,trigger_number)` maps the given block to the aperiodic trigger object, as identified by the trigger number, in the task configuration.

Input Arguments

block_handle

Handle (double) of block to map.

task_object

Task object to map to `block_handle`.

aperiodic_object

Aperiodic task object to map to `block_handle`.

trigger_number

Identifier of trigger object.

task_name

Name of task to associate with the trigger to identify the task object.

trigger_name

Name of trigger object.

Default: none

Examples

Create a new task and map a block to the new task.

```
% Open demo model
sldemo_concurrent_execution
% Get the configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
taskConfig = configSet.concurrentExecutionComponents;
% Get the block of interest
block = get_param('sldemo_concurrent_execution/Plant', 'Handle');
% Create a new task
newTask = taskConfig.Triggers(1).addTask('NewTask');
newTask.Period = '0.1';
% Get the task to which plant was mapped to
oldTask = taskConfig.Triggers(1).findTask('Plant');
% Map the plant block to the new task
taskConfig.unmap(block, oldTask);
taskConfig.map(block, newTask);
```

Alternatives

“Customize Concurrent Execution Settings”

See Also

`Simulink.SoftwareTarget.TaskConfiguration` | `Simulink.SoftwareTarget.TaskConfiguration.addAperiodicTriger` | `Simulink.SoftwareTarget.TaskConfiguration.deleteTrigger` | `Simulink.SoftwareTarget.TaskConfiguration.unmap`

How To

- “Customize Concurrent Execution Settings”

Simulink.SoftwareTarget.TaskConfiguration.unmap

Purpose

Remove mapping from block to specified object

Syntax

```
unmap(block_handle,task_object)
unmap(block_handle,aperiodic_object)
unmap(block_handle,trigger_number, task_name)
unmap(block_handle,trigger_name, task_name)
unmap(block_handle,trigger_name)
unmap(block_handle,trigger_number)
```

Description

`unmap(block_handle,task_object)` removes the mapping from the given block to the task object.

`unmap(block_handle,aperiodic_object)` removes the mapping from the given block to the aperiodic trigger object.

`unmap(block_handle,trigger_number, task_name)` removes the mapping from the given block to the task object, as identified by the trigger number and task name.

`unmap(block_handle,trigger_name, task_name)` removes the mapping from the given block to the task object, as identified by the trigger name and task name.

`unmap(block_handle,trigger_name)` removes the mapping from the given block to the aperiodic trigger object, as identified by the trigger name, in the task configuration.

`unmap(block_handle,trigger_number)` removes the mapping from the given block to the aperiodic trigger object, as identified by the trigger number, in the task configuration.

Input Arguments

block_handle

Handle (double) of block from which to remove mapping.

task_object

Task object from which to remove mapping.

aperiodic_object

Aperiodic task object from which to remove mapping.

trigger_number

Identifier of trigger object.

task_name

Name of task to associate with the trigger to identify the task object.

trigger_name

Name of trigger object.

Default: none

Examples

Create a new task, map a block to the new task, and then unmap the block.

```
% Open demo model
sldemo_concurrent_execution
% Get the configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
taskConfig = configSet.concurrentExecutionComponents;
% Get the block of interest
block = get_param('sldemo_concurrent_execution/Plant', 'Handle');
% Create a new task
newTask = taskConfig.Triggers(1).addTask('NewTask');
newTask.Period = '0.1';
% Get the task to which plant was mapped to
oldTask = taskConfig.Triggers(1).findTask('Plant');
% Map the plant block to the new task
taskConfig.unmap(block, oldTask);
taskConfig.map(block, newTask);
```

See Also

`Simulink.SoftwareTarget.TaskConfiguration | Simulink.SoftwareTarget.TaskConfiguration.addAperiodicTrigger | Simulink.SoftwareTarget.TaskConfiguration.deleteTrigger | Simulink.SoftwareTarget.TaskConfiguration.map`

Simulink.SoftwareTarget.Trigger.addTask

Purpose

Add task to specified trigger

Syntax

`addTask(task_name)`

Description

`addTask(task_name)` adds a task to the trigger. If `task_name` already exists, the method renames the task to the next available incremental name, such as `task_name1`.

Input Arguments

task_name

Name of task to associate with the trigger to identify the task object.

Output Arguments

task_object

Task object, with name `task_name`, associated with the trigger.

Examples

Create a new task.

```
% Open demo model
sldemo_concurrent_execution
% Get the configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
taskConfig = configSet.concurrentExecutionComponents;
% Get the block of interest
block = get_param('sldemo_concurrent_execution/Plant', 'Handle');
% Create a new task
newTask = taskConfig.Triggers(1).addTask('NewTask');
newTask.Period = '0.1';
% Get the task to which plant was mapped to
oldTask = taskConfig.Triggers(1).findTask('Plant');
% Map the plant block to the new task
taskConfig.unmap(block, oldTask);
taskConfig.map(block, newTask);
```

Alternatives

“Customize Concurrent Execution Settings”

See Also

`Simulink.SoftwareTarget.Trigger` | `Simulink.SoftwareTarget.Trigger.deleteTask` | `Simulink.SoftwareTarget.Trigger.findTask`

How To

- "Customize Concurrent Execution Settings"

Simulink.SoftwareTarget.Trigger.deleteTask

Purpose

Delete specified task from trigger

Syntax

```
deleteTask(task_name)
deleteTask(task_object)
```

Description

`deleteTask(task_name)` deletes a task, `task_name`, from the trigger. The method returns an error if the task does not exist in the trigger.

`deleteTask(task_object)` deletes a task object from the trigger.

Input Arguments

task_name

Name of task to associate with the trigger to identify the task object.

task_object

Task object to remove from the trigger.

Examples

Create a new task and then delete the task.

```
% Load example model
sldemo_concurrent_execution
% Access its configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
% Access task configuration data
taskConfig = configSet.concurrentExecutionComponents;
% Add a new task object
taskObject = taskConfig.Triggers(1).addTask('MyTask');
% Set the color of the task object
taskObject.Color = [1 0 0];
% Delete the task object
taskConfig.Triggers(1).deleteTask(taskObject);
```

See Also

`Simulink.SoftwareTarget.Trigger` |
`Simulink.SoftwareTarget.Trigger.addTask` |
`Simulink.SoftwareTarget.Trigger.findTask`

Simulink.SoftwareTarget.Trigger.findTask

Purpose

Return task object

Syntax

`task_object = findTask(task_name)`

Description

`task_object = findTask(task_name)` returns the specified task object. The method returns an error if it does not find a task with the specified name.

Input Arguments

task_name

Name of task to associate with the trigger to identify the task object.

Output Arguments

task_object

Task object found.

Examples

Create a new task and find the old task.

```
% Open demo model
sldemo_concurrent_execution
% Get the configuration set
configSet = getActiveConfigSet('sldemo_concurrent_execution');
taskConfig = configSet.concurrentExecutionComponents;
% Get the block of interest
block = get_param('sldemo_concurrent_execution/Plant', 'Handle');
% Create a new task
newTask = taskConfig.Triggers(1).addTask('NewTask');
newTask.Period = '0.1';
% Get the task to which plant was mapped to
oldTask = taskConfig.Triggers(1).findTask('Plant');
% Map the plant block to the new task
taskConfig.unmap(block, oldTask);
taskConfig.map(block, newTask);
% Delete task
taskConfig.Triggers(1).deleteTask('NewTask')
```

See Also

`Simulink.SoftwareTarget.Trigger` |
`Simulink.SoftwareTarget.Trigger.addTask` |
`Simulink.SoftwareTarget.Trigger.deleteTask`

Purpose

Container for subsystem's signal data logs

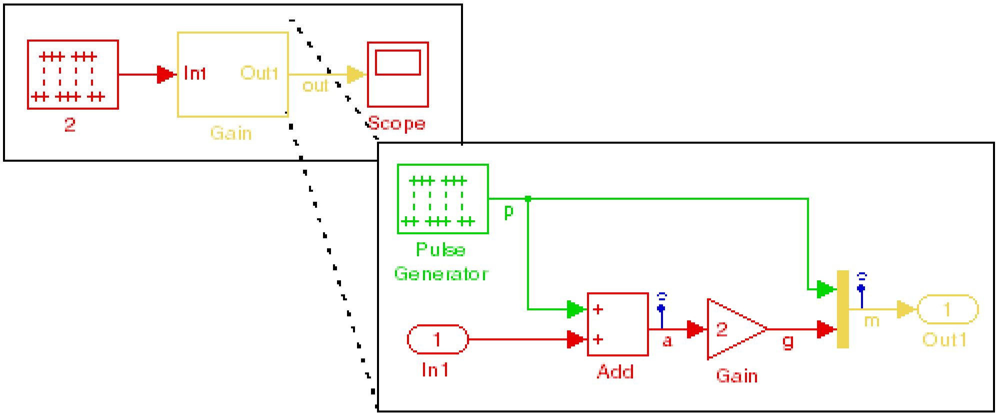
Description

Note The `Simulink.SubsysDataLogs` class is used in conjunction with the `ModelDataLogs` logging data format. The `ModelDataLogs` format is supported for backwards compatibility. The `ModelDataLogs` format will be removed in a future release.

For new models, use the `Dataset` logging format.

Simulink software creates instances of this class to contain logs for signals belonging to a subsystem (see "Export Signal Data Using Signal Logging"). 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.

Consider, for example, the following model.



After simulation of this model, the MATLAB workspace contains a `Simulink.ModelDataLogs` object, named `logsout`, that contains a

`Simulink.SubsysDataLogs` object, named `Gain`, that contains the log data for signals `a` and `m` in the subsystem named Gain.

```
>> logsout.Gain

ans =

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

"Signal Logging", `Simulink.ModelDataLogs`, `Simulink.ScopeDataLogs`, `Simulink.Timeseries`, `Simulink.TsArray`, `Simulink.SimulationData.Dataset`, `who`, `whos`, `unpack`

Purpose

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.

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.
`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.

Name	Access	Description
`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`

Purpose

Store data for any signal except mux or bus signal

Description

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.
`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.ScopeDataLogs`, `Simulink.TsArray`, `who`, `whos`, `unpack`

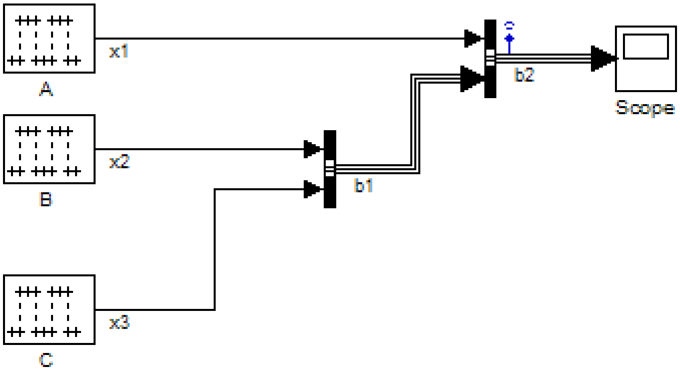
Simulink.TsArray

Purpose Store data for mux or bus signal

Description Simulink software creates instances of this class to contain the data that it logs for a mux or bus signal. Other types of signals are stored in a `Simulink.Timeseries`. See “Export Signal Data Using Signal Logging” for more information.

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, consider the following model.



This model specifies that Simulink software should log the values of the signal `b2` during simulation. After simulation of this model, the MATLAB workspace contains a `Simulink.ModelDataLogs` object, named `logsout`, that contains a `Simulink.TsArray` object, named `b2`, that contains the logs for the elements of `b2`, i.e., for the elementary signal `x1` and the bus signal `b1`. entering the fully qualified name of the `Simulink.TsArray` object, i.e., `logsout.b2`, at the MATLAB command line reveals the structure of the signal log for this model.

```
>> logsout.b2
Simulink.TsArray (untitled/Bus Creator1):
  Name                   elements   Simulink Class

  x1                        1       Timeseries
  b1                        2       TsArray
```

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 could enter the following at the MATLAB command line:

```
>> data = logsout.b2.x1.Data;
```

or

```
>> logsout.unpack('all');
data = x1.Data;
```

See Also

"Export Signal Data Using Signal Logging",`Simulink.ModelDataLogs`, `Simulink.SubsysDataLogs`, `Simulink.ScopeDataLogs`, `Simulink.Timeseries`, `Simulink.SimulationData.Dataset`, `who`, `whos`, `unpack`

Simulink.Variant

Purpose

Specify a model reference variant and its execution environment

Description

A `Simulink.Variant` object specifies a variant object, which is defined using a Boolean expression called a variant condition. For more information, see "How Variant Controls Work". A Model Variants block and a Variant Subsystem block use variant objects to associate an execution environment with each of its variants.

The variant condition references MATLAB variables and/or Simulink parameter objects in the base workspace called variant control variables. The variant condition must evaluate to `true` or `false` when the model is compiled.

For a given Model Variants block or Variant Subsystem block, you must define the variant objects so that only one evaluates to `true`. The variant object, whose variant condition evaluates to `true`, is the active variant for that variant block.

Construction

`variant=Simulink.Variant(VariantCondition)` creates a variant object in the base workspace. The object name is *`variant`*, and the associated variant condition is *`VariantCondition`*. The object can have any unique legal MATLAB name.

The *`VariantCondition`* must be a Boolean expression that references at least one base workspace variable or parameter. A variant condition can include scalar variables, enumerated values, the operators `==`, `~=`, `&&`, `||`, `~`, as well as parentheses for grouping. If you specify the condition literally, surround it with single quotes.

You can also create a variant object using the Model explorer. Select the **Base Workspace**, choose **Add > `Simulink.Variant`**, and specify the desired name and variant condition in the Contents and Dialog panes. If you specify the condition literally, do *not* surround it with single quotes (unlike the MATLAB API, which requires single quotes).

Properties

`Condition`

Description

The Boolean expression associated with the variant object.

Access

`RW`

Examples

Create a variant object with a variant condition:

```
GU=Simulink.Variant('Fuel==1 && Emis==1')
```

See Also

- "Working with Variant Systems"
- "Variant Systems"

Simulink.WorkspaceVar

Purpose

Contains information about workspace variables and blocks that use them

---

Note `Simulink.WorkspaceVar` will be removed in a future release. Use `Simulink.VariableUsage` instead.

Description

A `Simulink.WorkspaceVar` object describes attributes of a workspace variable and lists the blocks that use the variable. The `Simulink.findVars` function returns one or more `Simulink.WorkspaceVar` objects that embody the results of searching for variables.

Tip

Only a `Simulink.WorkspaceVar` constructor can set any field value in a `Simulink.WorkspaceVar` object. The fields are otherwise read-only.

Fields

Name

The name of the variable described by the `Simulink.WorkspaceVar` object

Workspace

The name of the workspace in which the variable resides, for example:

`Workspace` value	Meaning
`'base workspace'`	The MATLAB base workspace
`'MyModel'`	The model workspace for the model `MyModel`.
`'MyModel/Mask1'`	The mask workspace for the masked block `Mask1` in the model `MyModel`.

WorkspaceType

The type of workspace in which the variable resides. The possible values are:

- `'base'`
- `'model'`
- `'mask'`

UsedByBlocks

A cell array of strings. Each string names a block that uses the variable. `Simulink.findVars` populates this field.

Construction

`var = Simulink.WorkspaceVar (VarName, WkspName)`, where both arguments are strings, returns a `Simulink.WorkspaceVar` object with `Name` *`VarName`* and `Workspace` *`WkspName`*. The inputs need not exist in the model. Simulink will provide a *`WorkspaceType`* automatically.

`vars = Simulink.WorkspaceVar (VarNames, WkspName)`, where *`VarNames`* is a cell array of strings, returns a vector of `Simulink.WorkspaceVar` objects, each with a specified name and `Workspace` = *`WkspName`*.

Methods

[*`VarsOut`*] = *`VarsIn1`*`.setdiff (`*`VarsIn2`*`)` — Calls `setdiff` to return the difference between *`VarsIn1`* and *`VarsIn2`*. The arguments and return value are vectors of `Simulink.WorkspaceVar` objects.

[*`VarsOut`*] = *`VarsIn1`*`.intersect (`*`VarsIn2`*`)` — Calls `intersect` to return the intersection between *`VarsIn1`* and *`VarsIn2`*. The arguments and return value are vectors of `Simulink.WorkspaceVar` objects.

Examples

Create a `Simulink.WorkspaceVar` object for the variable `'k'` in the base workspace.

```
var = Simulink.WorkspaceVar('k', 'base workspace');
```

Return a vector of `Simulink.WorkspaceVar` objects, one object for each variable returned by `who`.

```
[vars] = Simulink.WorkspaceVar (who,  WkspName)
```

Return a vector of `Simulink.WorkspaceVar` objects, one object for each variable returned by `whos`.

```
[vars] = Simulink.WorkspaceVar (who,  WkspName)
```

Create a vector of `Simulink.WorkspaceVar` objects that describes all the variables in a model workspace

```
hws = get_param('mymodel', 'ModelWorkspace');
vars=Simulink.WorkspaceVar(hws.whos, 'MyModel')
```

Create a vector of `Simulink.WorkspaceVar` objects that describes all the variables in a mask workspace

```
maskVars = get_param('mymodel/maskblock', 'MaskWSVariables');
vars = Simulink.WorkspaceVar(maskVars, 'mymodel/maskblock');
```

See Also

- `Simulink.findVars`
- `setdiff`
- `intersect`

Purpose

Get information about workspace variables and blocks that use them

Tip

Only a `Simulink.VariableUsage` constructor can set any field value in a `Simulink.VariableUsage` object. The fields are otherwise read-only.

Description

Create a `Simulink.VariableUsage` object to get the attributes of a workspace variable and determine the blocks that use the variable.

Construction

Create a `Simulink.VariableUsage` object as follows.

`var = Simulink.VariableUsage (VarName, WkspName)`, where both arguments are strings, returns a `Simulink.VariableUsage` object with `Name` *`VarName`* and `Workspace` *`WkspName`*. The inputs need not exist in the model. Simulink will provide a *`WorkspaceType`* automatically.

`vars = Simulink.VariableUsage (VarNames, WkspName)`, where *`VarNames`* is a cell array of strings, returns a vector of `Simulink.VariableUsage` objects, each with a specified name and `Workspace` = *`WkspName`*.

Properties

Name

The name of the variable described by the `Simulink.VariableUsage` object

Source

The name of the workspace in which the variable resides, 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`.

SourceType

The type of workspace in which the variable resides. The possible values are:

- `'base workspace'`
- `'`*`model`*`'`
- `'`*`mask`*`'`

Users

A cell array of strings. Each string names a block that uses the variable. `Simulink.findVars` populates this field.

Methods

[*`VarsOut`*] = *`VarsIn1`*`.setdiff (`*`VarsIn2`*`)` — Calls `setdiff` to return the difference between *`VarsIn1`* and *`VarsIn2`*. The arguments and return value are vectors of `Simulink.VariableUsage` objects.

[*`VarsOut`*] = *`VarsIn1`*`.intersect (`*`VarsIn2`*`)` — Calls `intersect` to return the intersection between *`VarsIn1`* and *`VarsIn2`*. The arguments and return value are vectors of `Simulink.VariableUsage` objects.

Examples

Create a `Simulink.VariableUsage` object for the variable `'k'` in the base workspace.

```
var = Simulink.VariableUsage('k', 'base workspace');
```

Return a vector of `Simulink.VariableUsage` objects, one object for each variable returned by `who`.

```
[vars] = Simulink.VariableUsage (who,  WkspName)
```

Return a vector of `Simulink.VariableUsage` objects, one object for each variable returned by `whos`.

```
[vars] = Simulink.VariableUsage (who,  WkspName)
```

Create a vector of `Simulink.VariableUsage` objects that describes all the variables in a model workspace

```
hws = get_param('mymodel', 'ModelWorkspace');
vars = Simulink.VariableUsage(hws.whos, 'MyModel')
```

Create a vector 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.findVars` | `setdiff` | `intersect`

Simulink.ModelManagement.Project.CurrentProject.loadProject

Purpose

(To be removed) Load an existing Simulink project and close any currently open project

---

Note `Simulink.ModelManagement.Project.CurrentProject.loadProject` will be removed in a future release. Use `slproject.loadProject` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.loadProject(pathtoprojectfolder)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.loadProject(pathtoprojectfolder)` loads the Simulink project in the `pathtoprojectfolder` specified.

Loading a project closes any currently open project. If the currently open project is configured to run shutdown shortcuts, the shutdown tasks run before loading the next project.

Input Argument

pathtoprojectfolder

The complete path to the folder that contains the `.SimulinkProject` folder.

Example

To load an existing Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `loadProject` method specifying the path to your project folder (that contains the .SimulinkProject folder), for example:

   ```
   p.loadProject('C:\work\airframe');
   ```

See Also `Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.isProjectLoa

Purpose

(To be removed) Return true when a Simulink project is loaded and false when no project is loaded

---

Note `Simulink.ModelManagement.Project.CurrentProject.isProjectLoaded` will be removed in a future release. Use `isLoaded` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
isLoaded = p.isProjectLoaded
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`isLoaded = p.isProjectLoaded` returns `1` or `0` depending on whether a Simulink project is loaded or not loaded, respectively.

Example

To determine if a Simulink project is loaded, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `isProjectLoaded` method.

   ```
   isLoaded = p.isProjectLoaded;
   ```

   `isLoaded = 0` indicates that no Simulink project is loaded

   `isLoaded = 1` indicates a Simulink project is loaded

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Get the name of the root folder for the currently open Simulink project

---

Note `getRootDirectory` will be removed in a future release. Access the `RootFolder` property of a project object instead. Use `simulinkproject` to get a project object.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.getRootDirectory
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.getRootDirectory` returns the name of the root folder for the currently open Simulink project.

This method is particularly useful when creating start up and shutdown scripts, or functions, for a project. Use this method to determine the project root folder so that sub-folders of the project can be added to the MATLAB path without using absolute paths.

Example

To get the name of the root folder for the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `getRootDirectory` method.

   ```
   p.getRootDirectory;
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Get the name of the root folder for the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.getRootFolder` will be removed in a future release. Access the `RootFolder` property of a project object instead. Use `simulinkproject` to get a project object.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.getRootFolder
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.getRootFolder` returns the name of the root folder for the currently open Simulink project.

This method is particularly useful when creating start up and shutdown scripts, or functions, for a project. Use this method to determine the project root folder so that sub-folders of the project can be added to the MATLAB path without using absolute paths.

Example

To get the name of the root folder for the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `getRootFolder` method.

   ```
   p.getRootFolder;
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Get the name of the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.getProjectName` will be removed in a future release. See "Simulink Projects" for links to project functions.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
name = p.getProjectName
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`name = p.getProjectName` returns the name of the currently open Simulink project.

Example

To get the name of the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `getProjectName` method.

   ```
   name = p.getProjectName;
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.getLabels

Purpose

(To be removed) Get a list of labels in a category in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.getLabels` will be removed in a future release. Instead, use `simulinkproject` and then access the `Labels` property of project files, or use `findLabel` to get labels by name.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
labels = p.getLabels(category)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`labels = p.getLabels(category)` returns an array of label names in a specified `category` in the currently open Simulink project.

Input Argument

category

The name of a category in the currently open Simulink project.

Examples

- To get a list of labels in category `Review` in the currently open Simulink project, use the following commands.

  1. Create an instance of the `CurrentProject` class.

     ```
     p = Simulink.ModelManagement.Project.CurrentProject();
     ```

  2. Use the `getLabels` method.

     ```
     labels = p.getLabels('Review');
     ```

  3. To view the cell array containing the category names, use

     ```
     labels{:}
     ```

- To display all the labels within the currently open Simulink project, run the following code.

```
p = Simulink.ModelManagement.Project.CurrentProject();
categories = p.getCategories;
for jj = 1:numel(categories)
    thisCategory = categories{jj};
    disp([thisCategory ' : ']);
    labels = p.getLabels(thisCategory);
    for kk = 1:numel(labels)
        thisLabel = labels(kk);
        disp([' - ' thisLabel.Name]);
    end
end
```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Get a list of files in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.getFilesInProject` will be removed in a future release. Instead, use `simulinkproject` and then access the `Files` property of the project object.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
projFiles = p.getFilesInProject
projFiles = p.getFilesInProject(includeFolders)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`projFiles = p.getFilesInProject` returns a cell array of strings containing the fully qualified filenames of all the files and the folders in the currently open Simulink project.

`projFiles = p.getFilesInProject(includeFolders)` specifies whether to include folders in the list of project files. `includeFolders` is an optional logical input specifying whether to include folders (`true`) or exclude folders (`false`). Default: `true`.

Example

To get the names of files in the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject()
   ```

2. Use the `getFilesInProject` method.

   ```
   projFiles = p.getFilesInProject;
   ```

3. To view the cell array containing the file names, use

   ```
   projFiles{:}
   ```

4 Use the `getFilesInProject` method and exclude folders.

```
projFiles = p.getFilesInProject(false);
```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.getCategor

Purpose

(To be removed) Get a list of the categories of file labels in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.getCategories` will be removed in a future release. Instead, use `simulinkproject` and then access the `Categories` property of the project object.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
categories = p.getCategories
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`categories = p.getCategories` returns a cell array of category names in the currently open Simulink project.

Example

To get a list of categories in the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `getCategories` method.

   ```
   categories = p.getCategories;
   ```

3. To view the cell array containing the category names, use

   ```
   categories{:}
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.getAtt

Purpose

(To be removed) Get a list of labels attached to a file in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.getAttachedLabel` will be removed in a future release. Instead, use `simulinkproject` to get a project object, access the `Files` property of the project, and get the `Labels` property of a file.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
labels = p.getAttachedLabels(filePath)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`labels = p.getAttachedLabels(filePath)` returns an array of label objects for a file specified using `filePath`.

Input Argument

filePath

The fully qualified path to the file within the root folder of the project.

Example

To get a list of labels in a file in the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Create a fully qualified path for the file.

   Note The use of the `getRootDirectory` method in this step is optional.

```
file = fullfile(p.getRootDirectory, 'work', 'myExample.c');
```

3. Use the `getAttachedLabels` method.

```
labels = p.getAttachedLabels(file);
```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.detach

Purpose

(To be removed) Remove a label from a file in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.detachLabelFromF` will be removed in a future release. Use `removeLabel` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.detachLabelFromFile(filePath,labelObj)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.detachLabelFromFile(filePath,labelObj)` removes a label from a file specified using `filePath` using the label object `labelObj` .

Input Argument

filePath

The fully qualified path to the file from which the label is to be removed.

labelObj

The label object used to remove the label from the file.

Example

To remove a label from a file labelled with review status `Approved`, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Create a fully qualified path for the file.

   Note The use of the `getRootDirectory` method in this step is optional.

```
file = fullfile(p.getRootDirectory, 'work', 'myExample.c');
```

3. Create a label object.

```
labelObj = Simulink.ModelManagement.Project.Label('Review','Approved');
```

4. Use the `detachLabelFromFile` method.

```
p.detachLabelFromFile(file,labelObj);
```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Create a label in a category in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.createLabel` will be removed in a future release. Use `createLabel` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
labelObj =
Simulink.ModelManagement.Project.Label(categoryName
   ,labelName);
p.createLabel(labelObj)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`labelObj = Simulink.ModelManagement.Project.Label(categoryName,labelName);` creates a label object named `labelObj` for the category `categoryName`.

`p.createLabel(labelObj)` creates the label in the category.

If the category for the new label does not exist, it is created with the label object.

Input Argument

categoryName

The name of the category in which the new label is to be added.

labelName

The name of the new label.

Examples

To create a new label `John` in category `Owner` in the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

```
p = Simulink.ModelManagement.Project.CurrentProject();
```

2 Create a label object.

```
labelObj= Simulink.ModelManagement.Project.Label('Owner','John');
```

3 Add the label to the category.

```
p.createLabel(labelObj);
```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Create a category in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.createCategory` will be removed in a future release. Use `createCategory` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.createCategory(categoryName)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.createCategory(categoryName)` creates a category named `categoryName` in the currently open Simulink project.

The new category is created immediately. If you are using a version control tool for your project, you must commit the resulting changes to the `.SimulinkProject` folder to source control.

Input Argument

categoryName

The name of the new category.

Examples

To create a new category `Owner` in the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `createCategory` method.

   ```
   p.createCategory('Owner');
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.close

Purpose

(To be removed) Close the currently open Simulink project

Note `Simulink.ModelManagement.Project.CurrentProject.close` will be removed in a future release. Use `close` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.close
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.close` closes the currently open Simulink project.

Example

To close a currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `close` method.

   ```
   p.close;
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.attach

Purpose

(To be removed) Add a label to a file in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.attachLabelToFil` will be removed in a future release. Use `addLabel` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.attachLabelToFile(filePath,labelObj)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.attachLabelToFile(filePath,labelObj)` adds a label to a file specified using `filePath` using the label object `labelObj` .

Input Argument

filePath

The fully qualified path to the file to which the label is to be attached.

labelObj

The label object used to attach the label to the file.

Example

To attach a label to a file in the currently open Simulink project with a review status `Approved`, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Create a fully qualified path for the file.

   Note The use of the `getRootDirectory` method in this step is optional.

```
file = fullfile(p.getRootDirectory, 'work', 'myExample.c');
```

3. Create a label object.

```
labelObj = Simulink.ModelManagement.Project.Label('Review','Approved')
```

4. Use the `attachLabelToFile` method.

```
p.attachLabelToFile(file,labelObj);
```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.addFil

Purpose

(To be removed) Add a file to the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.addFileToProject` will be removed in a future release. Use `addFile` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.addFileToProject(filePath)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.addFileToProject(filePath)` adds a file on the specified `filePath` to the currently open Simulink project.

Input Argument

filePath

The fully qualified path to the file within the root folder of the project.

Example

To add a file to the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Create a fully qualified path for the file.

   Note The use of the `getRootDirectory` method in this step is optional.

   ```
   file = fullfile(p.getRootDirectory, 'work', 'newshortcut.m');
   ```

3. Use the `addFileToProject` method.

```
p.addFileToProject(file);
```

See Also `Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.remov

Purpose

(To be removed) Remove a category from the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.removeCategory` will be removed in a future release. Use `removeCategory` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.removeCategory(categoryName)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.removeCategory(categoryName)` removes the category.

Input Argument

categoryName

The name of the category that is to be removed.

Examples

To remove a category `Owner` from the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Remove the category.

   ```
   p.removecategory('Owner');
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Simulink.ModelManagement.Project.CurrentProject.removeFile

Purpose

(To be removed) Remove a file from the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.removeFileFromProje` will be removed in a future release. Use `removeFile` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.removeFileFromProject(filePath)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.removeFileFromProject(filePath)` removes a file on the specified `filePath` from the currently open Simulink project.

Input Argument

filePath

The fully qualified path to the file within the root folder of the project.

Example

To remove a file from the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Create a fully qualified path for the file.

   Note The use of the `getDirectory` method in this step is optional.

   ```
   file = fullfile(p.getRootDirectory, 'work', 'newshortcut.m');
   ```

3. Use the `removeFileFromProject` method.

```
p.removeFileFromProject(file);
```

See Also `Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Remove a label from a category in the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.removeLabel` will be removed in a future release. Use `removeLabel` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
labelObj =
Simulink.ModelManagement.Project.Label(categoryName
   ,labelName);
p.removeLabel(labelObj)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`labelObj = Simulink.ModelManagement.Project.Label(categoryName,labelName);` creates a label object named `labelObj` in category `categoryName`.

`p.removeLabel(labelObj)` removes the label from the category.

Input Argument

categoryName

The name of the category from which the label is to be removed.

labelName

The name of the label.

Examples

To remove a label `John` from category `Owner` in the currently open Simulink project, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Create a label object.

```
labelObj= Simulink.ModelManagement.Project.Label('Owner','John');
```

3. Remove the label from the category.

   ```
   p.removeLabel(labelObj);
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Change the name of the currently open Simulink project

---

Note `Simulink.ModelManagement.Project.CurrentProject.setProjectName` will be removed in a future release. See “Simulink Projects” for links to project functions.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
p.setProjectName(newName)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`p.setProjectName(newName)` changes the name of the currently open Simulink project to the specified `newName`.

The name change occurs immediately within your current project. If you are using a version control tool for your project, you must commit the changes to the `.SimulinkProject` folder to source control.

Input Argument

newName

The new specified name for an open Simulink project that replaces the existing name.

Example

To rename the currently open Simulink project to `myProject`, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `setProjectName` method.

   ```
   p.setProjectName('myProject');
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

(To be removed) Export Simulink project to Zip file

Note `Simulink.ModelManagement.Project.CurrentProject.export` will be removed in a future release. Use `export` instead.

Syntax

```
p = Simulink.ModelManagement.Project.CurrentProject()
export(p, filename.zip)
export(p, filename.zip, definitionType)
```

Description

`p = Simulink.ModelManagement.Project.CurrentProject()` creates an instance of the `CurrentProject` class.

`export(p, filename.zip)` exports the project `p` to the Zip file specified in `filename`. `filename` can be a file name, in which case Simulink exports the file to the current folder, or a fully qualified path name.

`export(p, filename.zip, definitionType)` exports the project using the specified `definitionType` for the project definition files. 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. `definitionType` can be `Simulink.ModelManagement.Project.DefinitionFiles.SingleFile` or `Simulink.ModelManagement.Project.DefinitionFiles.MultiFile`. `MultiFile` is better for avoiding merging issues on shared projects. `SingleFile` is faster but is likely to cause merge issues when two users both submit changes in the same project to a source control tool.

Example

To export the currently open Simulink project to a Zip file, use the following commands.

1. Create an instance of the `CurrentProject` class.

   ```
   p = Simulink.ModelManagement.Project.CurrentProject();
   ```

2. Use the `export` method.

   ```
   export(p, 'myfilename.zip')
   ```

See Also

`Simulink.ModelManagement.Project.CurrentProject`

Purpose

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: `string`

Default: `Empty String`

Description

Specifies the block description of the associated block.

Type: `string`

Default: `Empty String`

Help

Specifies the help text that is displayed for the mask.

Type: `string`

Default: `Empty String`

Initialization

Specifies the initialization commands for the associated block.

Type: `string`

Default: `Empty String`

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: `string`

Default: `Empty String`

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: `boolean`

Values: `'on'|'off'`

Default: `` `off' ``

IconRotate

Sets icon to rotate with the block.

Type: `enum`

Values: `'none'|'fixed'|'rotates'`

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

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
removeAllParameters	Remove all existing parameters from a mask
set	Set the properties of an existing mask

How To

- "Control Masks Programmatically"

Purpose

Add a parameter to a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.addParameter(Name,Value)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.addParameter(Name,Value)` appends a parameter to the mask. If you do not specify name–value pairs as arguments with this command, Simulink creates an unnamed mask parameter with control type set to `edit`.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Type'

Type of control that is used to specify the value of this parameter.

Default: edit

'TypeOptions'

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

Default: empty

'Name'

The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Default: empty

'Prompt'

Text that identifies the parameter on the Mask Parameters dialog box.

Default: empty

'Value'

The default value of the mask parameter in the Mask Parameters dialog box.

Default: Type specific; depends on the `Type` of the parameter

'Evaluate'

Option to specify whether parameter must be evaluated.

Default: `'on'`

'Tunable'

Option to specify whether parameter is tunable.

Default: `'on'`

'Enabled'

Option to specify whether user can set parameter value.

Default: `'on'`

'Visible'

Option to set whether mask parameter is hidden or visible to the user.

Default: `'on'`

'Callback'

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog box and clicks **Apply**.

Default: empty

'TabName'

The name of the tab in the Mask Parameters dialog box where the parameter appears.

Default: empty

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Add a parameter to the mask without specifying name–value pairs for parameter attributes.

   ```
   p.addParameter;
   ```

3. Add a mask parameter of type popup that cannot be evaluated and appears on a tab named **Properties** in the Mask Parameters dialog box.

   ```
   p.addParameter('Type','popup','TypeOptions',{'Red' 'Blue' 'Green'},
   ```

See Also

`Simulink.Mask`

Simulink.Mask.copy

Purpose

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

`Simulink.Mask`

Simulink.Mask.create

Purpose

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 mask to block.

   ```
   p = Simulink.Mask.create(gcbh);
   ```

2. Create a mask using the block's path.

   ```
   p = Simulink.Mask.create('myModel/Subsystem');
   ```

See Also

`Simulink.Mask`

Purpose

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

`Simulink.Mask`

Simulink.Mask.get

Purpose

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

`Simulink.Mask`

Simulink.Mask.getOwner

Purpose

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

`Simulink.Mask`

Simulink.Mask.getParameter

Purpose

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 the number of parameters in the mask.

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

`Simulink.Mask`

Purpose

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

`Simulink.Mask`

Simulink.Mask.numParameters

Purpose

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

`Simulink.Mask`

Simulink.Mask.removeAllParameters

Purpose

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

`Simulink.Mask`

Purpose

Set the properties of an existing mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.set(Name,Value)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.set(Name,Value)` sets mask properties that you specify using name–value pairs as arguments.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Type'

Text used as title for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

'Description'

Text used as summary for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

'Help'

Text used as body text for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

'Initialization'

MATLAB code that initializes the mask.

Default: empty

'SelfModifiable'

Option to set whether the mask can modify itself during simulation.

Default: `'off'`

'Display'

MATLAB code that draws the mask icon.

Default: empty

'IconFrame'

Option to specify whether the mask icon appears inside a visible block frame.

Default: `'on'`

'IconOpaque'

Option to set the mask icon as opaque or transparent.

Default: `'on'`

'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'
```

See Also

`Simulink.Mask`

Purpose

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: `string`

Values:
`'edit'|'checkbox'|'popup'|'unidt'|'min'|'max'|'promote'`

Default: `'edit'`

TypeOptions

Specifies the option for the parameter if it exists, otherwise, it is empty. Applicable for parameters of type `popup` and `promote` .

Type: `cell array of strings`

Default: `{''}`

Name

Specifies the name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Type: `string`

Default: `Empty String`

Prompt

Specifies a string that appears as the label associated with the parameter on the mask dialog.

Type: `string`

Default: `Empty String`

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 string 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'`

Internal

Indicates if the parameter should never show on the mask dialog box.

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'`

ToolTip

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: `string`

Default: `Empty String`

TabName

Specifies the tab name of the mask dialog box where the parameter is displayed.

Type: `string`

Default: `Empty String`

Methods

set	Set properties of mask parameters

How To

- “Control Masks Programmatically”

Purpose

Set properties of mask parameters

Syntax

`Simulink.MaskParameter.set(Name,Value)`

Description

`Simulink.MaskParameter.set(Name,Value)` sets the properties of a mask parameter.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Type'

Type of control that is used to specify the value of this parameter.

Default: edit

'TypeOptions'

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

Default: empty

'Name'

The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Default: empty

'Prompt'

Text that identifies the parameter on the Mask Parameters dialog.

Default: empty

'Value'

The default value of the mask parameter in the Mask Parameters dialog.

Default: Type specific; depends on the `Type` of the parameter

'Evaluate'

Option to specify whether parameter must be evaluated.

Default: `'on'`

'Tunable'

Option to specify whether parameter is tunable.

Default: `'on'`

'Enabled'

Option to specify whether user can set parameter value.

Default: `'on'`

'Visible'

Option to set whether mask parameter is hidden or visible to the user.

Default: `'on'`

'Callback'

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog and clicks **Apply**.

Default: empty

'TabName'

The name of the tab in the Mask Parameters dialog where the parameter appears.

Default: empty

Examples

1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

2 Get a mask parameter.

```
a = p.Parameters(1);
```

3 Edit mask parameter so it is of type popup, cannot be evaluated, and appears on a tab named **Properties** in the Mask Parameters dialog.

```
a.set('Type','popup','TypeOptions',{'Red' 'Blue' 'Green'},'Evaluate
```

See Also

`Simulink.Mask` | `Simulink.MaskParameter`

Simulink.MaskUIElement

Purpose

Control mask user interface elements programmatically.

Description

Use an instance of `Simulink.MaskUIElement` to set the properties of mask user interface elements.

Properties

Type

The type of container. This can be one of ‘panel, ‘group’, ‘tab’, ‘edit’ or ‘parameter’.

Format: string **Example:** **Required By:** text, hyperlink, pushbutton, imagebutton, radiobutton, image, edit, checkbox, combobox, radiobutton, panel, tabcontainer, tab, group, parameter.

Tag

A unique tag to identify this container.

Format: string **Example:** **Required By:** text, hyperlink, pushbutton, imagebutton, radiobutton, image, edit, checkbox, combobox, radiobutton, panel, tabcontainer, tab, group, parameter.

Name

The text which is displayed along with the UI element. Depending on the container, the name is used in different ways.

Format: string/MessageID **Example:** **Required By:** edit, tab, checkbox, group, button, hyperlink, text

Rowspan

RowSpan is given by two numbers, the first number is the start row and the second is the end row. The widget will be placed so that it spans all cells from the start row to the end row. By default, the widget will take up all the space in the cells that it spans.

Format: 1x2 vector **Example:** [2 3] **Required By:** All except tabcontainer

ColSpan

ColSpan is given by two numbers, the first number is the start column and the second is the end column. The widget will be placed so that it spans all cells from the start column to the end column By default, the widget will take up all the space in the cells that it spans.

Format: 1x2 vector **Example:** [2 3] **Required By:** All except tabcontainer

LayoutGrid

This field is used to specify the size of the layout grid. The first number is the number of rows and the second is the number of columns. All widgets in this container are inserted in one or more of these grid cells. Note that all rows and columns are not guaranteed to be of equal size. The grid will adjust so that it is large enough to fully accommodate all of its widgets.

Format: 1x2 vector **Example:** [3 2] **Required By:** panel, tab, group

Callback

This is required if the widget is sensitive to events. For Mask dialogs, this is a MATLAB command string/strings. Built-in blocks can use a function pointer as well for the callback.

Format: MATLAB command string/function pointer. **Example:** **Required By:** edit, checkbox, combobox, hyperlink, button

Enabled

True if the UI element is enabled and false otherwise. True by default.

Format: boolean **Example:** **Required By:** text, hyperlink, pushbutton, imagebutton, radiobutton, image, edit, checkbox, combobox, radiobutton, panel, tabcontainer, tab, group, parameter.

Visible

True is, the widget is visible and false otherwise True by default.

Format: boolean **Example: Required By:** text, hyperlink, pushbutton, imagebutton, radiobutton, image, edit, checkbox, combobox, radiobutton, panel, tabcontainer, tab, group, parameter.

DefaultValue

The value which shows up by default. Not applicable to all the widgets.

Format: Depends on the UI element as described in the table below. **Example: Required By:** edit, combobox, checkbox, radiobutton

FilePath

It is used to find the image file to display in the UI element.

Format: string**Example: Required By:**

NameLocation

The location of the prompt string for the widget. Only certain widgets support this option. The location values that can be used are given in the table below.

Format: integer **Example: Required By:** edit, combobox

RowStretch

RowStretch is used along with LayoutGrid to specify the vertical resize behavior of this tab. The number of elements in this vector should be the same as the number of rows defined in the LayoutGrid. All the entries in the vector are normalized and together, they determine the proportion of the extra vertical space each item in the container will get. For example, if there are two rows and the RowStretch is defined as [2 1], it means that row1 will get twice the amount of extra space that row2 gets.

Format: 1xn vector **Example:** [1 0.5 0 1 …… 1] **Required By:** panel, tab, group

ColStretch

This is used exactly the same as RowStretch except that it determines the resize behavior horizontally. See above for a detailed explanation of RowStretch.

Format: 1xn vector **Example:** [1 0.5 0 1 …… 1] **Required By:** panel, tab, group

ParameterName

This is used to specify the parameter which is wrapped by this UI element. The value of this property is same as the name of the variable representing the parameter.

Format: string **Example: Required By:** parameter

Methods

Inherited Methods

Examples

Concepts

- Class Attributes
- Property Attributes

Model and Block Parameters

- "Model Parameters" on page 6-2
- "Common Block Properties" on page 6-96
- "Block-Specific Parameters" on page 6-109
- "Mask Parameters" on page 6-260

Model Parameters

In this section...

"About Model Parameters" on page 6-2

"Examples of Setting Model Parameters" on page 6-95

About Model Parameters

You can query and/or modify the properties (parameters) of a Simulink diagram from the command line. Parameters that describe a model are known as model parameters, while parameters that describe a Simulink block are known as block parameters. Block parameters that are common to Simulink blocks are described as common block parameters. There are also block-specific parameters that are specific to particular blocks. Finally, there are mask parameters, which are parameters that describe a masked block.

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.

Parameter values must be specified as quoted strings. The string contents depend on the parameter and can be numeric (scalar, vector, or matrix), a variable name, a filename, or a particular value. 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 following sections list parameters that you can set for Simulink models blocks, or signals, using the `set_param` command.

This table lists and describes, in alphabetical order, parameters that describe a model. The table also includes model callback parameters (see "Callbacks for Customized Model Behavior"). The **Description** column indicates where you can set the value on a dialog box. For examples, see "Examples of Setting Model Parameters" on page 6-95.

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.	string — `{'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 **Optimization** pane of the Configuration Parameters dialog box.	string — `{'off'} | 'on'`
`AlgebraicLoopMsg`	Specifies diagnostic action to take when there is an algebraic loop. Set by **Algebraic loop** on the **Solver Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none' | {'warning'} | 'error'`

Model Parameters in Alphabetical Order (Continued)

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 **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'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 Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none' | {'warning'} | 'error'`
`AssertControl`	Enable model verification blocks in the current model either globally or locally. Set by **Model Verification block enabling** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'UseLocalSettings'} | 'EnableAll' | 'DisableAll'`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'on'` | `{'off'}`
`BlockDescription-StringDataTip`	Specifies whether to display the user description string for a block as a data tip. In the Simulink Editor, set by **Description** on the **Display > Blocks > Block Tool Tip Options** menu.	string — `'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.	string — `'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.	string — `'on'` | `{'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	**Description**	**Values**
`BlockPriorityViolationMsg`	Select the diagnostic action to take if Simulink software detects a block priority specification error. Set by **Block priority violation** on the **Solver Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'warning'} | 'error'`
`BlockReduction`	Enables block reduction optimization. Set by **Block reduction** on the **Optimization** pane of the Configuration Parameters dialog box.	string — `{'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 **Optimization** pane of the Configuration Parameters dialog box.	string — `{'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.	string — `'on' | {'off'}`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'on' | {'off'}`
`BufferReusableBoundary`	For internal use.	
`BufferReuse`	Enable reuse of block I/O buffers. Set by **Reuse block outputs** on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.	string — `{'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.	string — `{'WarnAndRepair'} | 'ErrorWithoutRepair'`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'none'` | `{'warning'}` | `'error'`
`CheckExecutionContext-PreStartOutputMsg`	Specify whether to display a warning if Simulink software detects potential initial output differences from previous releases. Set by **Check preactivation output of execution context** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `'on'` | `{'off'}`
`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 **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `'on'` | `{'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`CheckForMatrixSingularity`	See `CheckMatrixSingularityMsg` parameter for more information.	
`CheckMatrixSingularityMsg`	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.	string — `{'none'} | 'warning' | 'error'`
`CheckModelReference-TargetMessage`	Select the diagnostic action to take if Simulink software detects a target that needs to be rebuilt. Set by **"Never rebuild diagnostic"** on the **Model Referencing** pane of the Configuration Parameters dialog box.	string — `'none' | 'warning' | {'error'}`
`CheckSSInitialOutputMsg`	Enable checking for undefined initial subsystem output. Set by **Check undefined subsystem initial output** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{''}`
`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 "View Information about Buses".	Return values are `'NOT_BUS'`, `VIRTUAL_BUS`, and `NON_VIRTUAL_BUS`
`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 **Optimization** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ConfigurationManager`	Configuration manager for this model.	string — `{'None'}`
`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.	string — `{'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 Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'none'} | 'warning' | 'error'`
`ContinueFcn`	Continue simulation callback. Set by **Simulation continue function** on the **Callbacks** pane of the Model Properties dialog box.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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 **Additional data to include in report (cvdata objects)** on the **Reporting** pane of the Coverage Settings dialog box.	string — `{''}`
`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 Verification and Validation 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 **Cumulative runs** (on) / **Last run** (off) options on the **Reporting** pane of the Coverage Settings dialog box.	string — `'on' | {'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`CovCumulativeVarName`	If `CovSaveCumulativeToWorkspaceVar` is set to `on`, the Simulink Verification and Validation software saves the results of successive simulations in the workspace variable specified by this property. Set by **cvdata object name** below the selected **Save cumulative results in workspace variable** check box on the **Results** pane of the Coverage Settings dialog box.	string — `{'covCumulativeData'}`
`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 **Coverage for MATLAB Files** on the Coverage Settings dialog box.	string — `'on' | {'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`CovForceBlockReductionOff`	If `CovForceBlockReductionOff` is set to `on`, the Simulink Verification and Validation software ignores the value of the Simulink **Block reduction** parameter. The software provides coverage data for every block in the model that collects coverage.	string — `{'on'} | 'off'`
`CovHTMLOptions`	If `CovHtmlReporting` is set to `on`, use this parameter to select from a set of display options for the resulting model coverage report. On the **Reporting** pane of the Coverage Settings dialog box, select **HTML Settings** to open a dialog box for selecting these options.	String 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): • `'-aTS=1'` — Include each test in the model summary • `'-bRG=1'` — Produce bar graphs in the model summary • `'-bTC=0'` — Use two color bar graphs (red, blue) • `'-hTR=0'` — Display hit/count ratio in the model summary • `'-nFC=0'` — Do not report fully covered model objects • `'-scm=1'` — Include cyclomatic complexity numbers in summary

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
		• `'-bcm=1'` — Include cyclomatic complexity numbers in block details • `'-xEv=0'` — Filter Stateflow events from report
`CovHtmlReporting`	Set to `on` to tell the Simulink Verification and Validation software to create an HTML report containing the coverage data at the end of simulation. Set by **Generate HTML report** on the **Reporting** pane of the Coverage Settings dialog box.	string — `{'on'} | 'off'`
`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 Coverage Settings 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 **Options** pane of the Coverage Settings dialog box. Disable option `'e'` by selecting **Display coverage results**	string — `{'dw'}` Each order-independent character in the string enables a coverage metric or option as follows: • `'d'` — Enable decision coverage • `'c'` — Enable condition coverage • `'m'` — Enable MCDC coverage • `'t'` — Enable lookup table coverage • `'r'` — Enable signal range coverage

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
	using model coloring on the **Results** pane of the Coverage Settings dialog box.	• `'z'` — Enable signal size coverage • `'o'` — Enable coverage for Simulink Design Verifier™ blocks • `'s'` — Treat Simulink logic blocks as short-circuited • `'w'` — Warn when unsupported blocks exist in model • `'e'` — Eliminate model coloring for coverage results
`CovModelRefEnable`	If `CovModelRefEnable` is set to `on` or `all`, the Simulink Verification and Validation software generates coverage data for the entire model. Set by **Coverage for referenced models** on the **Coverage** pane of the Coverage Settings dialog box.	string — `'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 Coverage Settings dialog box and then clicking **Select Models**.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`CovNameIncrementing`	If `CovSaveSingleToWorkspaceVar` is set to `on`, setting `CovNameIncrementing` to `on` causes the Simulink Verification and Validation 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 **Results** pane of the Coverage Settings dialog box.	string — `'on' | {'off'}`
`CovPath`	Model path of the subsystem for which the Simulink Verification and Validation software gathers and reports coverage data. Set by selecting **Coverage for this model: <model name>** on the **Coverage** pane of the Coverage Settings dialog box and then clicking **Select Subsystem**.	string — `{'/'}`

Model Parameters in Alphabetical Order (Continued)

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. Set by **Update results on pause** on the **Results** pane of the Coverage Settings dialog box.	string — `{'on'} | 'off'`
`CovSaveCumulativeToWorkspaceVar`	If set to `on`, the Simulink Verification and Validation software accumulates and saves the results of successive simulations in the workspace variable specified by `CovCumulativeVarName`. Set by **Save cumulative results in workspace variable** on the **Results** pane of the Coverage Settings dialog box.	string — `{'on'} | 'off'`
`CovSaveName`	If `CovSaveSingleToWorkspaceVar` is set to `on`, the Simulink Verification and Validation 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 **Results**	string — `{'covdata'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
	pane of the Coverage Settings dialog box.	
`CovSaveSingleTo-WorkspaceVar`	If set to `on`, the Simulink Verification and Validation 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 **Results** pane of the Coverage Settings dialog box.	string — `{'on'} | 'off'`
`Created`	Date and time model was created. Set by **Created on** on the **History** pane of the Model Properties dialog box. See "Viewing and Editing the Model History Log" for more information.	string
`Creator`	Name of model creator. Set by **Created by** on the **History** pane of the Model Properties dialog box. See "Viewing and Editing the Model History Log" for more information.	string
`CurrentBlock`	For internal use.	
`CurrentOutputPort`	For internal use.	

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`DataLoggingOverride`	A `Simulink.SimulationData.ModelLoggingInfo` object that specifies the signal logging override settings for a model. See "Override Signal Logging Settings".	`Simulink.SimulationData.ModelLoggingI` — `{'OverrideSignals'} | 'LogAllAsSpecifiedInModel'`
`DataTransfer`	future reA Simulink.GlobalDataTransfer object that configures data transfers for models configured for concurrent execution.	string — `'on' | {'off'}`
`DataTypeOverride`	Specifies data type used to override fixed-point data types. Set by **Data type override** on the Fixed-Point Tool.	string — `{'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.	string — `{'1'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`DeleteChildFcn`	Delete child callback function. Created on the **Callbacks** pane of the Block Properties dialog box. See "Create Block Callbacks" for more information.	string — `{''}`
`Description`	Description of this model. Set by **Model description** on the **Description** pane of the Model Properties dialog box.	string — `{''}`
`Dirty`	If the parameter is `on`, the model has unsaved changes.	string — `'on' | {'off'}`
`DiscreteInherit-ContinuousMsg`	Specifies diagnostic action to take when a Unit Delay block inherits a continuous sample time. Set by **Discrete used as continuous** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none' | {'warning'} | 'error'`
`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.	
`DisplayErrorDirections`	For internal use.	

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`DisplayInvisibleSources`	For internal use.	
`DisplaySortedLists`	For internal use.	
`DisplayVectorAndFunctionCounts`	For internal use.	
`DisplayVectPropagationResults`	For internal use.	
`Echo`	For internal use.	
`EnableOverflowDetection`	For internal use.	
`ExecutionContextIcon`	Toggles display of execution context icons on this model's block diagram. In the Simulink Editor, set by **Sorted Execution Order** on the **Display > Blocks** menu.	string — `'on' | {'off'}`
`ExplicitPartitioning`	Specifies whether or not to manually map tasks (explicit mapping) or use the rate-based tasks.	string — `'on' | {'off'}`
`ExpressionFolding`	Enables expression folding. Set by **Eliminate superfluous local variables (Expression folding)** on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`

Model Parameters in Alphabetical Order (Continued)

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.	string — `{'[t, u]'}`
`ExtMode...`	Parameters whose names start with `ExtMode` apply to Simulink External Mode. For more information, see External Mode.	
`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}`
`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	string — `{'EnableAllAsError'} | 'EnableAllAsWarning' | 'UseLocalSettings' | 'DisableAll'` **Note** The `Use local settings` and `Disable all` settings are maintained for backward compatibility, but

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
	Configuration Parameters dialog box.	may be deprecated in a future release.
`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.	string — `{'xFinal'}`
`FixedStep`	Fixed-step size. Set by **Fixed step size (fundamental sample time)** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'auto'}`
`FixptConstOverflowMsg`	Specifies diagnostic action to take when a fixed-point constant underflow occurs during simulation. Set by **Detect overflow** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'none'} | 'warning' | 'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'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.	string — `{'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.	string — `'on' | {'off'}`
`ForceArrayBoundsChecking`	For internal use.	
`ForceConsistencyChecking`	For internal use.	
`ForceModelCoverage`	For internal use.	
`ForwardingTable`	Specifies the forwarding table for this library. See “Forwarding Tables” for more information.	string — `{{'old_path_1', 'new_path_1'} ... {'old_path_n', 'new_path_n'}}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ForwardingTableString`	For internal use.	
`GeneratePreprocessorConditionals`	Globally controls whether generated code contains preprocessor conditionals. This parameter is relevant only to code generation, and has no effect on the behavior of a model in Simulink. The parameter is available only for an ERT target when **Inline parameters** is selected. Set by Configuration Parameters > Code Generation > Interface > **"Generate preprocessor conditionals"**. See for more information.	string — `{'Use local settings'} | '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
`HiliteAncestors`	For internal use.	
`IgnoreBidirectionalLines`	For internal use.	
`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.	string — `'none' | {'warning'} | 'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{''}`
`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'}`
`InitialStep`	Initial step size. Set by **Initial step size** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'auto'}`
`InlineParams`	Enable inline of parameters in generated code. Set by **Inline parameters** on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.	string — `'on' | {'off'}`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'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 and inspect simulation output”** on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `'on'` | `{'off'}`
`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.	string — `'none'` | `{'warning'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`IntegerOverflowMsg`	Specify message behavior when an integer overflow occurs. Set by **Detect overflow** in the **Signals** section on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none' | {'warning'} | 'error'`
`InvalidFcnCallConnMsg`	Specify message behavior when an invalid function-call connection exists. Set by **Invalid function-call connection** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none' | 'warning' | {'error'}`
`Jacobian`	For internal use.	
`LastModifiedBy`	User name of the person who last modified this model.	string
`LastModifiedDate`	Date used for version control.	string
`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.	can'tstring — `'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.

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
		Set to `all`, displays all links.
`LibraryType`	For internal use.	
`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 **Optimization** pane of the Configuration Parameters dialog box.	string — `{'inf'} | 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 to last** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `{'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.	string — `'on' | {'off'}`
`LoadInitialState`	Load initial state from	string — `'on' | {'off'}`
`Location`	For internal use.	

Set by the **Initial state** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box.

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`Lock`	Lock or unlock a block library. Setting this parameter to `on` prevents a user from inadvertently changing a library.	string — `'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.	string — `'on' | {'off'}`
`MAModelExclusionFile`	Specifies the location of the Model Advisor exclusion file. Set by the **File Name** field on the Model Advisor Exclusion Editor dialog box.	string — `{' '}`
`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.	string — `{'1'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'1000'}`
`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 Diagnostics** pane of the Configuration Parameters dialog box.	string — `'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.	string — `{'1000'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.
`MaxNumMinSteps`	Maximum number of times the solver uses the minimum step size.	string — `{'-1'}`
`MaxOrder`	Maximum order for `ode15s`. Set by **Maximum order** on the **Solver** pane of the Configuration Parameters dialog box.	string — `'1' | '2' | '3' | '4' | {'5'}`
`MaxStep`	Maximum step size. Set by **Max step size** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'auto'}`
`MdlSubVersion`	For internal use	

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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 **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'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 strings, numeric matrices of type "double", or more structures.
`MinMaxOverflowArchiveData`	For internal use	
`MinMaxOverflowArchiveMode`	Logging type for fixed-point logging. Set by **Overwrite or merge model simulation results** in the Fixed-Point Tool.	string — `{'Overwrite'} | 'Merge'`
`MinMaxOverflowLogging`	Setting for fixed-point logging. Set by **Fixed-point instrumentation mode** in the Fixed-Point Tool.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`MinStep`	Minimum step size for the solver. Set by **Min step size** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'auto'}`
`MinStepSizeMsg`	Message shown when minimum step size is violated. Set by **Min step size violation** on the **Solver Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'warning'} | 'error'`
`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 “Model Block Normal Mode Visibility Dialog Box”.	With `set_param`, use an array of `Simulink.BlockPath` objects or cell array of cell arrays of strings, with the strings being paths to individual 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.
`ModelBlockNormaModeVisiblityBlockPath`	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 “Normal Mode Visibility”.

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ModelBrowserVisibility`	Show the Model Browser. In the Simulink Editor, set by **Model Browser** on the **View** menu.	string — `'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.	string — `{''}`
`ModelDependencies`	List of model dependencies. Set by **Model dependencies** on the **Model Referencing** pane of the Configuration Parameters dialog box.	string — `{''}`
`ModelReferenceCS-MismatchMessage`	Message shown when there is a model configuration mismatch. Set by **Model configuration mismatch** on the **Model Referencing Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'none'} | 'warning' | '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.	string — `'none' | {'warning'} | 'error'`

Model Parameters in Alphabetical Order (Continued)

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 Diagnostics** pane of the Configuration Parameters dialog box.	string — `'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.	string — `{'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.	string — `{'none'}` | `'warning'` | `'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ModelReferenceMinAlgLoopOccurrences`	Toggles the minimization of algebraic loop occurrences. Set by **Minimize algebraic loop occurrences** on the **Model Referencing** pane of the Configuration Parameters dialog box.	string — `'on' | {'off'}`
`ModelReferenceNumInstancesAllowed`	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.	string — `'Zero' | 'Single' | {'Multi'}`
`ModelReferencePassRootInputsByReference`	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.	string — `{'on'} | 'off'`
`ModelReferenceSimTargetVerbose`	This parameter is deprecated and has no effect. Use `AccelVerboseBuild` instead.	
`ModelReferenceSymbolNameMessage`	For internal use.	
`ModelReferenceTargetType`	For internal use.	

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'none'} | 'warning' | 'error'`
`ModelVersion`	Version number of model.	string — `{'1.1'}`
`ModelVersionFormat`	Format of model's version number. Set by **Model version** on the **History** pane of the Model Properties dialog box. See "Viewing and Editing the Model History Log" for more information.	string — `{'1.%<AutoIncrement:0>'}`
`ModelWorkspace`	References this model's model workspace object.	an instance of the `Simulink.ModelWorkspace` class
`ModifiedBy`	Last person to modify this model.	string

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ModifiedByFormat`	Format for the display of last modifier. Set by **Last saved by** on the **History** pane of the Model Properties dialog box. See "Viewing and Editing the Model History Log" for more information. Can also be set by **Last saved by** on the **Model history** field on the **History** pane of the Model Explorer.	string — `{'%<Auto>'}`
`ModifiedComment`	Field for user comments.	string — `{''}`
`ModifiedDate`	Date of last model modification.	string
`ModifiedDateFormat`	Format of modified date. Set by **Last saved on** on the **History** pane of the Model Properties dialog box. See "Viewing and Editing the Model History Log" for more information.	string — `{'%<Auto>'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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 "Viewing and Editing the Model History Log" for more information. Can also be set by the **Model history** field on the **History** pane of the Model Explorer.	string — `{''}`
`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.	string — `'none' | 'warning' | {'error'}`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'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.	string — `'warning' | {'error'}`
`Name`	Model name.	string
`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.	string — `{'none'} | 'warning' | 'error'`

Model Parameters in Alphabetical Order (Continued)

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}`
`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 **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`

`OutputOption`	Time step output options for variable-step solvers.
	Set by **Output options** on the **Data Import/Export** pane of the Configuration Parameters dialog box.
		string — `'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.	string — `{'yout'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`OutputTimes`	Output times set when **Output options** on the **Data Import/Export** pane of the Configuration Parameters dialog box is set to **Produce additional output**. Set by **Output times** on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `{'[]'}`
`PaperOrientation`	Printing paper orientation.	string — `'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]`
`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.	string — `{'auto'} | 'manual' | 'tiled'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
	- `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.	
`PaperSize`	Size of `PaperType` in `PaperUnits`.	vector — `[width height]` (read only)
`PaperType`	Printing paper type.	string — `'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.	string — `'normalized' | {'inches'} | 'centimeters' | 'points'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ParameterArgumentNames`	List of parameters used as arguments when this model is called as a reference. Set by **Model arguments (for referencing this model)** in the **Model Workspace** pane of the Model Explorer.	string — `{''}`
`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.	string — `'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.	string — `'none'` | `'warning'` | `{'error'}`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'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.	string — `'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.	string — `{'none'}` | `'warning'` | `'error'`
`ParamWorkspaceSource`	For internal use.	

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`Parent`	Name of the model or subsystem that owns this object. The value of this parameter for a model is an empty string.	string — `{''}`
`Pause`	Pause simulation callback. Set by **Simulation pause function** on the **Callbacks** pane of the Model Properties dialog box.	string — `{''}`
`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.	string — `'on' | {'off'}`
`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.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`PostSaveFcn`	Function invoked just after this model is saved to disk. Set by **Model post-save function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	string — `{''}`
`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.	string — `{''}`
`PreSaveFcn`	Function invoked just before this model is saved to disk. Set by **Model pre-save function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ProdBitPerChar`	Describes the length in bits of the C `char` data type supported by the production hardware to be used by this model. Set by **Number of bits: char** in the **Embedded hardware** section 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 production hardware to be used by this model. Set by **Number of bits: int** in the **Embedded hardware** section on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`
`ProdBitPerLong`	Describes the length in bits of the C `long` data type supported by the production hardware to be used by this model. Set by **Number of bits: long** in the **Embedded hardware** section on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ProdBitPerShort`	Describes the length in bits of the C `short` data type supported by the production hardware to be used by this model. Set by **Number of bits: short** in the **Embedded hardware** section on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{16}`
`ProdEndianess`	Describes the significance of the first byte of a data word of the production hardware to be used by this model. Set by **Byte ordering** in the **Embedded hardware** section on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'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. Set by **None** in the **Emulation hardware** section on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ProdHWDeviceType`	Predefined hardware device to specify the C language constraints for your microprocessor. Set by **“Device vendor”** and **Device type** in the **Embedded hardware** section on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'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** in the **Embedded hardware** section on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `'Floor' | 'Zero' | {'Undefined'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ProdLargestAtomicFloat`	Specify the largest floating-point data type that can be atomically loaded and stored on the production hardware. Set by **“Largest atomic size: floating-point”** in the **Embedded hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `'Float' | 'Double' | {'None'}`
`ProdLargestAtomicInteger`	Specify the largest integer data type that can be atomically loaded and stored on the production hardware. Set by **“Largest atomic size: integer”** in the **Embedded hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'Char'} | 'Short' | 'Int' | 'Long'`
`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** in the **Embedded hardware** section on the **Hardware Implementation** pane of the	string — `{'on'} | 'off'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
	Configuration Parameters dialog box.	
`ProdWordSize`	Describes the word length in bits of the production hardware to be used by this model. Set by **Number of bits: native** in the **Embedded hardware** section 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.	string — `'on' | {'off'}`
`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 **“Model Referencing Pane”**for more information.	string — `'on' | {'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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 Referencing Pane"**for more information.	string — `| '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.	string — `{'UseLocalSettings'} | 'DisableAll' | 'EnableAllAsWarning' | 'EnableAllAsError'`

`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`
		string — `'on' | {'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values

	`CovHtmlReporting`
	`CovMetricSettings`
	`CovModelRefEnable`
	`CovModelRefExcluded`
	`CovNameIncrementing`
	`CovPath`
	`CovReportOnPause`
	`CovSaveCumulativeToWork-SpaceVar`
	`CovSaveName`
	`CovSaveSingleToWorkspace-Var`
	If set to `off`, no model coverage data is collected or reported.
	Set by **Coverage for this model: <model name>** on the **Coverage** pane of the Coverage Settings dialog box.	

`Refine`	Refine factor. Set by **Refine factor** on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `{'1'}`
`RelTol`	Relative error tolerance. Set by **Relative tolerance** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'1e-3'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ReportName`	Name of the associated file for the Report Generator.	string — `{'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.	string — `'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.	string — `'none' | {'warning'} | 'error'`
`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.	string — `'none' | 'warning' | {'error'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `'on' | {'off'}`
`SampleTimeColors`	In the Simulink Editor, set by **Colors** on the **Display > Sample Time Display** menu.	string — `'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.	string — `{'Unconstrained'} | 'STIndependent' | 'Specified'`
`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`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`SavedCharacterEncoding`	Specifies the character set used to encode this model. See the `slCharacterEncoding` command for more information.	string
`SaveDefaultBlockParams`	For internal use.	
`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.	string — `'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.	string — `{'Array'} | 'Structure' | 'StructureWithTime'`
`SaveOutput`	Save simulation output to workspace. Set by the **Output** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `{'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.	string — `'on' | {'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`SaveTime`	Save simulation time to workspace. Set by the **Time** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`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.	string — `'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.	string — `'none' | {'warning'} | 'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ScreenColor`	Background color of the model window. In the Simulink Editor, set by **Canvas Color** on the **Diagram > Format** menu.	string — `'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.	
`SFcnCompatibilityMsg`	See `SfunCompatibilityCheckMsg` parameter for more information.	
`SFInvalidInputDataAccessInChartInitDiag`	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.	string — `'none'` | `{'warning'}` | `'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`SFNoUnconditionalDefaultTransitionDiag`	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.	string — `'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.	string — `{'on'}` | `'off'`
`SFSimEnableDebug`	Enables debugging and animation during simulation of a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by **Enable debugging/animation** on the **Simulation Target** pane of the Configuration Parameters dialog box.	string — `{'on'}` | `'off'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`SFSimOverflowDetection`	Enables overflow detection of data during simulation of a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Overflow occurs for data when a value assigned to it exceeds the numeric capacity of the data type. **Note** To enable this option, you must also select the **Data Range** check box in the Stateflow Debugger window. Set by **Enable overflow detection (with debugging)** on the **Simulation Target** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`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.	string — `'none' | {'warning'} | 'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `'none'` | `{'warning'}` | `'error'`
`SfunCompatibilityCheckMsg`	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.	string — `{'none'}` | `'warning'` | `'error'`
`SFUnconditionalTransitionShadowingDiag`	Select the diagnostic action to take when a chart contains multiple unconditional transitions that originate from the same state or the same junction. Set by **Transition shadowing** on the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box.	string — `'none'` | `{'warning'}` | `'error'`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'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.	string — `'none'` | `{'warning'}` | `'error'`
`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.	string — `'none'` | `{'warning'}` | `'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `'EnableAll'` | `{'DisableAll'}`
`ShowGrid`	Has no effect in Simulink Editor. This parameter will be removed in a future release.	string — `'on'` | `{'off'}`
`ShowLinearizationAnnotations`	Toggles linearization icons in the model.	string — `{'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.	string — `'on'` | `{'off'}`
`ShowLineDimensionsOnError`	For internal use.	
`ShowLineWidths`	Deprecated. Use `ShowLineDimensions` instead.	
`ShowLoopsOnError`	Highlight invalid loops graphically.	string — `{'on'}` | `'off'`
`ShowModelReferenceBlockIO`	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.	string — `'on'` | `{'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ShowModelReference-BlockVersion`	Toggles display of version on block. In the Simulink Editor, set by **Block Version for Referenced Models** on the **Display > Blocks** menu.	string — `'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.	string — `'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.	string — `'on'` | `{'off'}`
`ShowPortDataTypesOnError`	For internal use.	
`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.	string — `'on'` | `{'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'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.	string — `{'on'} | 'off'`
`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 "View 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.	string — `{'none'} | 'warning' | 'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'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.	string — `{'on'} | 'off'`
`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.	string — `{'logsout'}`
`SignalLoggingSaveFormat`	Format for saving signal logging data. Set by the **“Signal logging format”** field on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `{'ModelDataLogs'}| 'Dataset'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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 string specifying the signal name to propagate.	string — `{''}`
`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.	string — `{'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.	string — `{'UseLocalSettings'} | 'TryResolveAll' | 'TryResolveAll-WithWarning'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`SigSpecEnsureSampleTimeMsg`	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.	string — `'none'` | `{'warning'}` | `'error'`
`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.	string — `{'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 **Optimization** pane of the Configuration Parameters dialog box.	string — `'on'` | `{'off'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'on'} | 'off'`
`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.	string — `{''}`
`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.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{''}`
`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.	string — `{''}`
`SimIntegrity`	Detects violations of memory integrity in code generated for MATLAB Function blocks and stops execution with a diagnostic. Set by **“Ensure memory integrity”** on the **Simulation Target** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'on'} | 'off'`
`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.	string array — `{{}}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `'start'` | `'stop'` | `'pause'` | `'continue'` | `'step'` | `'update'` | `'WriteDataLogs'` | `'SimParamDialog'` | `'connect'` | `'disconnect'` | `'WriteExtModeParamVect'` | `'AccelBuild'`
`SimulationMode`	Indicates whether Simulink software should run in Normal, Accelerator, Rapid Accelerator, or External mode. In the Simulink Editor, set by the **Simulation > Mode** menu.	string — `{'normal'}` | `'accelerator'` | `'rapid-accelerator'` | `'external'`
`SimulationStatus`	Indicates simulation status.	string — `{'stopped'}` | `'updating'` | `'initializing'` | `'running'` | `'paused'` | `'terminating'` | `'external'`
`SimulationTime`	Current time value for the simulation.	double — `{0}`

Model Parameters in Alphabetical Order (Continued)

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.	string — `{''}` **Note** If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, 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.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{''}`
`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.	string — `{'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.	string — `'VariableStepDiscrete' | {'ode45'} | 'ode23' | 'ode113' | 'ode15s' | 'ode23s' | 'ode23t' | 'ode23tb' | 'FixedStepDiscrete' | 'ode5' | 'ode4' | 'ode3' | 'ode2' | 'ode1' | 'ode14x'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`SolverMode`	Solver mode for this model. This option appears when the solver type is `Fixed-step`. Set by **Tasking mode for periodic sample times** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'Auto'} | 'SingleTasking' | 'MultiTasking'`
`SolverName`	Solver used for the simulation. See `Solver` parameter for more information.	
`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 Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none' | {'warning'} | 'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'Fast'} | 'Robust'`
`SolverType`	Solver type used for the simulation. Set by **Type** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'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.	string — `'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.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`StartTime`	Simulation start time. Set by **Start time** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'0.0'}`
`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 Diagnostics** pane of the Configuration Parameters dialog box.	string — `'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.	string — `{'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.	string — `{'on'} | 'off'`

Model Parameters in Alphabetical Order (Continued)

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.	string — `{''}`
`StopTime`	Simulation stop time. Set by **Stop time** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'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.
	Set by **Mux blocks used to create bus signals** and **Bus signal treated as vector** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.
	See “Avoid Mux/Bus Mixtures” for more information.
		string — `{'None'} |`
		`'Warning' |`
		`'ErrorLevel1' |`
		`'WarnOnBusTreatedAsVector'|`
		`'ErrorOnBusTreatedAsVector'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`SupportModelReferenceSimTargetCustomCode`	For SIM target Accelerator mode code generation, include Stateflow or MATLAB custom code for code generation for a referenced model. Select this option for each referenced model for which you want code generation to include Stateflow or MATLAB code for code generation	string — `{'off'} | 'on'`
`Tag`	User-specified text that is assigned to the model's `Tag` parameter and saved with the model.	string — `{''}`
`TargetBitPerChar`	Describes the length in bits of the C `char` data type supported by the hardware used to test generated code. Set by **Number of bits: char** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{8}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`TargetBitPerInt`	Describes the length in bits of the C `int` data type supported by the hardware used to test generated code. Set by **Number of bits: int** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`
`TargetBitPerLong`	Describes the length in bits of the C `long` data type supported by the hardware used to test generated code. Set by **Number of bits: long** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`
`TargetBitPerShort`	Describes the length in bits of the C `short` data type supported by the hardware used to test generated code. Set by **Number of bits: short** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{16}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`TargetEndianess`	Describes the significance of the first byte of a data word of the hardware used to test generated code. Set by **Byte ordering** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'Unspecified'} | 'LittleEndian' | 'BigEndian'`
`TargetFcnLib`	For internal use.	
`TargetHWDeviceType`	Describes the characteristics of the hardware used to test generated code. Set by **Device vendor** and **Device type** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'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. Set by **Signed integer division rounds to** in the **Emulation hardware** section of the **Hardware Implementation** pane of the	string — `'Floor' | 'Zero' | {'Undefined'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
	Configuration Parameters dialog box.	
`TargetLargestAtomicFloat`	Specify the largest floating-point data type that can be atomically loaded and stored on the hardware used to test code. Set by **“Largest atomic size: floating-point”** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `'Float' | 'Double' | {'None'}`
`TargetLargestAtomicInteger`	Specify the largest integer data type that can be atomically loaded and stored on the hardware used to test code. Set by **“Largest atomic size: integer”** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'Char'} | 'Short' | 'Int' | 'Long'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`TargetShiftRightIntArith`	Describes whether the C compiler that creates test 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** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'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. Set by **Number of bits: native** in the **Emulation hardware** section of the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`

Model Parameters in Alphabetical Order (Continued)

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.	string — `'none' | {'warning'} | 'error'`
`TiledPageScale`	Scales the size of the tiled page relative to the model.	string — `{'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 **Solver Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'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'}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`TLC...`	Parameters whose names begin with TLC are used for code generation. See the Simulink Coder documentation for more information.	
`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.	string — `{'on'} | 'off'`
`TryForcingSFcnDF`	This flag is used for backward compatibility with user S-functions that were written prior to R12.	string — `'on' | {'off'}`
`TunableVars`	List of global (tunable) parameters. Set in the Model Parameter Configuration dialog box.	string — `{''}`
`TunableVarsStorageClass`	List of storage classes for their respective tunable parameters. Set in the Model Parameter Configuration dialog box.	string — `{''}`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`TunableVarsTypeQualifier`	List of storage type qualifiers for their respective tunable parameters. Set in the Model Parameter Configuration dialog box.	string — `{''}`
`Type`	Simulink object type (read only).	string — `{'block_diagram'}`
`UnconnectedInputMsg`	Unconnected input ports diagnostic. Set by **Unconnected block input ports** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none'` | `{'warning'}` | `'error'`
`UnconnectedLineMsg`	Unconnected lines diagnostic. Set by **Unconnected line** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	string — `'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.	string — `'none'` | `{'warning'}` | `'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'none'} | 'warning' | 'error'`
`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 **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'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.	string — `{'none'} | 'warning' | 'error'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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 **Solver Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none'` | `{'warning'}` | `'error'`
`UnnecessaryDatatype-ConvMsg`	Detect unnecessary data type conversion blocks. Set by **Unnecessary type conversions** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'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 "Viewing and Editing the Model History Log" for more information.	string — `{'UpdateHistoryNever'}` | `'UpdateHistoryWhenSave'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`UpdateModelReference-Targets`	Specify whether to rebuild simulation and Simulink Coder targets for referenced models before updating, simulating, or generating code for this model. Set by **Rebuild options** on the **Model Referencing** pane of the Configuration Parameters dialog box.	string — `'IfOutOfDate' | 'Force' | 'AssumeUpToDate' | {'IfOutOfDateOrStructuralChange'}`
`UseAnalysisPorts`	For internal use.	
`UseIntDivNetSlope`	Use integer division to handle net slopes that are reciprocals of integers.	string — `'on' | {'off'}`
`VectorMatrixConversionMsg`	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.	string — `{'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)

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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). To get version information without loading the block diagram into memory, see `Simulink.MDLInfo`.	double (read only)
`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.	string — `'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.	string — `{'UseLocalSettings'} | 'DisableAll' | 'EnableAllAsWarning' | 'EnableAllAsError'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`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.	string — `{'UseLocalSettings'} | 'DisableAll' | 'EnableAllAsWarning' | 'EnableAllAsError'`
`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.	string — `{'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.	string — `{'Nonadaptive'} | 'Adaptive'`

Model Parameters in Alphabetical Order (Continued)

Parameter	Description	Values
`ZeroCrossControl`	Enable zero-crossing detection. Set by **Zero-crossing control** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'UseLocalSettings'} | 'EnableAll' | 'DisableAll'`
`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.	string — `{'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 `SaveFcn` callback.

```
set_param('mymodel','SaveFcn','my_save_cb')
```

Common Block Properties

In this section...

"About Common Block Properties" on page 6-96

"Examples of Setting Block Properties" on page 6-107

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 6-107).

Common Block Properties

Property	Description	Values
`AncestorBlock`	Name of the library block that the block is linked to (for blocks with a disabled link).	string
`AttributesFormatString`	String format specified for block annotations in the Block Parameters dialog box.	string
`BackgroundColor`	Block background color.	RGB value array string | `[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.
`BlockDescription`	Block description shown in the Block Properties dialog box.	string
`BlockType`	Block type (read only).	string

Common Block Properties (Continued)

Property	Description	Values
`ClipboardFcn`	Function called when block is copied to the clipboard (**Ctrl+C**)	string
`CloseFcn`	Function called when `close_system` is run on block.	string
`Commented`	Exclude block from simulation.	`{'off'} | 'on'`
`CompiledPort-ComplexSignals`	Complexity of port signals after updating diagram.	structure array
`CompiledPortDataTypes`	Data types of port signals after updating diagram.	structure array
`CompiledPortDimensions`	Dimensions of port signals after updating diagram.	structure array
`CompiledPortFrameData`	Frame mode of port signals after updating diagram.	structure array
`CompiledPortWidths`	Structure of port widths after updating diagram.	structure array
`CompiledSampleTime`	Block sample time after updating diagram.	vector [`sample time`, `offset time`]
`CopyFcn`	Function called when block is copied.	string
`DataTypeOverrideCompiled`	For internal use.	

Common Block Properties (Continued)

Property	Description	Values
`DeleteFcn`	Function called when block is deleted. If a block is graphically deleted, you can still undo the operation and call the block's `UndoDeleteFcn`. In addition, for graphically deleted blocks, the block's `DestroyFcn` is still called when the model is closed or any subsystem containing the block is destroyed using `delete_block`.	MATLAB expression
`DestroyFcn`	Function called when block is destroyed. If you run the `delete_block` command for a block, it first calls the block's`DeleteFcn`, then calls the `DestroyFcn` for that block; no undo is possible. The `DestroyFcn` is also called when you close the model or invoke `delete_block` on a subsystem containing the block.	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`	Names/attributes of parameters in block parameter dialog box.	structure
`DropShadow`	Display drop shadow.	`{'off'} | 'on'`

Common Block Properties (Continued)

Property	Description	Values
`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 property setting.	`{'none'} | 'log' | 'monitor'`
`FontAngle`	Font angle.	`'normal' | 'italic' | 'oblique' | {'auto'}`
`FontName`	Font.	string
`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'}`

Common Block Properties (Continued)

Property	Description	Values
`ForegroundColor`	Foreground color of block's icon.	string `{'black'} |` `[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.
`Handle`	Block handle.	real
`HiliteAncestors`	For internal use.	
`InitFcn`	Initialization function for a masked block. Created on the **Callbacks** pane of the Model Properties dialog box. See "Create Model Callbacks" in the Using Simulink documentation for further information.	MATLAB expression
`InputSignalNames`	Names of input signals.	cell array
`IOSignalStrings`		list
`IOType`	Signal & Scope Manager type. For internal use.	
`LineHandles`	Handles of lines connected to block.	struct
`LinkStatus`	Link status of block. Updates out-of-date reference blocks when queried using `get_param`.	`{'none'} | 'resolved' | 'unresolved' | 'implicit' | 'inactive' | 'restore' | 'propagate'`
`LoadFcn`	Function called when block is loaded.	MATLAB expression
`MinMaxOverflow-Logging_Compiled`	For internal use.	

Common Block Properties (Continued)

Property	Description	Values
`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.	string
`NameChangeFcn`	Function called when block name is changed.	MATLAB expression
`NamePlacement`	Position of block name.	`{'normal'} | 'alternate'`
`ObjectParameters`	Names/attributes of block 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.	string `{'untitled'}`
`ParentCloseFcn`	Function called when parent subsystem is closed. The `ParentCloseFcn` of blocks at the root model level is not called when the model is closed.	MATLAB expression

Common Block Properties (Continued)

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.	structure array

Common Block Properties (Continued)

Property	Description	Values
	- `SrcBlock` Handle of the block connected to this port. This field is null for output ports. - `SrcPort` Number of the port connected to this port. This field is null for output ports. - `DstBlock` Handle of the block to which this port is connected. This field is null for input ports. - `DstPort` Number of the port to which this port is connected. This field is null for input ports.	
`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.	structure array

Common Block Properties (Continued)

Property	Description	Values
	• `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. • `RConn` Handles of the block's right connection ports. • `Ifaction` Handle of the block's action port.	
`Ports`	The value of this property is a vector that specifies the numbers of each kind of port. The order of the vector's elements corresponds to the following port types: • `Inport` • `Outport` • `Enable`	vector

Common Block Properties (Continued)

Property	Description	Values
	• `Trigger` • `State` • `LConn` • `RConn` • `Ifaction`	
`Position`	Position of block in model window.	vector of coordinates (in pixels) not enclosed in quotation marks: `[left top right bottom]` **Note** The origin is located in the upper left corner of the model window. The maximum value for a coordinate is 32767.
`PostSaveFcn`	Function called after the block is saved. Created on the **Callbacks** pane of the Model Properties dialog box. See "Create Model Callbacks" in the Using Simulink documentation for further information.	MATLAB expression
`PreCopyFcn`	Function called before the block is copied. See in the Using Simulink documentation for details.	MATLAB expression

Common Block Properties (Continued)

Property	Description	Values
`PreDeleteFcn`	Function called before the block is deleted. See in the Using Simulink documentation for details.	MATLAB expression
`PreSaveFcn`	Function called before the block is saved.	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.	string `{''}`
`ReferenceBlock`	Name of the library block to which this block links.	string `{''}`
`RequirementInfo`	For internal use.	
`RTWData`	User specified data, used by Simulink Coder software.	
`SampleTime`	Value of the sample time parameter.	string
`Selected`	Status of whether or not block is selected.	`{'on'} | 'off'`
`ShowName`	Display block name.	`{'on'} | 'off'`
`StartFcn`	Function called at the start of a simulation.	MATLAB expression

Common Block Properties (Continued)

Property	Description	Values
`StatePerturbation-ForJacobian`	State perturbation size to use during linearization. See "Perturbation Level of Blocks Perturbed During Linearization" in the Simulink Control Design documentation for details.	string
`StaticLinkStatus`	Link status of block. Does not update out-of-date reference blocks when queried using `get_param`.	`{'none'} | 'resolved' | 'unresolved' | 'implicit' | 'inactive' | 'restore' | 'propagate'`
`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.	string `{''}`
`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. The position vector is *not* enclosed in quotation marks.

```
set_param('mymodel/Gain','Position', [50 250 125 275])
```

Block-Specific Parameters

You can query and/or modify the properties (parameters) of a Simulink diagram from the command line. Parameters that describe a model are known as model parameters, while parameters that describe a Simulink block are known as block parameters. Block parameters that are common to Simulink blocks are described as common block parameters. There are also block-specific parameters that are specific to particular blocks. Finally, there are mask parameters, which are parameters that describe a masked block.

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.

These tables list block-specific parameters for all 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 6-96). The type listed for masked blocks is the value of the `MaskType` parameter (see “Mask Parameters” on page 6-260).

The **Dialog Box Prompt** column indicates the text of the prompt for the parameter on 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).

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.

- Continuous Library Block Parameters

- Discontinuities Library Block Parameters
- Discrete Library Block Parameters
- Logic and Bit Operations Library Block Parameters
- Lookup Tables Block Parameters
- Math Operations Library Block Parameters
- Model Verification Library Block Parameters
- Model-Wide Utilities Library Block Parameters
- Ports & Subsystems Library Block Parameters
- Signal Attributes Library Block Parameters
- Signal Routing Library Block Parameters
- Sinks Library Block Parameters
- Sources Library Block Parameters
- User-Defined Functions Library Block Parameters
- Additional Discrete Block Library Parameters
- Additional Math: Increment - Decrement Block Parameters

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	string — `{'inf'}`
Integrator (`Integrator`)		
`ExternalReset`	External reset	string — `{'none'} | 'rising' | 'falling' | 'either' | 'level' | 'level hold'`

Continuous Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`InitialConditionSource`	Initial condition source	string — `{'internal'} | 'external'`
`InitialCondition`	Initial condition	scalar or vector — `{'0'}`
`LimitOutput`	Limit output	string — `{'off'} | 'on'`
`UpperSaturationLimit`	Upper saturation limit	scalar or vector — `{'inf'}`
`LowerSaturationLimit`	Lower saturation limit	scalar or vector — `{'-inf'}`
`ShowSaturationPort`	Show saturation port	string — `{'off'} | 'on'`
`ShowStatePort`	Show state port	string — `{'off'} | 'on'`
`AbsoluteTolerance`	Absolute tolerance	string, scalar, or vector — `{'auto'} | {' 1'} |` any real scalar or vector
`IgnoreLimit`	Ignore limit and reset when linearizing	string — `{'off'} | 'on'`
`ZeroCross`	Enable zero–crossing detection	string — `'off' | {'on'}`
`ContinuousStateAttributes`	State Name	string — `{''}` | user-defined
Second-Order Integrator (`SecondOrderIntegrator`)		
`ICSourceX`	Initial condition source x	string — `{'internal'} | 'external'`
`ICX`	Initial condition x	scalar or vector — `{'0'}`
`LimitX`	Limit x	string — `{'off'} | 'on'`
`UpperLimitX`	Upper limit x	scalar or vector — `{'inf'}`
`LowerLimitX`	Lower limit x	scalar or vector — `{'-inf'}`
`AbsoluteToleranceX`	Absolute tolerance x	string, scalar, or vector — `{'auto'} | {' 1'} |` any real scalar or vector
`StateNameX`	State name x	string — `{}` | user-defined

Continuous Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ICSourceDXDT`	Initial condition source dx/dt	string — `{'internal'}` | `'external'`
`ICDXDT`	Initial condition dx/dt	scalar or vector — `{'0'}`
`LimitDXDT`	Limit dx/dt	string — `{'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	string, scalar, or vector — `{'auto'}` | `{' 1'}` | any real scalar or vector
`StateNameDXDT`	State name dx/dt	string — `{}` | user-defined
`ExternalReset`	External reset	string — `{'none'}` | `'rising'` | `'falling'` | `'either'`
`ZeroCross`	Enable zero-crossing detection	string — `{'on'}` | `'off'`
`ReinitDXDTwhenXreachesSaturation`	Reinitialize dx/dt when x reaches saturation	string — `{'off'}` | `'on'`
`IgnoreStateLimitsAndResetForLinearization`	Ignore state limits and the reset for linearization	string — `{'off'}` | `'on'`
`ShowOutput`	Show output	string — `{'both'}` | `'x'` | `'dxdt'`
State-Space (`StateSpace`)		
`A`	A	matrix — `{'1'}`
`B`	B	matrix — `{'1'}`
`C`	C	matrix — `{'1'}`
`D`	D	matrix — `{'1'}`
`X0`	Initial conditions	vector — `{'0'}`

Continuous Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`AbsoluteTolerance`	Absolute tolerance	string, scalar, or vector — `{'auto'}` | `{' 1'}` | any real scalar or vector
`ContinuousStateAttributes`	State Name	string — `{''}` | user-defined
Transfer Fcn (`TransferFcn`)		
`Numerator`	Numerator coefficients	vector or matrix — `{'[1]'}`
`Denominator`	Denominator coefficients	vector — `{'[1 1]'}`
`AbsoluteTolerance`	Absolute tolerance	string, scalar, or vector — `{'auto'}` | `{' 1'}` | any real scalar or vector
`ContinuousStateAttributes`	State Name	string — `{''}` | 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	string — `{'off'}` | `'on'`
`TransDelayFeedthrough`	Direct feedthrough of input during linearization	string — `{'off'}` | `'on'`
`PadeOrder`	Pade order (for linearization)	string — `{'0'}`
Variable Time Delay (`VariableTimeDelay`)		
`VariableDelayType`	Select delay type	string — `'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	string — `{'off'}` | `'on'`

Continuous Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ZeroDelay`	Handle zero delay	string — `{'off'}` | `'on'`
`TransDelayFeedthrough`	Direct feedthrough of input during linearization	string — `{'off'}` | `'on'`
`PadeOrder`	Pade order (for linearization)	string — `{'0'}`
`ContinuousStateAttributes`	State Name	string — `{''}` | user-defined
Variable Transport Delay (`VariableTransportDelay`)		
`VariableDelayType`	Select delay type	string — `{'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	string — `{'off'}` | `'on'`
`TransDelayFeedthrough`	Direct feedthrough of input during linearization	string — `{'off'}` | `'on'`
`PadeOrder`	Pade order (for linearization)	string — `{'0'}`
`AbsoluteTolerance`	Absolute tolerance	string, scalar, or vector — `{'auto'}` | `{' 1'}` | any positive real scalar or vector
`ContinuousStateAttributes`	State Name	string — `{''}` | user-defined
Zero-Pole (`ZeroPole`)		
`Zeros`	Zeros	vector — `{'[1]'}`
`Poles`	Poles	vector — `{'[0 -1]'}`
`Gain`	Gain	vector — `{'[1]'}`

Continuous Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`AbsoluteTolerance`	Absolute tolerance	string, scalar, or vector — `{'auto'}` | `{' 1'}` | any positive real scalar or vector
`ContinuousStateAttributes`	State Name	string — `{''}` | 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	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Coulomb & Viscous Friction (`Coulombic and Viscous Friction`) (masked subsystem)		
`offset`	Coulomb friction value (Offset)	string — `{'[1 3 2 0]'}`
`gain`	Coefficient of viscous friction (Gain)	string — `{'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	string — `'off' | {'on'}`
`LinearizeAsGain`	Treat as gain when linearizing	string — `'off' | {'on'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Dead Zone Dynamic (`Dead Zone Dynamic`) (masked subsystem)		

Discontinuities Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Hit Crossing (`HitCross`)		
`HitCrossingOffset`	Hit crossing offset	scalar or vector — `{'0'}`
`HitCrossingDirection`	Hit crossing direction	string — `'rising' | 'falling' | {'either'}`
`ShowOutputPort`	Show output port	string — `'off' | {'on'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Quantizer (`Quantizer`)		
`QuantizationInterval`	Quantization interval	scalar or vector — `{'0.5'}`
`LinearizeAsGain`	Treat as gain when linearizing	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Rate Limiter (`RateLimiter`)		
`RisingSlewLimit`	Rising slew rate	string — `{'1'}`
`FallingSlewLimit`	Falling slew rate	string — `{'-1'}`
`SampleTimeMode`	Sample time mode	string — `'continuous' | {'inherited'}`
`InitialCondition`	Initial condition	string — `{'0'}`
`LinearizeAsGain`	Treat as gain when linearizing	string — `'off' | {'on'}`
Rate Limiter Dynamic (`Rate Limiter Dynamic`) (masked subsystem)		
Relay (`Relay`)		
`OnSwitchValue`	Switch on point	string — `{'eps'}`
`OffSwitchValue`	Switch off point	string — `{'eps'}`
`OnOutputValue`	Output when on	string — `{'1'}`
`OffOutputValue`	Output when off	string — `{'0'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`

Discontinuities Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `'Inherit: Inherit via back propagation' | {'Inherit: All ports same datatype'} | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>'`
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'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	string — `'off' | {'on'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Discontinuities Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `'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	string — `{'off'}` | `'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
Saturation Dynamic (`Saturation Dynamic`) (masked subsystem)		
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	

Discontinuities Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`DoSatur`	Saturate on integer overflow	string — `{'off'} | 'on'`
Wrap To Zero (`Wrap To Zero`) (masked subsystem)		
`Threshold`	Threshold	string — `{'255'}`

Discrete Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Delay (`Delay`)		
`DelayLengthSource`	Delay length > Source	string — `{'Dialog'} | 'Input port'`
`DelayLength`	Delay length > Value	string — `{'2'}`
`DelayLengthUpperLimit`	Delay length > Upper limit	string — `{'100'}`
`InitialConditionSource`	Initial condition > Source	string — `{'Dialog'} | 'Input port'`
`InitialCondition`	Initial condition > Value	string — `{'0.0'}`
`ExternalReset`	External reset	string — `{'None'} | 'Rising' | 'Falling' | 'Either' | 'Level' | 'Level hold'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`InputProcessing`	Input processing	string — `'Columns as channels (frame based)' | {'Elements as channels (sample based)'} | 'Inherited'`
`UseCircularBuffer`	Use circular buffer for state	string — `{'off'} | 'on'`
`PreventDirectFeedthrough`	Prevent direct feedthrough by increasing delay length to lower limit	string — `{'off'} | 'on'`
`RemoveProtectionDelayLength`	Remove protection against out-of-range delay length in generated code	string — `{'off'} | 'on'`
`DiagnosticForOutOfRangeDelayLength`	Diagnostic for out-of-range delay length	string — `{'None'} | 'Warning' | 'Error'`
`SampleTime`	Sample time (–1 for inherited)	string — `{'-1'}`
`StateName`	State name	string — `{''}`
`StateMustResolveToSignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`CodeGenStateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`CodeGenStateStorageTypeQualifier`	Code generation storage type qualifier	string — `{''}`
Difference (`Difference`) (masked subsystem)		
`ICPrevInput`	Initial condition for previous input	string — `{'0.0'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`
Discrete Derivative (`Discrete Derivative`) (masked subsystem)		
`gainval`	Gain value	string — `{'1.0'}`
`ICPrevScaledInput`	Initial condition for previous weighted input K*u/Ts	string — `{'0.0'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`
Discrete FIR Filter (`Discrete FIR Filter`)		
`CoefSource`	Coefficient source	string — `{'Dialog parameters'} | 'Input port'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`FilterStructure`	Filter structure	string — `{'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	string — `'Columns as channels (frame based)' | {'Elements as channels (sample based)'}`
`InitialStates`	Initial states	scalar or vector — `{'0'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`CoefMin`	Coefficients minimum	string — `{'[]'}`
`CoefMax`	Coefficients maximum	string — `{'[]'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`TapSumDataTypeStr`	Tap sum data type	string — `{'Inherit: Same as input'} | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`CoefDataTypeStr`	Coefficients data type	string — `{'Inherit: Same word length as input'} | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)'`
`ProductDataTypeStr`	Product output data type	string — `{'Inherit: Inherit via internal rule'} | 'Inherit: Same as input' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)'`
`AccumDataTypeStr`	Accumulator data type	string — `{'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	string — `'Inherit:  Same as input' | {'Inherit: Same as accumulator'} | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `'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	string — `{'off'}` | `'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'}` | `'on'`
Discrete Filter (`DiscreteFilter`)		
`Numerator`	Numerator coefficients	vector — `{'[1]'}`
`Denominator`	Denominator coefficients	vector — `{'[1 0.5]'}`
`IC`	Initial states	string — `{'0'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'1'}`
`a0EqualsOne`	Optimize by skipping divide by leading denominator coefficient (a0)	string — `{'off'}` | `'on'`
`NumCoefMin`	Numerator coefficient minimum	string — `{'[]'}`
`NumCoefMax`	Numerator coefficient maximum	string — `{'[]'}`
`DenCoefMin`	Denominator coefficient minimum	string — `{'[]'}`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`DenCoefMax`	Denominator coefficient maximum	string — `{'[]'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`StateDataTypeStr`	State data type	string — `{'Inherit: Same as input'} | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)'`
`NumCoefDataTypeStr`	Numerator coefficient data type	string — `{'Inherit: Inherit via internal rule'} | 'int8' | 'int16' | 'int32' | 'fixdt(1,16)' | 'fixdt(1,16,0)'`
`DenCoefDataTypeStr`	Denominator coefficient data type	string — `{'Inherit: Inherit via internal rule'} | 'int8' | 'int16' | 'int32' | 'fixdt(1,16)' | 'fixdt(1,16,0)'`
`NumProductDataTypeStr`	Numerator product output data type	string — `{'Inherit: Inherit via internal rule'} | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)'`
`DenProductDataTypeStr`	Denominator product output data type	string — `{'Inherit: Inherit via internal rule'} | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`NumAccumDataTypeStr`	Numerator accumulator data type	string — `{'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	string — `{'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	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveTo SignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWStateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageTypeQualifier`	Code generation storage type qualifier	string — `{''}`
Discrete State-Space (`DiscreteStateSpace`)		
`A`	A	matrix — `{'1'}`
`B`	B	matrix — `{'1'}`
`C`	C	matrix — `{'1'}`
`D`	D	matrix — `{'1'}`
`X0`	Initial conditions	vector — `{'0'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'1'}`
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveToSignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`RTWStateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageTypeQualifier`	Code generation storage type qualifier	string — `{''}`
Discrete Transfer Fcn (`DiscreteTransferFcn`)		
`Numerator`	Numerator coefficients	vector — `{'[1]'}`
`Denominator`	Denominator coefficients	vector — `{'[1 0.5]'}`
`InitialStates`	Initial states	string — `{'0'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'1'}`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`a0EqualsOne`	Optimize by skipping divide by leading denominator coefficient (a0)	string — `{'off'} | 'on'`
`NumCoefMin`	Numerator coefficient minimum	string — `{'[]'}`
`NumCoefMax`	Numerator coefficient maximum	string — `{'[]'}`
`DenCoefMin`	Denominator coefficient minimum	string — `{'[]'}`
`DenCoefMax`	Denominator coefficient maximum	string — `{'[]'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`StateDataTypeStr`	State data type	string — `{'Inherit:  Same as input'} | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)'`
`NumCoefDataTypeStr`	Numerator coefficient data type	string — `{'Inherit: Inherit via internal rule'} | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)'`
`DenCoefDataTypeStr`	Denominator coefficient data type	string — `{'Inherit: Inherit via internal rule'} | 'int8' | 'int16' | 'int32' | 'fixdt(1,16,0)'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`NumProductDataTypeStr`	Numerator product output data type	string — {`'Inherit: Inherit via internal rule'`} | `'Inherit: Same as input'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'`
`DenProductDataTypeStr`	Denominator product output data type	string — {`'Inherit: Inherit via internal rule'`} | `'Inherit: Same as input'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'`
`NumAccumDataTypeStr`	Numerator accumulator data type	string — {`'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	string — {`'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	string — {`'Inherit: Inherit via internal rule'`} | `'Inherit: Same as input'` | `'int8'` | `'int16'` | `'int32'` | `'fixdt(1,16,0)'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`LockScale`	Lock data type settings against changes by the fixed-point tools	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnInteger Overflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveTo SignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`RTWStateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageType Qualifier`	Code generation storage type qualifier	string — `{''}`
Discrete Zero-Pole (`DiscreteZeroPole`)		
`Zeros`	Zeros	vector — `{'[1]'}`
`Poles`	Poles	vector — `{'[0 0.5]'}`
`Gain`	Gain	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'1'}`
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveTo SignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`RTWStateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWStateStorageTypeQualifier`	Code generation storage type qualifier	string — `{''}`
Discrete-Time Integrator (`DiscreteIntegrator`)		
`IntegratorMethod`	Integrator method	string — `{'Integration: Forward Euler'} | 'Integration:  Backward Euler' | 'Integration: Trapezoidal' | 'Accumulation:  Forward Euler' | 'Accumulation: Backward Euler' | 'Accumulation: Trapezoidal'`
`gainval`	Gain value	string — `{'1.0'}`
`ExternalReset`	External reset	string — `{'none'} | 'rising' | 'falling' | 'either' | 'level' | 'sampled level'`
`InitialConditionSource`	Initial condition source	string — `{'internal'} | 'external'`
`InitialCondition`	Initial condition	scalar or vector — `{'0'}`
`InitialConditionMode`	Use initial condition as initial and reset value for	string — `'State only (most efficient)' | {'State and output'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'1'}`
`LimitOutput`	Limit output	string — `{'off'} | 'on'`
`UpperSaturationLimit`	Upper saturation limit	scalar or vector — `{'inf'}`
`LowerSaturationLimit`	Lower saturation limit	scalar or vector — `{'-inf'}`
`ShowSaturationPort`	Show saturation port	string — `{'off'} | 'on'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ShowStatePort`	Show state port	string — `{'off'} | 'on'`
`IgnoreLimit`	Ignore limit and reset when linearizing	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveToSignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWStateStorageClass`	Code generation storage class	string — {`'Auto'`} | `'ExportedGlobal'` | `'ImportedExtern'` | `'ImportedExternPointer'`
`RTWStateStorageTypeQualifier`	Code generation storage type qualifier	string — {`''`}
First-Order Hold (`First-Order Hold`) (masked subsystem)		
`Ts`	Sample time	string — {`'1'`}
Memory (`Memory`)		
`X0`	Initial condition	scalar or vector — {`'0'`}
`InheritSampleTime`	Inherit sample time	string — {`'off'`} | `'on'`
`LinearizeMemory`	Direct feedthrough of input during linearization	string — {`'off'`} | `'on'`
`LinearizeAsDelay`	Treat as a unit delay when linearizing with discrete sample time	string — {`'off'`} | `'on'`
`StateIdentifier`	State name	string — {`''`}
`StateMustResolveToSignalObject`	State name must resolve to Simulink signal object	string — {`'off'`} | `'on'`
`RTWStateStorageClass`	Code generation storage class	string — {`'Auto'`} | `'ExportedGlobal'` | `'ImportedExtern'` | `'ImportedExternPointer'`
`RTWStateStorageTypeQualifier`	Code generation storage type qualifier	string — {`''`}
Tapped Delay (S-Function) (`Tapped Delay Line`) (masked subsystem)		
`vinit`	Initial condition	string — {`'0.0'`}
`samptime`	Sample time	string — {`'-1'`}
`NumDelays`	Number of delays	string — {`'4'`}

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`DelayOrder`	Order output vector starting with	string — `{'Oldest'} | 'Newest'`
`includeCurrent`	Include current input in output vector	string — `{'off'} | 'on'`
Transfer Fcn First Order (`First Order Transfer Fcn`) (masked subsystem)		
`PoleZ`	Pole (in Z plane)	string — `{'0.95'}`
`ICPrevOutput`	Initial condition for previous output	string — `{'0.0'}`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`
Transfer Fcn Lead or Lag (`Lead or Lag Compensator`) (masked subsystem)		
`PoleZ`	Pole of compensator (in Z plane)	string — `{'0.95'}`
`ZeroZ`	Zero of compensator (in Z plane)	string — `{'0.75'}`
`ICPrevOutput`	Initial condition for previous output	string — `{'0.0'}`
`ICPrevInput`	Initial condition for previous input	string — `{'0.0'}`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`

Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Transfer Fcn Real Zero (`Transfer Fcn Real Zero`) (masked subsystem)		
`ZeroZ`	Zero (in Z plane)	string — `{'0.75'}`
`ICPrevInput`	Initial condition for previous input	string — `{'0.0'}`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`
Unit Delay (`UnitDelay`)		
`InitialCondition`	Initial condition	scalar or vector — `{'0'}`
`InputProcessing`	Input processing	string — `'Columns as channels (frame based)' | {'Elements as channels (sample based)'} | 'Inherited'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`StateName`	State name	string — `{''}`
`StateMustResolveToSignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`CodeGenStateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`CodeGenStateStorageTypeQualifier`	Code generation storage type qualifier	string — `{''}`
Zero-Order Hold (`ZeroOrderHold`)		
`SampleTime`	Sample time (-1 for inherited)	string — `{'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)	string — `{'0'}`
Bit Set (`Bit Set`) (masked subsystem)		
`iBit`	Index of bit (0 is least significant)	string — `{'0'}`
Bitwise Operator (S-Function) (`Bitwise Operator`) (masked subsystem)		
`logicop`	Operator	string — `{'AND'} | 'OR' | 'NAND' | 'NOR' | 'XOR' | 'NOT'`
`UseBitMask`	Use bit mask ...	string — `'off' | {'on'}`
`NumInputPorts`	Number of input ports	string — `{'1'}`
`BitMask`	Bit Mask	string — `{'bin2dec('11011001')'}`
`BitMaskRealWorld`	Treat mask as	string — `'Real World Value' | {'Stored Integer'}`
Combinatorial Logic (`CombinatorialLogic`)		
`TruthTable`	Truth table	string — `{'[0 0;0 1;0 1;1 0;0 1;1 0;1 0;1 1]'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Compare To Constant (`Compare To Constant`) (masked subsystem)		
`relop`	Operator	string — `'==' | '~=' | '<' | {'<='} | '>=' | '>'`
`const`	Constant value	string — `{'3.0'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'} | 'uint8'`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
Compare To Zero (`Compare To Zero`) (masked subsystem)		

Logic and Bit Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`relop`	Operator	string — `'=='` | `'~='` | `'<'` | `{'<='}` | `'>='` | `'>'`
`OutDataTypeStr`	Output data type	string — `{'boolean'}` | `'uint8'`
`ZeroCross`	Enable zero-crossing detection	string — `'off'` | `{'on'}`
Detect Change (`Detect Change`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'}` | `'uint8'`
Detect Decrease (`Detect Decrease`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0.0'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'}` | `'uint8'`
Detect Fall Negative (`Detect Fall Negative`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'}` | `'uint8'`
Detect Fall Nonpositive (`Detect Fall Nonpositive`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'}` | `'uint8'`
Detect Increase (`Detect Increase`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0.0'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'}` | `'uint8'`
Detect Rise Nonnegative (`Detect Rise Nonnegative`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0'}`

Logic and Bit Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `{'boolean'} | 'uint8'`
Detect Rise Positive (`Detect Rise Positive`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'} | 'uint8'`
Extract Bits (`Extract Bits`) (masked subsystem)		
`bitsToExtract`	Bits to extract	string — `{'Upper half'} | 'Lower half' | 'Range starting with most significant bit' | 'Range ending with least significant bit' | 'Range of bits'`
`numBits`	Number of bits	string — `{'8'}`
`bitIdxRange`	Bit indices ([start end], 0-based relative to LSB)	string — `{'[0 7]'}`
`outScalingMode`	Output scaling mode	string — `{'Preserve fixed-point scaling'} | 'Treat bit field as an integer'`
Interval Test (`Interval Test`) (masked subsystem)		
`IntervalClosedRight`	Interval closed on right	string — `'off' | {'on'}`
`uplimit`	Upper limit	string — `{'0.5'}`
`IntervalClosedLeft`	Interval closed on left	string — `'off' | {'on'}`
`lowlimit`	Lower limit	string — `{'-0.5'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'} | 'uint8'`
Interval Test Dynamic (`Interval Test Dynamic`) (masked subsystem)		

Logic and Bit Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IntervalClosedRight`	Interval closed on right	string — `'off' | {'on'}`
`IntervalClosedLeft`	Interval closed on left	string — `'off' | {'on'}`
`OutDataTypeStr`	Output data type	string — `{'boolean'} | 'uint8'`
Logical Operator (`Logic`)		
`Operator`	Operator	string — `{'AND'} | 'OR' | 'NAND' | 'NOR' | 'XOR' | 'NXOR' | 'NOT'`
`Inputs`	Number of input ports	string — `{'2'}`
`IconShape`	Icon shape	string — `{'rectangular'} | 'distinctive'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`AllPortsSameDT`	Require all inputs and output to have the same data type	string — `{'off'} | 'on'`
`OutDataTypeStr`	Output data type	string — `'Inherit: Logical (see Configuration Parameters: Optimization)' | {'boolean'} | 'fixdt(1,16)'`
Relational Operator (`RelationalOperator`)		
`Operator`	Relational operator	string — `'==' | '~=' | '<' | {'<='} | '>=' | '>' | 'isInf' | 'isNaN' | 'isFinite'`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all inputs to have the same data type	string — `{'off'} | 'on'`

Logic and Bit Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `'Inherit: Logical (see Configuration Parameters: Optimization)'` | `{'boolean'}` | `'fixdt(1,16)'`
Shift Arithmetic (`ArithShift`)		
`BitShiftNumberSource`	Bits to shift > Source	string — `{'Dialog'}` | `'Input port'`
`BitShiftDirection`	Bits to shift > Direction	string — `'Left'` | `'Right'` | `{'Bidirectional'}`
`BitShiftNumber`	Bits to shift > Number	string — `{'8'}`
`BinPtShiftNumber`	Binary points to shift > Number	string — `{'0'}`
`DiagnosticForOORShift`	Diagnostic for out-of-range shift value	string — `{'None'}` | `'Warning'` | `'Error'`
`CheckOORBitShift`	Check for out-of-range 'Bits to shift' in generated code	string — `{'off'}` | `'on'`
`nBitShiftRight`	Deprecated in R2011a	
`nBinPtShiftRight`	Deprecated in R2011a	

Lookup Tables Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Cosine (`Cosine`) (masked subsystem)		
`Formula`	Output formula	string — `'sin(2*pi*u)'` | `{'cos(2*pi*u)'}` | `'exp(j*2*pi*u)'` | `'sin(2*pi*u) and cos(2*pi*u)'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`NumDataPoints`	Number of data points for lookup table	string — `{'(2^5)+1'}`
`OutputWordLength`	Output word length	string — `{'16'}`
Direct Lookup Table (n-D) (`LookupNDDirect`)		
`NumberOfTableDimensions`	Number of table dimensions	string — `'1' | {'2'} | '3' | '4'`
`InputsSelectThisObjectFromTable`	Inputs select this object from table	string — `{'Element'} | 'Column' | '2-D Matrix'`
`TableIsInput`	Make table an input	string — `{'off'} | 'on'`
`Table`	Table data	string — `{'[4 5 6;16 19 20;10 18 23]'}`
`DiagnosticForOutOfRangeInput`	Diagnostic for out-of-range input	string — `'None' | {'Warning'} | 'Error'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`TableMin`	Table minimum	string — `{'[]'}`
`TableMax`	Table maximum	string — `{'[]'}`
`TableDataTypeStr`	Table data type	string — `{'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 data type settings against changes by the fixed-point tools	string — `{'off'} | 'on'`
`maskTabDims`	Deprecated in R2009b	
`explicitNumDims`	Deprecated in R2009b	

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`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	string — `'1' | {'2'} | '3' | '4'`
`Table`	Table data > Value	string — `{'sqrt([1:11]' * [1:11])'}`
`TableSource`	Table data > Source	string — `{'Dialog'} | 'Input port'`
`InterpMethod`	Interpolation method	string — `'Flat' | {'Linear'}`
`ExtrapMethod`	Extrapolation method	string — `'Clip' | {'Linear'}`
`ValidIndexMayReachLast`	Valid index input may reach last index	string — `{'off'} | 'on'`
`DiagnosticForOutOfRangeInput`	Diagnostic for out-of-range input	string — `{'None'} | 'Warning' | 'Error'`
`RemoveProtectionIndex`	Remove protection against out-of-range index in generated code	string — `{'off'} | 'on'`
`NumSelectionDims`	Number of sub-table selection dimensions	string — `{'0'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`TableDataTypeStr`	Table data > Data Type	string — `'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	string — `{'[]'}`
`TableMax`	Table data > Maximum	string — `{'[]'}`
`IntermediateResultsDataTypeStr`	Intermediate results > Data Type	string — `{'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	string — `'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	string — `{'[]'}`
`OutMax`	Output > Maximum	string — `{'[]'}`
`LockScale`	Lock data type settings against changes by the fixed-point tools	string — `{'off'} | 'on'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'}` | `'on'`
`CheckIndexInCode`	Deprecated in R2011a	
1-D Lookup Table (`Lookup_n-D`)		
`NumberOfTableDimensions`	Number of table dimensions	string — `{'1'}` | `'2'` | `'3'` | `'4'`
`Table`	Table data	string — `{'tanh([-5:5])'}`
`BreakpointsForDimension1`	Breakpoints 1	string — `{'[-5:5]'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InterpMethod`	Interpolation method	string — `'Flat'` | `{'Linear'}` | `'Cubic spline'`
`ExtrapMethod`	Extrapolation method	string — `'Clip'` | `{'Linear'}` | `'Cubic spline'`
`UseLastTableValue`	Use last table value for inputs at or above last breakpoint	string — `{'off'}` | `'on'`
`IndexSearchMethod`	Index search method	string — `'Evenly spaced points'` | `'Linear search'` | `{'Binary search'}`
`BeginIndexSearchUsingPreviousIndexResult`	Begin index search using previous index result	string — `{'off'}` | `'on'`
`UseOneInputPortForAllInputData`	Use one input port for all input data	string — `{'off'}` | `'on'`
`DiagnosticForOutOfRangeInput`	Diagnostic for out-of-range input	string — `{'None'}` | `'Warning'` | `'Error'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RemoveProtectionInput`	Remove protection against out-of-range input in generated code	string — `{'off'} | 'on'`
`TableDataTypeStr`	Table data > Data Type	string — `'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	string — `{'[]'}`
`TableMax`	Table data > Maximum	string — `{'[]'}`
`BreakpointsForDimension1DataTypeStr`	Breakpoints 1 > Data Type	string — `{'Inherit:  Same as corresponding input'} | 'Inherit:  Inherit from 'Breakpoint data'' | 'double' | 'single' |`
`BreakpointsForDimension1Min`	Breakpoints 1 > Minimum	string — `{'[]'}`
`BreakpointsForDimension1Max`	Breakpoints 1 > Maximum	string — `{'[]'}`
`FractionDataTypeStr`	Fraction > Data Type	string — `{'Inherit:  Inherit via internal rule'} | 'double' | 'single' | 'fixdt(1,16,0)'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IntermediateResults DataTypeStr`	Intermediate results > Data Type	string — `'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	string — `'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	string — `{'[]'}`
`OutMax`	Output > Maximum	string — `{'[]'}`
`InputSameDT`	Require all inputs to have the same data type	string — `'off'` | `{'on'}`
`LockScale`	Lock data type settings against changes by the fixed-point tools	string — `{'off'}` | `'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `{'Simplest'}` | `'Zero'`
`SaturateOnInteger Overflow`	Saturate on integer overflow	string — `{'off'}` | `'on'`
2-D Lookup Table (`Lookup_n-D`)		
`NumberOfTableDimensions`	Number of table dimensions	string — `'1'` | `{'2'}` | `'3'` | `'4'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`Table`	Table data	string — `{'[4 5 6;16 19 20;10 18 23]'}`
`BreakpointsForDimension1`	Breakpoints 1	string — `{'[1:3]'}`
`BreakpointsForDimension2`	Breakpoints 2	string — `{'[1:3]'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InterpMethod`	Interpolation method	string — `'Flat' | {'Linear'} | 'Cubic spline'`
`ExtrapMethod`	Extrapolation method	string — `'Clip' | {'Linear'} | 'Cubic spline'`
`UseLastTableValue`	Use last table value for inputs at or above last breakpoint	string — `{'off'} | 'on'`
`IndexSearchMethod`	Index search method	string — `'Evenly spaced points' | 'Linear search' | {'Binary search'}`
`BeginIndexSearchUsingPreviousIndexResult`	Begin index search using previous index result	string — `{'off'} | 'on'`
`UseOneInputPortForAllInputData`	Use one input port for all input data	string — `{'off'} | 'on'`
`DiagnosticForOutOfRangeInput`	Diagnostic for out-of-range input	string — `{'None'} | 'Warning' | 'Error'`
`RemoveProtectionInput`	Remove protection against out-of-range input in generated code	string — `{'off'} | 'on'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
TableDataTypeStr	Table data > Data Type	string — '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	string — {'[]'}
TableMax	Table data > Maximum	string — {'[]'}
BreakpointsForDimension1DataTypeStr	Breakpoints 1 > Data Type	string — {'Inherit: Same as corresponding input'} | 'Inherit: Inherit from 'Breakpoint data'' | 'double' | 'single' |
BreakpointsForDimension1Min	Breakpoints 1 > Minimum	string — {'[]'}
BreakpointsForDimension1Max	Breakpoints 1 > Maximum	string — {'[]'}
BreakpointsForDimension2DataTypeStr	Breakpoints 2 > Data Type	string — {'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)'
BreakpointsForDimension2Min	Breakpoints 2 > Minimum	string — {'[]'}

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BreakpointsForDimension2Max`	Breakpoints 2 > Maximum	string — `{'[]'}`
`FractionDataTypeStr`	Fraction > Data Type	string — `{'Inherit:  Inherit via internal rule'} | 'double' | 'single' | 'fixdt(1,16,0)'`
`IntermediateResultsDataTypeStr`	Intermediate results > Data Type	string — `'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	string — `'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	string — `{'[]'}`
`OutMax`	Output > Maximum	string — `{'[]'}`
`InputSameDT`	Require all inputs to have the same data type	string — `'off' | {'on'}`
`LockScale`	Lock data type settings against changes by the fixed-point tools	string — `{'off'} | 'on'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | {'Simplest'} | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
n-D Lookup Table (`Lookup_n-D`)		
`NumberOfTableDimensions`	Number of table dimensions	string — `'1' | '2' | {'3'} | '4'`
`Table`	Table data	string — `{'reshape(repmat([4 5 6;16 19 20;10 18 23],1,2),[3,3,2])'}`
`BreakpointsForDimension1`	Breakpoints 1	string — `{'[10,22,31]'}`
`BreakpointsForDimension2`	Breakpoints 2	string — `{'[10,22,31]'}`
`BreakpointsForDimension3`	Breakpoints 3	string — `{'[5, 7]'}`
`...`	...	...
`BreakpointsForDimension30`	Breakpoints 30	string — `{'[1:3]'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InterpMethod`	Interpolation method	string — `'Flat' | {'Linear'} | 'Cubic spline'`
`ExtrapMethod`	Extrapolation method	string — `'Clip' | {'Linear'} | 'Cubic spline'`
`UseLastTableValue`	Use last table value for inputs at or above last breakpoint	string — `{'off'} | 'on'`
`DiagnosticForOutOfRangeInput`	Diagnostic for out-of-range input	string — `{'None'} | 'Warning' | 'Error'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
RemoveProtectionInput	Remove protection against out-of-range input in generated code	string — {'off'} | 'on'
IndexSearchMethod	Index search method	string — 'Evenly spaced points' | 'Linear search' | {'Binary search'}
BeginIndexSearchUsingPreviousIndexResult	Begin index search using previous index result	string — {'off'} | 'on'
UseOneInputPortForAllInputData	Use one input port for all input data	string — {'off'} | 'on'
SupportTunableTableSize	Support tunable table size in code generation	string — {'off'} | 'on'
MaximumIndicesForEachDimension	Maximum indices for each dimension	string — {'[]'}
TableDataTypeStr	Table data > Data Type	string — '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	string — {'[]'}
TableMax	Table data > Maximum	string — {'[]'}
BreakpointsForDimension1DataTypeStr	Breakpoints 1 > Data Type	string — {'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)'

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BreakpointsForDimension1 Min`	Breakpoints 1 > Minimum	string — {'[]'}
`BreakpointsForDimension1 Max`	Breakpoints 1 > Maximum	string — {'[]'}
`BreakpointsForDimension2 DataTypeStr`	Breakpoints 2 > Data Type	string — {'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	string — {'[]'}
`BreakpointsForDimension2 Max`	Breakpoints 2 > Maximum	string — {'[]'}
...	...	...
`BreakpointsForDimension30 DataTypeStr`	Breakpoints 30 > Data Type	string — {'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	string — {'[]'}

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BreakpointsForDimension30Max`	Breakpoints 30 > Maximum	string — `{'[]'}`
`FractionDataTypeStr`	Fraction > Data Type	string — `{'Inherit:  Inherit via internal rule'} | 'double' | 'single' | 'fixdt(1,16,0)'`
`IntermediateResultsDataTypeStr`	Intermediate results > Data Type	string — `'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	string — `'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	string — `{'[]'}`
`OutMax`	Output > Maximum	string — `{'[]'}`
`InputSameDT`	Require all inputs to have the same data type	string — `'off' | {'on'}`
`LockScale`	Lock data type settings against changes by the fixed-point tools	string — `{'off'} | 'on'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | {'Simplest'} | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`ProcessOutOfRangeInput`	Deprecated in R2009b	
Lookup Table Dynamic (`Lookup Table Dynamic`) (masked subsystem)		
`LookUpMeth`	Lookup Method	string — `'Interpolation-Extrapolation' | {'Interpolation-Use End Values'} | 'Use Input Nearest' | 'Use Input Below' | 'Use Input Above'`
`OutDataTypeStr`	Output data type	string — `{'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)'`
`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	string — `{'off'} | 'on'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round'| 'Simplest' | 'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`
Prelookup (`PreLookup`)		
`BreakpointsData`	Breakpoint data > Value	string — `{'[10:10:110]'}`
`BreakpointsDataSource`	Breakpoint data > Source	string — `{'Dialog'} | 'Input port'`
`IndexSearchMethod`	Index search method	string — `'Evenly spaced points' | 'Linear search' | {'Binary search'}`
`BeginIndexSearchUsingPreviousIndexResult`	Begin index search using previous index result	string — `{'off'} | 'on'`
`OutputOnlyTheIndex`	Output only the index	string — `{'off'} | 'on'`
`ExtrapMethod`	Extrapolation method	string — `'Clip' | {'Linear'}`
`UseLastBreakpoint`	Use last breakpoint for input at or above upper limit	string — `{'off'} | 'on'`
`DiagnosticForOutOfRangeInput`	Diagnostic for out-of-range input	string — `{'None'} | 'Warning' | 'Error'`
`RemoveProtectionInput`	Remove protection against out-of-range input in generated code	string — `{'off'} | 'on'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BreakpointDataTypeStr`	Breakpoint > Data Type	string — `{'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	string — `{'[]'}`
`BreakpointMax`	Breakpoint > Maximum	string — `{'[]'}`
`IndexDataTypeStr`	Index > Data Type	string — `'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | {'uint32'} | 'fixdt(1,16)'`
`FractionDataTypeStr`	Fraction > Data Type	string — `{'Inherit: Inherit via internal rule'} | 'double' | 'single' | 'fixdt(1,16,0)'`
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`ProcessOutOfRangeInput`	Deprecated in R2011a	
Sine (`Sine`) (masked subsystem)		
`Formula`	Output formula	string — `{'sin(2*pi*u)'} | 'cos(2*pi*u)' | 'exp(j*2*pi*u)' | 'sin(2*pi*u) and cos(2*pi*u)'`

Lookup Tables Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`NumDataPoints`	Number of data points for lookup table	string — `{'(2^5)+1'}`
`OutputWordLength`	Output word length	string — `{'16'}`

Math Operations Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Abs (`Abs`)		
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Add (`Sum`)		
`IconShape`	Icon shape	string — `{'rectangular'} | 'round'`
`Inputs`	List of signs	string — `{'++'}`
`CollapseMode`	Sum over	string — `{'All dimensions'} | 'Specified dimension'`
`CollapseDim`	Dimension	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all inputs to have the same data type	string — `{'off'} | 'on'`
`AccumDataTypeStr`	Accumulator data type	string — `{'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	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Algebraic Constraint (`Algebraic Constraint`) (masked subsystem)		
`z0`	Initial guess	string — `{'0'}`
Assignment (`Assignment`)		
`NumberOfDimensions`	Number of output dimensions	string — `{'1'}`
`IndexMode`	Index mode	string — `'Zero-based' | {'One-based'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutputInitialize`	Initialize output (Y)	string — `{'Initialize using input port <Y0>'} | 'Specify size for each dimension in table'`
`IndexOptionArray`	Index Option	string — `'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	string — `'Error' | 'Warning' | {'None'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`IndexOptions`	See `IndexOptionArray` parameter for more information.	
`Indices`	See `IndexParamArray` parameter for more information.	
`OutputSizes`	See `OutputSizeArray` parameter for more information.	
Bias (`Bias`)		
`Bias`	Bias	string — `{'0.0'}`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Complex to Magnitude-Angle (`ComplexToMagnitudeAngle`)		

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`Output`	Output	string — `'Magnitude' | 'Angle' | {'Magnitude and angle'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Complex to Real-Imag (`ComplexToRealImag`)		
`Output`	Output	string — `'Real' | 'Imag' | {'Real and imag'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Divide (`Product`)		
`Inputs`	Number of inputs	string — `{'*/'}`
`Multiplication`	Multiplication	string — `{'Element-wise(.*)'} | 'Matrix(*)'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all inputs to have same data type	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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)'`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Dot Product (`DotProduct`)		
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all inputs to have same data type	string — `'off' | {'on'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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)'`
`OutputDataTypeScalingMode`	Deprecated in R2007b	
`OutDataType`	Deprecated in R2007b	
`OutScaling`	Deprecated in R2007b	

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Find (`Find`)		
`IndexOutputFormat`	Index output format	string — `{'Linear indices'} | 'Subscripts'`
`NumberOfInputDimensions`	Number of input dimensions	integer — `{'1'}`
`IndexMode`	Index mode	string — `{'Zero-based'} | 'One-based'`
`ShowOutputForNonzeroInputValues`	Show output port for nonzero input values	string — `{'off'} | 'on'`
`SampleTime`	Sample time (–1 for inherited)	string — `{'-1'}`
`OutDataTypeStr`	Output data type	string — `{'Inherit: Inherit via internal rule'} | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)'`
Gain (`Gain`)		
`Gain`	Gain	string — `{'1'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`Multiplication`	Multiplication	string — `{'Element-wise(K.*u)'}` | `'Matrix(K*u)'` | `'Matrix(u*K)'` | `'Matrix(K*u) (u vector)'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'}` | `'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'}` | `'on'`
`ParamMin`	Parameter minimum	string — `{'[]'}`
`ParamMax`	Parameter maximum	string — `{'[]'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ParamDataTypeStr`	Parameter data type	string — `{'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	string — `'Magnitude' | 'Angle' | {'Magnitude and angle'}`
`ConstantPart`	Magnitude or Angle	string — `{'0'}`
`ApproximationMethod`	Approximation method	string — `{'None'} | 'CORDIC'`
`NumberOfIterations`	Number of iterations	string — `{'11'}`
`ScaleReciprocalGainFactor`	Scale output by reciprocal of gain factor	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Math Function (`Math`)		
`Operator`	Function	string — `{'exp'} | 'log' | '10^u' | 'log10' | 'magnitude^2' | 'square' | 'pow' | 'conj' | 'reciprocal' | 'hypot' | 'rem' | 'mod' | 'transpose' | 'hermitian'`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutputSignalType`	Output signal type	string — `{'auto'} | 'real' | 'complex'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `'off' | {'on'}`
Matrix Concatenate (`Concatenate`)		
`NumInputs`	Number of inputs	string — `{'2'}`
`Mode`	Mode	string — `'Vector' | {'Multidimensional array'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ConcatenateDimension`	Concatenate dimension	string — `{'2'}`
MinMax (`MinMax`)		
`Function`	Function	string — `{'min'} | 'max'`
`Inputs`	Number of input ports	string — `{'1'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all inputs to have the same data type	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
MinMax Running Resettable (`MinMax Running Resettable`) (masked subsystem)		
`Function`	Function	string — `{'min'} | 'max'`
`vinit`	Initial condition	string — `{'0.0'}`
Permute Dimensions (`PermuteDimensions`)		
`Order`	Order	string — `{'[2,1]'}`
Polynomial (`Polyval`)		
`coefs`	Polynomial Coefficients	string — `{'[ +2.081618890e-019, -1.441693666e-014, +4.719686976e-010, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001 ]'}`
Product (`Product`)		
`Inputs`	Number of inputs	string — `{'2'}`
`Multiplication`	Multiplication	string — `{'Element-wise(.*)'} | 'Matrix(*)'`
`CollapseMode`	Multiply over	string — `{'All dimensions'} | 'Specified dimension'`
`CollapseDim`	Dimension	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all inputs to have same data type	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | {'Zero'}`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Product of Elements (`Product`)		
`Inputs`	Number of inputs	string — `{'*'}`
`Multiplication`	Multiplication	string — `{'Element-wise(.*)'} | 'Matrix(*)'`
`CollapseMode`	Multiply over	string — `{'All dimensions'} | 'Specified dimension'`
`CollapseDim`	Dimension	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`InputSameDT`	Require all inputs to have same data type	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Real-Imag to Complex (`RealImagToComplex`)		
`Input`	Input	string — `'Real' | 'Imag' | {'Real and imag'}`
`ConstantPart`	Real part or Imag part	string — `{'0'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Reciprocal Sqrt (`Sqrt`)		

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`Operator`	Function	string — `'sqrt' | 'signedSqrt' | {'rSqrt'}`
`OutputSignalType`	Output signal type	string — `{'auto'} | 'real' | 'complex'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `'off' | {'on'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IntermediateResultsDataTypeStr`	Intermediate results data type	string — `{'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	string — `'Exact' | {'Newton-Raphson'}`
`Iterations`	Number of iterations	string — `{'3'}`
Reshape (`Reshape`)		
`OutputDimensionality`	Output dimensionality	string — `{'1-D array'} | 'Column vector (2-D)' | 'Row vector (2-D)' | 'Customize' | 'Derive from reference input port'`
`OutputDimensions`	Output dimensions	string — `{'[1,1]'}`
Rounding Function (`Rounding`)		
`Operator`	Function	string — `{'floor'} | 'ceil' | 'round' | 'fix'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Sign (`Signum`)		
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Signed Sqrt (`Sqrt`)		
`Operator`	Function	string — `'sqrt'` | `{'signedSqrt'}` | `'rSqrt'`
`OutputSignalType`	Output signal type	string — `{'auto'}` | `'real'` | `'complex'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `'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	string — `{'off'}` | `'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `'off'` | `{'on'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IntermediateResults DataTypeStr`	Intermediate results data type	string — `{'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	string — `{'Exact'} | 'Newton-Raphson'`
`Iterations`	Number of iterations	string — `{'3'}`
Sine Wave Function (`Sin`)		
`SineType`	Sine type	string — `{'Time based'} | 'Sample based'`
`TimeSource`	Time	string — `'Use simulation time' | {'Use external signal'}`
`Amplitude`	Amplitude	string — `{'1'}`
`Bias`	Bias	string — `{'0'}`
`Frequency`	Frequency	string — `{'1'}`
`Phase`	Phase	string — `{'0'}`
`Samples`	Samples per period	string — `{'10'}`
`Offset`	Number of offset samples	string — `{'0'}`
`SampleTime`	Sample time	string — `{'0'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Slider Gain (`Slider Gain`) (masked subsystem)		
`low`	Low	string — {`'0'`}
`gain`	Gain	string — {`'1'`}
`high`	High	string — {`'2'`}
Sqrt (`Sqrt`)		
`Operator`	Function	string — {`'sqrt'`} | `'signedSqrt'` | `'rSqrt'`
`OutputSignalType`	Output signal type	string — {`'auto'`} | `'real'` | `'complex'`
`SampleTime`	Sample time (-1 for inherited)	string — {`'-1'`}
`OutMin`	Output minimum	string — {`'[]'`}
`OutMax`	Output maximum	string — {`'[]'`}
`OutDataTypeStr`	Output data type	string — `'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	string — {`'off'`} | `'on'`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `'off' | {'on'}`
`IntermediateResultsDataTypeStr`	Intermediate results data type	string — `{'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	string — `{'Exact'} | 'Newton-Raphson'`
`Iterations`	Number of iterations	string — `{'3'}`
Squeeze (`Squeeze`) (masked subsystem)		
None	None	None
Subtract (`Sum`)		
`IconShape`	Icon shape	string — `{'rectangular'} | 'round'`
`Inputs`	List of signs	string — `{'+-'}`
`CollapseMode`	Sum over	string — `{'All dimensions'} | 'Specified dimension'`
`CollapseDim`	Dimension	string — `{'1'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time (-1 for inherited)	string — {`'-1'`}
`InputSameDT`	Require all inputs to have the same data type	string — {`'off'`} | `'on'`
`AccumDataTypeStr`	Accumulator data type	string — {`'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	string — {`'[]'`}
`OutMax`	Output maximum	string — {`'[]'`}
`OutDataTypeStr`	Output data type	string — {`'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	string — {`'off'`} | `'on'`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'}` | `'on'`
Sum (`Sum`)		
`IconShape`	Icon shape	string — `'rectangular'` | `{'round'}`
`Inputs`	List of signs	string — `{'|++'}`
`CollapseMode`	Sum over	string — `{'All dimensions'}` | `'Specified dimension'`
`CollapseDim`	Dimension	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all inputs to have the same data type	string — `{'off'}` | `'on'`
`AccumDataTypeStr`	Accumulator data type	string — `{'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	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Sum of Elements (`Sum`)		
`IconShape`	Icon shape	string — `{'rectangular'} | 'round'`
`Inputs`	List of signs	string — `{'+'}`
`CollapseMode`	Sum over	string — `{'All dimensions'} | 'Specified dimension'`
`CollapseDim`	Dimension	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`InputSameDT`	Require all inputs to have the same data type	string — `{'off'} | 'on'`
`AccumDataTypeStr`	Accumulator data type	string — `{'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	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'}` | `'on'`
Trigonometric Function (`Trigonometry`)		
`Operator`	Function	string — `{'sin'}` | `'cos'` | `'tan'` | `'asin'` | `'acos'` | `'atan'` | `'atan2'` | `'sinh'` | `'cosh'` | `'tanh'` | `'asinh'` | `'acosh'` | `'atanh'` | `'sincos'` | `'cos + jsin'`
`ApproximationMethod`	Approximation method	string — `{'None'}` | `'CORDIC'`
`NumberOfIterations`	Number of iterations	string — `{'11'}`
`OutputSignalType`	Output signal type	string — `{'auto'}` | `'real'` | `'complex'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Unary Minus (`UnaryMinus`)		
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'}` | `'on'`
Vector Concatenate (`Concatenate`)		
`NumInputs`	Number of inputs	string — `{'2'}`
`Mode`	Mode	string — `{'Vector'}` | `'Multidimensional array'`
Weighted Sample Time Math (`SampleTimeMath`)		

Math Operations Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`TsampMathOp`	Operation	string — `{'+'} | '-' | '*' | '/' | 'Ts Only' | '1/Ts Only'`
`weightValue`	Weight value	string — `{'1.0'}`
`TsampMathImp`	Implement using	string — `{'Online Calculations'} | 'Offline Scaling Adjustment'`
`OutDataTypeStr`	Output data type	string — `{'Inherit via internal rule'} | 'Inherit via back propagation'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'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	string — `'off' | {'on'}`
`AssertionFailFcn`	Simulation callback when assertion fails	string — `{''}`
`StopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`

Model Verification Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Check Dynamic Gap (`Checks_DGap`) (masked subsystem)		
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Dynamic Range (`Checks_DRange`) (masked subsystem)		
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Static Gap (`Checks_SGap`) (masked subsystem)		
`max`	Upper bound	string — `{'100'}`
`max_included`	Inclusive upper bound	string — `'off' | {'on'}`
`min`	Lower bound	string — `{'0'}`
`min_included`	Inclusive lower bound	string — `'off' | {'on'}`
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`

Model Verification Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Static Range (`Checks_SRange`) (masked subsystem)		
`max`	Upper bound	string — `{'100'}`
`max_included`	Inclusive upper bound	string — `'off' | {'on'}`
`min`	Lower bound	string — `{'0'}`
`min_included`	Inclusive lower bound	string — `'off' | {'on'}`
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Discrete Gradient (`Checks_Gradient`) (masked subsystem)		
`gradient`	Maximum gradient	string — `{'1'}`
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`

Model Verification Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Dynamic Lower Bound (`Checks_DMin`) (masked subsystem)		
`Enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Dynamic Upper Bound (`Checks_DMax`) (masked subsystem)		
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Input Resolution (`Checks_Resolution`) (masked subsystem)		
`resolution`	Resolution	string — `{'1'}`
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`

Model Verification Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Check Static Lower Bound (`Checks_SMin`) (masked subsystem)		
`min`	Lower bound	string — `{'0'}`
`min_included`	Inclusive boundary	string — `'off' | {'on'}`
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'graphic'} | 'text'`
Check Static Upper Bound (`Checks_SMax`) (masked subsystem)		
`max`	Upper bound	string — `{'0'}`
`max_included`	Inclusive boundary	string — `'off' | {'on'}`
`enabled`	Enable assertion	string — `'off' | {'on'}`
`callback`	Simulation callback when assertion fails (optional)	string — `{''}`
`stopWhenAssertionFail`	Stop simulation when assertion fails	string — `'off' | {'on'}`
`export`	Output assertion signal	string — `{'off'} | 'on'`
`icon`	Select icon type	string — `{'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)		

Model-Wide Utilities Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ECoderFlag`	Embedded Coder Flag	string — `{''}`
`DocumentType`	Document Type	string — `{'Text'} | 'RTF' | 'HTML'`
Model Info (`CMBlock`) (masked subsystem)		
`InitialSaveTempField`	InitialSaveTempField	string — `{''}`
`InitialBlockCM`	InitialBlockCM	string — `{'None'}`
`BlockCM`	BlockCM	string — `{'None'}`
`Frame`	Show block frame	string — `'off' | {'on'}`
`SaveTempField`	SaveTempField	string — `{''}`
`DisplayStringWithTags`	DisplayStringWithTags	string — `{'Model Info'}`
`MaskDisplayString`	MaskDisplayString	string — `{'Model Info'}`
`HorizontalTextAlignment`	Horizontal text alignment	string — `{'Center'}`
`LeftAlignmentValue`	LeftAlignmentValue	string — `{'0.5'}`
`SourceBlockDiagram`	SourceBlockDiagram	string — `{'untitled'}`
`TagMaxNumber`	TagMaxNumber	string — `{'20'}`
`CMTag1`	CMTag1	string — `{''}`
`CMTag2`	CMTag2	string — `{''}`
`CMTag3`	CMTag3	string — `{''}`
`CMTag4`	CMTag4	string — `{''}`
`CMTag5`	CMTag5	string — `{''}`
`CMTag6`	CMTag6	string — `{''}`
`CMTag7`	CMTag7	string — `{''}`
`CMTag8`	CMTag8	string — `{''}`
`CMTag9`	CMTag9	string — `{''}`
`CMTag10`	CMTag10	string — `{''}`

Model-Wide Utilities Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`CMTag11`	CMTag11	string — `{''}`
`CMTag12`	CMTag12	string — `{''}`
`CMTag13`	CMTag13	string — `{''}`
`CMTag14`	CMTag14	string — `{''}`
`CMTag15`	CMTag15	string — `{''}`
`CMTag16`	CMTag16	string — `{''}`
`CMTag17`	CMTag17	string — `{''}`
`CMTag18`	CMTag18	string — `{''}`
`CMTag19`	CMTag19	string — `{''}`
`CMTag20`	CMTag20	string — `{''}`
Timed-Based Linearization (`Timed Linearization`) (masked subsystem)		
`LinearizationTime`	Linearization time	string — `{'1'}`
`SampleTime`	Sample time (of linearized model)	string — `{'0'}`
Trigger-Based Linearization (`Triggered Linearization`) (masked subsystem)		
`TriggerType`	Trigger type	string — `{'rising'} | 'falling' | 'either' | 'function-call'`
`SampleTime`	Sample time (of linearized model)	string — `{'0'}`

Ports & Subsystems Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Action Port (`ActionPort`)		
`InitializeStates`	States when execution is resumed	string — `{'held'} | 'reset'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`PropagateVarSize`	Propagate sizes of variable-size signals	string — `{'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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`FunctionWithSeparateData`	"Function with separate data" on page 1-1656 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	string — `{'off'} | 'on'`
`RTWMemSecFuncInitTerm`	"Memory section for initialize/terminate functions" on page 1-1659 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	string — `{'Inherit from model'} | 'Default' |` list of memory sections from model's package

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWMemSecFuncExecute`	"Memory section for execution functions" on page 1-1661 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	string — `{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`RTWMemSecDataConstants`	"Memory section for constants" on page 1-1663 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	string — `{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`RTWMemSecDataInternal`	"Memory section for internal data" on page 1-1664 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	string — `{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`RTWMemSecDataParameters`	"Memory section for parameters" on page 1-1665 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	string — `{'Inherit from model'}` | `'Default'` | list of memory sections from model's package
`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.	string — `{'UseLocalSettings'}` | `'ScaledDouble'` | `'Double'` | `'Single'` | `'Off'`

Ports & Subsystems Library Block Parameters (Continued)

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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`
`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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`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.	string — `{'UseLocalSettings'} | 'ScaledDouble' | 'Double' | 'Single' | 'Off'`

Ports & Subsystems Library Block Parameters (Continued)

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.	string — `{'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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{'self'}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`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.	string — `{'UseLocalSettings'} | 'ScaledDouble' | 'Double' | 'Single' | 'Off'`

Ports & Subsystems Library Block Parameters (Continued)

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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`
`IsSubsystemVirtual`		boolean — `{'on'} | 'off'` Read-only
Enable (`EnablePort`)		
`StatesWhenEnabling`	States when enabling	string — `{'held'} | 'reset'`
`PropagateVarSize`	Propagate sizes of variable-size signals	string — `{'Only when enabling'} | 'During execution'`
`ShowOutputPort`	Show output port	string — `{'off'} | 'on'`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
Enabled and Triggered Subsystem (`SubSystem`)		
`ShowPortLabels`	Show port labels **Note** The values `'off'` and `'on'` are for backward compatibility only and should not be used in new models or when updating existing models.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`

Ports & Subsystems Library Block Parameters (Continued)

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.	string — `{'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.	string — `{'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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`

Ports & Subsystems Library Block Parameters (Continued)

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.	string — `{'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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'` Read-only
For Each(`ForEach`)		
`InputPartition`	Partition	cell array of strings
`InputPartitionDimension`	Partition dimension for input signal	cell array of strings
`InputPartitionWidth`	Width of partition for input signal	cell array of strings
`OutputConcatenationDimension`	Concatenation dimension of output signal	cell array of strings
For Iterator (`ForIterator`)		
`ResetStates`	States when starting	string — `{'held'} | 'reset'`
`IterationSource`	Iteration limit source	string — `{'internal'} | 'external'`
`IterationLimit`	Iteration limit	string — `{'5'}`
`ExternalIncrement`	Set next i (iteration variable) externally	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ShowIterationPort`	Show iteration variable	string — `'off' | {'on'}`
`IndexMode`	Index mode	string — `'Zero-based' | {'One-based'}`
`IterationVariableDataType`	Iteration variable data type	string — `{'int32'} | 'int16' | 'int8' | 'double'`
For 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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`PropExecContext OutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInp InsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`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.	string — `{'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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'` Read-only
Function-Call Generator (`Function-Call Generator`) (masked subsystem)		
`sample_time`	Sample time	string — `{'1'}`
`numberOfIterations`	Number of iterations	string — `{'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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`PropExecContext OutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInp InsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`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.	string — `{'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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'` Read-only
If (`If`)		
`NumInputs`	Number of inputs	string — `{'1'}`
`IfExpression`	If expression (e.g., u1 ~= 0)	string — `{'u1 > 0'}`
`ElseIfExpressions`	Elseif expressions (comma-separated list, e.g., u2 ~= 0, u3(2) < u2)	string — `{''}`
`ShowElse`	Show else condition	string — `'off' | {'on'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`

Ports & Subsystems Library Block Parameters (Continued)

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.	string — `{'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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'Off'`
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'` Read-only
In1 (`Inport`)		
`Port`	Port number	string — `{'1'}`
`IconDisplay`	Icon display	string — `'Signal name' | {'Port number'} | 'Port number and signal name'`
`LatchByDelayingOutsideSignal`	Latch input by delaying outside signal	string — `{'off'} | 'on'`
`LatchInputForFeedbackSignals`	Latch input for feedback signals of function-call subsystem outputs	string — `{'off'} | 'on'`
`Interpolate`	Interpolate data	string — `'off' | {'on'}`
`UseBusObject`	Specify properties via bus object	string — `{'off'} | 'on'`
`BusObject`	Bus object for specifying bus properties	string — `{'BusObject'}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BusOutputAsStruct`	Output as nonvirtual bus	string — `{'off'} | 'on'`
`PortDimensions`	Port dimensions (-1 for inherited)	string — `{'-1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Minimum	string — `{'[]'}`
`OutMax`	Maximum	string — `{'[]'}`
`OutDataTypeStr`	Data type	string — `{'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	string — `{'off'} | 'on'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
`SamplingMode`	Sampling mode	string — `{'auto'} | 'Sample based' | 'Frame based'`
Model (`ModelReference`)		

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`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`.	string — `{'<Enter Model Name>'}`
`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.	string — Set automatically when `ModelNameDialog` is set.

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`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.	string — 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`	Model arguments	string — `{''}`
`ParameterArgumentValues`	Model argument values (for this instance)	string — `{''}`
`SimulationMode`	Specifies whether to simulate the model by generating and executing code or by interpreting the model in Simulink software.	string — `{'Normal'} | 'Accelerator' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'`
`Variant`	Specifies whether the Model block references variant models.	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`Variants`	An array of `variant` structures where each element specifies one variant. The structure fields are as follows:	array — [ ]
	`variant.Name` – The name of the `Simulink.Variant` object that represents the variant to which this element applies.	string — `{''}`
	`variant.ModelName` – The name of the referenced model associated with the specified variant object in this Model block.	string — `{''}`
	`variant.ParameterArgumentNames` – Noneditable string containing the names of the model arguments for which the Model block must supply values.	string — `{''}`
	`variant.ParameterArgumentValues` – The values to supply for the model arguments when this variant is the active variant.	string — `{''}`
	`variant.SimulationMode` – The execution mode to use when this variant is the active variant.	string — `{'Accelerator'} | 'Normal' | 'Processor-in-the-loop (PIL)'`
`OverrideUsingVariant`	Whether to override the variant conditions and make a specified variant the active variant, and if so, the name of that variant.	string — `{''}` The value is the empty string if no overriding variant object is specified; or the name of the overriding object.

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ActiveVariant`	The variant that is currently active, either because its variant condition is `true` or `OverrideUsingVariant` has overridden the variant conditions and specified this variant.	string — `{''}` The value is the empty string if no variant is active; or the name of the active variant.
`GeneratePreprocessorConditionals`	Locally controls whether generated code contains preprocessor conditionals. This parameter applies only to Simulink Coder code generation and has no effect on the behavior of a model in Simulink. The parameter is available only for an ERT target when you select **Inline parameters**. For more information, see "Variant Systems".	string — `{'off'} | 'on'`
`AvailSigsInstanceProps`		handle vector — `{''}`
`AvailSigsDefaultProps`		handle vector — `{''}`
`DefaultDataLogging`		string — `{'off'} | 'on'`
Out1 (`Outport`)		
Port	Port number	string — `{'1'}`
`IconDisplay`	Icon display	string — `'Signal name' | {'Port number'} | 'Port number and signal name'`
`UseBusObject`	Specify properties via bus object	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BusObject`	Bus object for validating input bus	string — `{'BusObject'}`
`BusOutputAsStruct`	Output as nonvirtual bus in parent model	string — `{'off'} | 'on'`
`PortDimensions`	Port dimensions (-1 for inherited)	string — `{'-1'}`
`VarSizeSig`	Variable-size signal	string — `{'Inherit'} | 'No' | 'Yes'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Minimum	string — `{'[]'}`
`OutMax`	Maximum	string — `{'[]'}`
`OutDataTypeStr`	Data type	string — `{'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	string — `{'off'} | 'on'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
`SamplingMode`	Sampling mode	string — `{'auto'} | 'Sample based' | 'Frame based'`
`OutputWhenDisabled`	Output when disabled	string — `{'held'} | 'reset'`
`InitialOutput`	Initial output	string — `{'[]'}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
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.	string — `'none'` | `{'FromPortIcon'}` | `'FromPortBlockName'` | `'SignalName'` | `'off'` | `'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'}` | `'ReadOnly'` | `'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'}` | `'ExplicitOnly'` | `'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `{'off'}` | `'on'`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'}` | `'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'}` | `'on'`
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'}` | `'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`DataTypeOverride`	Specifies data type used to override fixed-point data types. Set by **Data type override** on the Fixed-Point Tool.	string — `{'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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`
`IsSubsystemVirtual`		boolean — `{'on'} | 'off'` Read-only
`Virtual`	For internal use	
Switch Case (`SwitchCase`)		
`CaseConditions`	Case conditions (e.g., {1,[2,3]})	string — `{'{1}'}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ShowDefaultCase`	Show default case	string — `'off' | {'on'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`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.	string — `{'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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'` Read-only
Trigger (`TriggerPort`)		
`TriggerType`	Trigger type	string — `{'rising'} | 'falling' | 'either' | 'function-call'`
`StatesWhenEnabling`	States when enabling	string — `{'held'} | 'reset' | 'inherit'`
`PropagateVarSize`	Propagate sizes of variable-size signals	string — `{'During execution'} | 'Only when enabling'`
`ShowOutputPort`	Show output port	string — `{'off'} | 'on'`
`OutputDataType`	Output data type	string — `{'auto'} | 'double' | 'int8'`
`SampleTimeType`	Sample time type	string — `{'triggered'} | 'periodic'`
`SampleTime`	Sample time	string — `{'1'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`PortDimensions`	Port dimensions (-1 for inherited)	string — `{'-1'}`
`TriggerSignalSampleTime`	Trigger signal sample time	string — `{'-1'}`
`OutMin`	Minimum	string — `{'[]'}`
`OutMax`	Maximum	string — `{'[]'}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Data type	string — `{'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	string — `'off' | {'on'}`
Triggered Subsystem (`SubSystem`)		
`ShowPortLabels`	Show port labels **Note** The values `'off'` and `'on'` are for backward compatibility only and should not be used in new models or when updating existing models.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContext OutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInp InsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`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.	string — `{'UseLocalSettings'} | 'ScaledDouble' | 'Double' | 'Single' | 'Off'`

Ports & Subsystems Library Block Parameters (Continued)

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.	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'` Read-only
While Iterator (`WhileIterator`)		
`MaxIters`	Maximum number of iterations (-1 for unlimited)	string — `{'5'}`
`WhileBlockType`	While loop type	string — `{'while'} | 'do-while'`
`ResetStates`	States when starting	string — `{'held'} | 'reset'`
`ShowIterationPort`	Show iteration number port	string — `{'off'} | 'on'`
`OutputDataType`	Output data type	string — `{'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.	string — `'none' | {'FromPortIcon'} | 'FromPortBlockName' | 'SignalName' | 'off' | 'on'`
`BlockChoice`	Block choice	string — `{''}`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchicalResolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`
`PropExecContextOutsideSubsystem`	Propagate execution context across subsystem boundary	string — `{'off'} | 'on'`
`CheckFcnCallInpInsideContextMsg`	Warn if function-call inputs are context-specific	string — `{'off'} | 'on'`
`SystemSampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`RTWSystemCode`	Code generation function packaging	string — `{'Auto'} | 'Inline' | 'Function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Code generation function name	string — `{''}`
`RTWFileNameOpts`	Code generation file name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`

Ports & Subsystems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWFileName`	Code generation file name (no extension)	string — `{''}`
`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.	string — `{'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.	string — `{'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	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`ConvertRealWorld`	Input and output to have equal	string — `{'Real World Value (RWV)'} | 'Stored Integer (SI)'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Data Type Conversion Inherited (`Conversion Inherited`) (masked subsystem)		
`ConvertRealWorld`	Input and Output to have equal	string — `{'Real World Value'} | 'Stored Integer'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`
Data Type Duplicate (`DataTypeDuplicate`)		
`NumInputPorts`	Number of input ports	string — `{'2'}`
Data Type Propagation (`Data Type Propagation`) (masked subsystem)		
`PropDataTypeMode`	1. Propagated data type	string — `'Specify via dialog' | {'Inherit via propagation rule'}`
`PropDataType`	1.1. Propagated data type (e.g., `fixdt(1,16)`, `fixdt('single')`)	string — `{'fixdt(1,16)'}`
`IfRefDouble`	1.1. If any reference input is double, output is	string — `{'double'} | 'single'`
`IfRefSingle`	1.2. If any reference input is single, output is	string — `'double' | {'single'}`
`IsSigned`	1.3. Is-Signed	string — `'IsSigned1' | 'IsSigned2' | {'IsSigned1 or IsSigned2'} | 'TRUE' | 'FALSE'`
`NumBitsBase`	1.4.1. Number-of-Bits: Base	string — `'NumBits1' | 'NumBits2' | {'max([NumBits1 NumBits2])'} | 'min([NumBits1 NumBits2])' | 'NumBits1+NumBits2'`
`NumBitsMult`	1.4.2. Number-of-Bits: Multiplicative adjustment	string — `{'1'}`
`NumBitsAdd`	1.4.3. Number-of-Bits: Additive adjustment	string — `{'0'}`

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`NumBitsAllowFinal`	1.4.4. Number-of-Bits: Allowable final values	string — `{'1:128'}`
`PropScalingMode`	2. Propagated scaling	string — `'Specify via dialog' | {'Inherit via propagation rule'} | 'Obtain via best precision'`
`PropScaling`	2.1. Propagated scaling: Slope or [Slope Bias] ex. 2^-9	string — `{'2^-10'}`
`ValuesUsedBestPrec`	2.1. Values used to determine best precision scaling	string — `{'[5 -7]'}`
`SlopeBase`	2.1.1. Slope: Base	string — `'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	string — `{'1'}`
`SlopeAdd`	2.1.3. Slope: Additive adjustment	string — `{'0'}`

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BiasBase`	2.2.1. Bias: Base	string — `{'Bias1'} | 'Bias2' | 'max([Bias1 Bias2])' | 'min([Bias1 Bias2])' | 'Bias1*Bias2' | 'Bias1/Bias2' | 'Bias1+Bias2' | 'Bias1-Bias2'`
`BiasMult`	2.2.2. Bias: Multiplicative adjustment	string — `{'1'}`
`BiasAdd`	2.2.3. Bias: Additive adjustment	string — `{'0'}`
Data Type Scaling Strip (`Scaling Strip`) (masked subsystem)		
IC (`InitialCondition`)		
`Value`	Initial value	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Probe (`Probe`)		
`ProbeWidth`	Probe width	string — `'off' | {'on'}`
`ProbeSampleTime`	Probe sample time	string — `'off' | {'on'}`
`ProbeComplexSignal`	Detect complex signal	string — `'off' | {'on'}`
`ProbeSignalDimensions`	Probe signal dimensions	string — `'off' | {'on'}`
`ProbeFramedSignal`	Detect framed signal	string — `'off' | {'on'}`
`ProbeWidthDataType`	Data type for width	string — `{'double'} | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'`

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ProbeSampleTimeDataType`	Data type for sample time	string — `{'double'} | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'`
`ProbeComplexityDataType`	Data type for signal complexity	string — `{'double'} | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'Same as input'`
`ProbeDimensionsDataType`	Data type for signal dimensions	string — `{'double'} | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'`
`ProbeFrameDataType`	Data type for signal frames	string — `{'double'} | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'Same as input'`
Rate Transition (`RateTransition`)		
`Integrity`	Ensure data integrity during data transfer	string — `'off' | {'on'}`
`Deterministic`	Ensure deterministic data transfer (maximum delay)	string — `'off' | {'on'}`
`X0`	Initial conditions	string — `{'0'}`

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutPortSampleTimeOpt`	Output port sample time options	string — `{'Specify'} | 'Inherit' | 'Multiple of input port sample time'`
`OutPortSampleTimeMultiple`	Sample time multiple (>0)	string — `{'1'}`
`OutPortSampleTime`	Output port sample time	string — `{'-1'}`
Signal Conversion (`SignalConversion`)		
`ConversionOutput`	Output	string — `{'Signal copy'} | 'Virtual bus' | 'Nonvirtual bus'`
`OutDataTypeStr`	Data type	string — `{'Inherit:  auto'} | 'Bus:  <object name>'`
`OverrideOpt`	Exclude this block from 'Block reduction' optimization	string — `{'off'} | 'on'`
Signal Specification (`SignalSpecification`)		
`Dimensions`	Dimensions (-1 for inherited)	string — `{'-1'}`
`VarSizeSig`	Variable-size signal	string — `{'Inherit'} | 'No' | 'Yes'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Minimum	string — `{'[]'}`
`OutMax`	Maximum	string — `{'[]'}`
`OutDataTypeStr`	Data type	string — `{'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>'`

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`BusOutputAsStruct`	Require nonvirtual bus	string — `{'off'} | 'on'`
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
`SamplingMode`	Sampling mode	string — `{'auto'} | 'Sample based' | 'Frame based'`
Weighted Sample Time (`SampleTimeMath`)		
`TsampMathOp`	Operation	string — `'+' | '-' | '*' | '/' | {'Ts Only'} | '1/Ts Only'`
`weightValue`	Weight value	string — `{'1.0'}`
`TsampMathImp`	Implement using	string — `{'Online Calculations'} | 'Offline Scaling Adjustment'`
`OutDataTypeStr`	Output data type	string — `{'Inherit via internal rule'} | 'Inherit via back propagation'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`OutputDataTypeScalingMode`	Deprecated in R2009b	
`DoSatur`	Deprecated in R2009b	
Width (`Width`)		

Signal Attributes Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutputDataTypeScalingMode`	Output data type mode	string — `{'Choose intrinsic data type'} | 'Inherit via back propagation' | 'All ports same datatype'`
`DataType`	Output data type	string — `{'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	string — `{''}`
`InputSignals`	Signals in the bus	matrix — `{'{}'}`
Bus Creator (`BusCreator`)		
`Inputs`	Number of inputs	string — `{'2'}`
`DisplayOption`		string — `'none' | 'signals' | {'bar'}`
`NonVirtualBus`	Output as nonvirtual bus	string — `{'off'} | 'on'`
`OutDataTypeStr`	Data type	string — `{'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>'`

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Bus Selector (`BusSelector`)		
`OutputSignals`	Selected signals	string — in the form `'signal1,signal2'`
`OutputAsBus`	Output as bus	string — `{'off'} | 'on'`
`InputSignals`	Signals in bus	matrix — `{'{}'}`
Data Store Memory (`DataStoreMemory`)		
`DataStoreName`	Data store name	string — `{'A'}`
`ReadBeforeWriteMsg`	Detect read before write	string — `'none' | {'warning'} | 'error'`
`WriteAfterWriteMsg`	Detect write after write	string — `'none' | {'warning'} | 'error'`
`WriteAfterReadMsg`	Detect write after read	string — `'none' | {'warning'} | 'error'`
`InitialValue`	Initial value	string — `{'0'}`
`StateMustResolveToSignalObject`	Data store name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`DataLogging`	Log Signal Data	string — `'off' | {'on'}`
`DataLoggingNameMode`	Logging Name	string — `{'SignalName'} | 'Custom'`
`DataLoggingName`	Logging Name	string — `{''}`
`DataLoggingLimitDataPoints`	Limit data points to last	string — `'off' | {'on'}`
`DataLoggingMaxPoints`	Limit data points to last	non-zero integer `{5000}`
`DataLoggingDecimateData`	Decimation	string — `'off' | {'on'}`
`DataLoggingLimitDataPoints`	Decimation	non-zero integer `{2}`

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWStateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageTypeQualifier`	Code generation type qualifier	string — `{''}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`
`ShowAdditionalParam`	Show additional parameters	string — `{'off'} | 'on'`
`OutMin`	Minimum	string — `{'[]'}`
`OutMax`	Maximum	string — `{'[]'}`
`OutDataTypeStr`	Data type	string — `{'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	string — `{'off'} | 'on'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
Data Store Read (`DataStoreRead`)		
`DataStoreName`	Data store name	string — `{'A'}`
`SampleTime`	Sample time	string — `{'0'}`
Data Store Write (`DataStoreWrite`)		
`DataStoreName`	Data store name	string — `{'A'}`

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Demux (`Demux`)		
`Outputs`	Number of outputs	string — `{'2'}`
`DisplayOption`	Display option	string — `'none' | {'bar'}`
`BusSelectionMode`	Bus selection mode	string — `{'off'} | 'on'`
Environment Controller (`Environment Controller`) (masked subsystem)		
From (`From`)		
`GotoTag`	Goto tag	string — `{'A'}`
`IconDisplay`	Icon display	string — `'Signal name' | {'Tag'} | 'Tag and signal name'`
Goto (`Goto`)		
`GotoTag`	Tag	string — `{'A'}`
`IconDisplay`	Icon display	string — `'Signal name' | {'Tag'} | 'Tag and signal name'`
`TagVisibility`	Tag visibility	string — `{'local'} | 'scoped' | 'global'`
Goto Tag Visibility (`GotoTagVisibility`)		
`GotoTag`	Goto tag	string — `{'A'}`
Index Vector (`MultiPortSwitch`)		
`DataPortOrder`	Data port order	string — `{'Zero-based contiguous'} | 'One-based contiguous' | 'Specify indices'`
`Inputs`	Number of data ports	string — `{'1'}`
`zeroidx`	Deprecated in R2010a	

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all data port inputs to have the same data type	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`AllowDiffInputSizes`	Allow different data input sizes (Results in variable-size output signal)	string — `{'off'} | 'on'`
Manual Switch (`Manual Switch`) (masked subsystem)		
`varsize`	Allow different input sizes (Results in variable-size output signal)	string — `{'off'} | 'on'`

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Merge (`Merge`)		
`Inputs`	Number of inputs	string — `{'2'}`
`InitialOutput`	Initial output	string — `{'[]'}`
`AllowUnequalInputPortWidths`	Allow unequal port widths	string — `{'off'} | 'on'`
`InputPortOffsets`	Input port offsets	string — `{'[]'}`
Multiport Switch (`MultiPortSwitch`)		
`DataPortOrder`	Data port order	string — `'Zero-based contiguous' | {'One-based contiguous'} | 'Specify indices'`
`Inputs`	Number of data ports	string — `{'3'}`
`zeroidx`	Deprecated in R2010a	
`DataPortIndices`	Data port indices	string — `{'{1,2,3}'}`
`DataPortForDefault`	Data port for default case	string — `{'Last data port'} | 'Additional data port'`
`DiagnosticForDefault`	Diagnostic for default case	string — `'None' | 'Warning' | {'Error'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InputSameDT`	Require all data port inputs to have the same data type	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Output data type	string — {`'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	string — {`'off'`} | `'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | {`'Floor'`} | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — {`'off'`} | `'on'`
`AllowDiffInputSizes`	Allow different data input sizes (Results in variable-size output signal)	string — {`'off'`} | `'on'`
Mux (`Mux`)		
`Inputs`	Number of inputs	string — {`'2'`}
`DisplayOption`	Display option	string — `'none'` | `'signals'` | {`'bar'`}
`UseBusObject`	For internal use	
`BusObject`	For internal use	
`NonVirtualBus`	For internal use	
Selector (`Selector`)		
`NumberOfDimensions`	Number of input dimensions	string — {`'1'`}

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IndexMode`	Index mode	string — `'Zero-based'` | `{'One-based'}`
`IndexOptionArray`	Index Option	string — `'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	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`IndexOptions`	See `IndexOptionArray` parameter for more information.	
`Indices`	See `IndexParamArray` parameter for more information.	
`OutputSizes`	See `OutputSizeArray` parameter for more information.	
Switch (`Switch`)		
`Criteria`	Criteria for passing first input	string — `{'u2 >= Threshold'}` | `'u2 > Threshold'` | `'u2 ~= 0'`
`Threshold`	Threshold	string — `{'0'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off'` | `{'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

Signal Routing Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`InputSameDT`	Require all data port inputs to have the same data type	string — `{'off'} | 'on'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`AllowDiffInputSizes`	Allow different input sizes (Results in variable-size output signal)	string — `{'off'} | 'on'`
Vector Concatenate (`Concatenate`)		
`NumInputs`	Number of inputs	string — `{'2'}`
`Mode`	Mode	string — `{'Vector'} | 'Multidimensional array'`

Sinks Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Display (`Display`)		
`Format`	Format	string — `{'short'}` | `'long'` | `'short_e'` | `'long_e'` | `'bank'` | `'hex (Stored Integer)'` | `'binary (Stored Integer)'` | `'decimal (Stored Integer)'` | `'octal (Stored Integer)'`
`Decimation`	Decimation	string — `{'1'}`
`Floating`	Floating display	string — `{'off'}` | `'on'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Floating Scope (`Scope`)		
`Floating`		string — `'off'` | `{'on'}`
`Location`		vector — `{'[376 294 700 533]'}`
`Open`		string — `{'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`		string — `'on'` | `'off'` | `{'OneTimeTick'}`
`ZoomMode`		string — `{'on'}` | `'xonly'` | `'yonly'`
`AxesTitles`		string
`Grid`		string — `'off'` | `{'on'}` | `'xonly'` | `'yonly'`
`TimeRange`		string — `{'auto'}`

Sinks Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`YMin`		string — `{'-5'}`
`YMax`		string — `{'5'}`
`SaveToWorkspace`		string — `{'off'} | 'on'`
`SaveName`		string — `{'ScopeData'}`
`DataFormat`		string — `{'StructureWithTime'} | 'Structure' | 'Array'`
`LimitDataPoints`		string — `'off' | {'on'}`
`MaxDataPoints`		string — `{'5000'}`
`Decimation`		string — `{'1'}`
`SampleInput`		string — `{'off'} | 'on'`
`SampleTime`		string — `{'0'}`
Out1 (`Outport`)		
Port	Port number	string — `{'1'}`
`IconDisplay`	Icon display	string — `'Signal name' | {'Port number'} | 'Port number and signal name'`
`BusOutputAsStruct`	Output as nonvirtual bus in parent model	string — `{'off'} | 'on'`
`PortDimensions`	Port dimensions (-1 for inherited)	string — `{'-1'}`
`VarSizeSig`	Variable-size signal	string — `{'Inherit'} | 'No' | 'Yes'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Minimum	string — `{'[]'}`
`OutMax`	Maximum	string — `{'[]'}`

Sinks Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`OutDataTypeStr`	Data type	string — `{'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	string — `{'off'} | 'on'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
`SamplingMode`	Sampling mode	string — `{'auto'} | 'Sample based' | 'Frame based'`
`OutputWhenDisabled`	Output when disabled	string — `{'held'} | 'reset'`
`InitialOutput`	Initial output	string — `{'[]'}`
Scope (`Scope`)		
`Floating`		string — `{'off'} | 'on'`
Location		vector — `{'[188 390 512 629]'}`
`Open`		string — `{'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`		string — `'on' | 'off' | {'OneTimeTick'}`

Sinks Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`ZoomMode`		string — `{'on'} | 'xonly' | 'yonly'`
`AxesTitles`		string
`Grid`		string — `'off' | {'on'} | 'xonly' | 'yonly'`
`TimeRange`		string — `{'auto'}`
`YMin`		string — `{'-5'}`
`YMax`		string — `{'5'}`
`SaveToWorkspace`		string — `{'off'} | 'on'`
`SaveName`		string — `{'ScopeData1'}`
`DataFormat`		string — `{'StructureWithTime'} | 'Structure' | 'Array'`
`LimitDataPoints`		string — `'off' | {'on'}`
`MaxDataPoints`		string — `{'5000'}`
`Decimation`		string — `{'1'}`
`SampleInput`		string — `{'off'} | 'on'`
`SampleTime`		string — `{'0'}`
Stop Simulation		
Terminator		
To File (`ToFile`)		
`FileName`	File name	string — `{'untitled.mat'}`
`MatrixName`	Variable name	string — `{'ans'}`
`SaveFormat`	Save format	string — `{'Timeseries'} | 'Array'`
`Decimation`	Decimation	string — `{'1'}`

Sinks Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
To Workspace (`ToWorkspace`)		
`VariableName`	Variable name	string — `{'simout'}`
`MaxDataPoints`	Limit data points to last	string — `{'inf'}`
`Decimation`	Decimation	string — `{'1'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`SaveFormat`	Save format	string — `{'Timeseries'} | 'Structure With Time' | 'Structure' | 'Array'`
`FixptAsFi`	Log fixed-point data as an fi object	string — `{'off'} | 'on'`
XY Graph (`XY scope`) (masked subsystem)		
`xmin`	x-min	string — `{'-1'}`
`xmax`	x-max	string — `{'1'}`
`ymin`	y-min	string — `{'-1'}`
`ymax`	y-max	string — `{'1'}`
`st`	Sample time	string — `{'-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	string — `{'[0.1]'}`
`Ts`	Sample time	string — `{'0.1'}`
`seed`	Seed	string — `{'[23341]'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Chirp Signal (`chirp`) (masked subsystem)		
`f1`	Initial frequency	string — `{'0.1'}`
`T`	Target time	string — `{'100'}`
`f2`	Frequency at target time	string — `{'1'}`
`VectorParams1D`	Interpret vectors parameters as 1-D	string — `'off' | {'on'}`
Clock (`Clock`)		
`DisplayTime`	Display time	string — `{'off'} | 'on'`
`Decimation`	Decimation	string — `{'10'}`
Constant (`Constant`)		
`Value`	Constant value	string — `{'1'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`
`SamplingMode`	Sampling mode	string — `{'Sample based'} | 'Frame based'`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'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>'`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — {`'off'`} | `'on'`
`SampleTime`	Sample time	string — {`'inf'`}
`FramePeriod`	Frame period	string — {`'inf'`}
Counter Free-Running (`Counter Free-Running`) (masked subsystem)		
`NumBits`	Number of Bits	string — {`'16'`}
`tsamp`	Sample time	string — {`'-1'`}
Counter Limited (`Counter Limited`) (masked subsystem)		
`uplimit`	Upper limit	string — {`'7'`}
`tsamp`	Sample time	string — {`'-1'`}
Digital Clock (`DigitalClock`)		
`SampleTime`	Sample time	string — {`'1'`}
Enumerated Constant (`EnumeratedConstant`)		
`OutDataTypeStr`	Output data type	string — {`'Enum: SlDemoSign'`}
`Value`	Value	string — {`'SlDemoSign.Positive'`} | `'SlDemoSign.Zero'` | `'SlDemoSign.Negative'`
`SampleTime`	Sample time	string — {`'inf'`}
From File (`FromFile`)		
`FileName`	File name	string — {`'untitled.mat'`}
`ExtrapolationBeforeFirstDataPoint`	Data extrapolation before first data point	string — {`'Linear extrapolation'`} | `'Hold first value'` | `'Ground value'`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`InterpolationWithinTimeRange`	Data interpolation within time range	string — `{'Linear interpolation'} | 'Zero order hold'`
`ExtrapolationAfterLastDataPoint`	Data extrapolation after last data point	string — `{'Linear extrapolation'} | 'Hold last value' | 'Ground value'`
`SampleTime`	Sample time	string — `{'0'}`
From Workspace (`FromWorkspace`)		
`VariableName`	Data	string — `{'simin'}`
`OutDataTypeStr`	Output Data type	string — `{'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	string — `{'0'}`
`Interpolate`	Interpolate data	string — `'off' | {'on'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off' | {'on'}`
`OutputAfterFinalValue`	Form output after final data value by	string — `{'Extrapolation'} | 'Setting to zero' | 'Holding final value' | 'Cyclic repetition'`
Ground		
In1 (`Inport`)		
`Port`	Port number	string — `{'1'}`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`IconDisplay`	Icon display	string — `'Signal name' | {'Port number'} | 'Port number and signal name'`
`BusOutputAsStruct`	Output as nonvirtual bus	string — `{'off'} | 'on'`
`PortDimensions`	Port dimensions (-1 for inherited)	string — `{'-1'}`
`VarSizeSig`	Variable-size signal	string — `{'Inherit'} | 'No' | 'Yes'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`OutMin`	Minimum	string — `{'[]'}`
`OutMax`	Maximum	string — `{'[]'}`
`OutDataTypeStr`	Data type	string — `{'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	string — `{'off'} | 'on'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
`SamplingMode`	Sampling mode	string — `{'auto'} | 'Sample based' | 'Frame based'`
`LatchByDelayingOutsideSignal`	Latch input by delaying outside signal	string — `{'off'} | 'on'`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`LatchInputForFeedbackSignals`	Latch input for feedback signals of function-call subsystem outputs	string — `{'off'} | 'on'`
`OutputFunctionCall`	Output a function-call trigger signal	string — `{'off'} | 'on'`
`Interpolate`	Interpolate data	string — `'off' | {'on'}`
Pulse Generator (`DiscretePulseGenerator`)		
`PulseType`	Pulse type	string — `{'Time based'} | 'Sample based'`
`TimeSource`	Time (t)	string — `{'Use simulation time'} | 'Use external signal'`
`Amplitude`	Amplitude	string — `{'1'}`
`Period`	Period	string — `{'10'}`
`PulseWidth`	Pulse width	string — `{'5'}`
`PhaseDelay`	Phase delay	string — `{'0'}`
`SampleTime`	Sample time	string — `{'1'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`
Ramp (`Ramp`) (masked subsystem)		
`slope`	Slope	string — `{'1'}`
`start`	Start time	string — `{'0'}`
`X0`	Initial output	string — `{'0'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`
Random Number (`RandomNumber`)		
`Mean`	Mean	string — `{'0'}`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`Variance`	Variance	string — {`'1'`}
`Seed`	Seed	string — {`'0'`}
`SampleTime`	Sample time	string — {`'0.1'`}
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off'` | {`'on'`}
Repeating Sequence (`Repeating table`) (masked subsystem)		
`rep_seq_t`	Time values	string — {`'[0 2]'`}
`rep_seq_y`	Output values	string — {`'[0 2]'`}
Repeating Sequence Interpolated (`Repeating Sequence Interpolated`) (masked subsystem)		
`OutValues`	Vector of output values	string — {`'[3 1 4 2 1].'''`}
`TimeValues`	Vector of time values	string — {`'[0 0.1 0.5 0.6 1].'''`}
`LookUpMeth`	Lookup Method	string — {`'Interpolation-Use End Values'`} | `'Use Input Nearest'` | `'Use Input Below'` | `'Use Input Above'`
`tsamp`	Sample time	string — {`'0.01'`}
`OutMin`	Output minimum	string — {`'[]'`}
`OutMax`	Output maximum	string — {`'[]'`}
`OutDataTypeStr`	Output data type	string — `'Inherit:  Inherit via back propagation'` | {`'double'`} | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`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	string — `{'off'} | 'on'`
Repeating Sequence Stair (`Repeating Sequence Stair`) (masked subsystem)		
`OutValues`	Vector of output values	string — `{'[3 1 4 2 1].''}`
`tsamp`	Sample time	string — `{'-1'}`
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `'Inherit: Inherit via back propagation' | {'double'} | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>'`
`OutputDataTypeScaling Mode`	Deprecated in R2007b	
`OutDataType`	Deprecated in R2007b	
`ConRadixGroup`	Deprecated in R2007b	
`OutScaling`	Deprecated in R2007b	
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Signal Builder (`Sigbuilder block`) (masked subsystem)		
Signal Generator (`SignalGenerator`)		
`WaveForm`	Wave form	string — `{'sine'} | 'square' | 'sawtooth' | 'random'`
`TimeSource`	Time (t)	string — `{'Use simulation time'} | 'Use external signal'`
`Amplitude`	Amplitude	string — `{'1'}`
`Frequency`	Frequency	string — `{'1'}`
`Units`	Units	string — `'rad/sec' | {'Hertz'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`
Sine Wave (`Sin`)		
`SineType`	Sine type	string — `{'Time based'} | 'Sample based'`
`TimeSource`	Time	string — `{'Use simulation time'} | 'Use external signal'`
`Amplitude`	Amplitude	string — `{'1'}`
`Bias`	Bias	string — `{'0'}`
`Frequency`	Frequency	string — `{'1'}`
`Phase`	Phase	string — `{'0'}`
`Samples`	Samples per period	string — `{'10'}`
`Offset`	Number of offset samples	string — `{'0'}`
SampleTime	Sample time	string — `{'0'}`

Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
VectorParams1D	Interpret vector parameters as 1-D	string — `'off'` | `{'on'}`
Step (`Step`)		
`Time`	Step time	string — `{'1'}`
`Before`	Initial value	string — `{'0'}`
`After`	Final value	string — `{'1'}`
`SampleTime`	Sample time	string — `{'0'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off'` | `{'on'}`
`ZeroCross`	Enable zero-crossing detection	string — `'off'` | `{'on'}`
Uniform Random Number (`UniformRandomNumber`)		
`Minimum`	Minimum	string — `{'-1'}`
`Maximum`	Maximum	string — `{'1'}`
`Seed`	Seed	string — `{'0'}`
`SampleTime`	Sample time	string — `{'0.1'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off'` | `{'on'}`

User-Defined Functions Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
MATLAB Function (`Stateflow`) (masked subsystem)		
Fcn (`Fcn`)		
`Expr`	Expression	string — `{'sin(u(1)*exp(2.3*(-u(2))))'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`

User-Defined Functions Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Level-2 MATLAB S-Function (`M-S-Function`)		
`FunctionName`	S-function name	string — `{'mlfile'}`
`Parameters`	Parameters	string — `{''}`
Interpreted MATLAB Function (`MATLABFcn`)		
`MATLABFcn`	MATLAB function	string — `{'sin'}`
`OutputDimensions`	Output dimensions	string — `{'-1'}`
`OutputSignalType`	Output signal type	string — `{'auto'} | 'real' | 'complex'`
`Output1D`	Collapse 2-D results to 1-D	string — `'off' | {'on'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
S-Function (`S-Function`)		
`FunctionName`	S-function name	string — `{'system'}`
`Parameters`	S-function parameters	string — `{''}`
`SFunctionModules`	S-function modules	string — `{''}`
S-Function Builder (`S-Function Builder`) (masked subsystem)		
`FunctionName`	S-function name	string — `{'system'}`
`Parameters`	S-function parameters	string — `{''}`
`SFunctionModules`	S-function modules	string — `{''}`

Additional Discrete Block Library Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Fixed-Point State-Space (`Fixed-Point State-Space`) (masked subsystem)		
`A`	State Matrix A	string — `{'[2.6020 -2.2793 0.6708; 1 0 0; 0 1 0]'}`

Additional Discrete Block Library Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`B`	Input Matrix B	string — `{'[ 1; 0; 0]'}`
`C`	Output Matrix C	string — `{'[0.0184 0.0024 0.0055]'}`
`D`	Direct Feedthrough Matrix D	string — `{'[0.0033]'}`
`X0`	Initial condition for state	string — `{'0.0'}`
`InternalDataType`	Data type for internal calculations	string — `{'fixdt('double')'}`
`StateEqScaling`	Scaling for State Equation AX+BU	string — `{'2^0'}`
`OutputEqScaling`	Scaling for Output Equation CX+DU	string — `{'2^0'}`
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'} | 'on'`
Transfer Fcn Direct Form II (`Transfer Fcn Direct Form II`) (masked subsystem)		
`NumCoefVec`	Numerator coefficients	string — `{'[0.2 0.3 0.2]'}`
`DenCoefVec`	Denominator coefficients excluding lead (which must be 1.0)	string — `{'[-0.9 0.6]'}`
`vinit`	Initial condition	string — `{'0.0'}`

Additional Discrete Block Library Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'}` | `'on'`
Transfer Fcn Direct Form II Time Varying (`Transfer Fcn Direct Form II Time Varying`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0.0'}`
`RndMeth`	Integer rounding mode	string — `'Ceiling'` | `'Convergent'` | `{'Floor'}` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`
`DoSatur`	Saturate to max or min when overflows occur	string — `{'off'}` | `'on'`
Unit Delay Enabled (`Unit Delay Enabled`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0.0'}`
`tsamp`	Sample time	string — `{'-1'}`
Unit Delay Enabled External IC (`Unit Delay Enabled External Initial Condition`) (masked subsystem)		
`tsamp`	Sample time	string — `{'-1'}`
Unit Delay Enabled Resettable (`Unit Delay Enabled Resettable`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0.0'}`
`tsamp`	Sample time	string — `{'-1'}`
Unit Delay Enabled Resettable External IC (`Unit Delay Enabled Resettable External Initial Condition`) (masked subsystem)		
`tsamp`	Sample time	string — `{'-1'}`
Unit Delay External IC (`Unit Delay External Initial Condition`) (masked subsystem)		

Additional Discrete Block Library Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
`tsamp`	Sample time	string — {`'-1'`}
Unit Delay Resettable (`Unit Delay Resettable`) (masked subsystem)		
`vinit`	Initial condition	string — {`'0.0'`}
`tsamp`	Sample time	string — {`'-1'`}
Unit Delay Resettable External IC (`Unit Delay Resettable External Initial Condition`) (masked subsystem)		
`tsamp`	Sample time	string — {`'-1'`}
Unit Delay With Preview Enabled (`Unit Delay With Preview Enabled`) (masked subsystem)		
`vinit`	Initial condition	string — {`'0.0'`}
`tsamp`	Sample time	string — {`'-1'`}
Unit Delay With Preview Enabled Resettable (`Unit Delay With Preview Enabled Resettable`) (masked subsystem)		
`vinit`	Initial condition	string — {`'0.0'`}
`tsamp`	Sample time	string — {`'-1'`}
Unit Delay With Preview Enabled Resettable External RV (`Unit Delay With Preview Enabled Resettable External RV`) (masked subsystem)		
`vinit`	Initial condition	string — {`'0.0'`}
`tsamp`	Sample time	string — {`'-1'`}
Unit Delay With Preview Resettable (`Unit Delay With Preview Resettable`) (masked subsystem)		
`vinit`	Initial condition	string — {`'0.0'`}
`tsamp`	Sample time	string — {`'-1'`}
Unit Delay With Preview Resettable External RV (`Unit Delay With Preview Resettable External RV`) (masked subsystem)		
`vinit`	Initial condition	string — {`'0.0'`}
`tsamp`	Sample time	string — {`'-1'`}

Additional Math: Increment - Decrement Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Decrement Real World (`Real World Value Decrement`) (masked subsystem)		
Decrement Stored Integer (`Stored Integer Value Decrement`) (masked subsystem)		
Decrement Time To Zero (`Decrement Time To Zero`) (masked subsystem)		
Decrement To Zero (`Decrement To Zero`) (masked subsystem)		
Increment Real World (`Real World Value Increment`) (masked subsystem)		
Increment Stored Integer (`Stored Integer Value Increment`) (masked subsystem)		

Mask Parameters

About Mask Parameters

This section lists parameters that describe masked blocks. You can use these descriptive parameters with `get_param` and `set_param` to obtain and specify the properties of a block mask.

The descriptive mask parameters listed in this section apply to all masks, and provide access to all mask properties. Be careful not to confuse these descriptive mask parameters with the mask-specific parameters defined for an individual mask in the Mask Editor **Parameters** pane.

See “What Are Masks?” and “Masking” 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.	pipe-delimited string `{''}`
`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.	string `{''}`

Mask Parameters (Continued)

Parameter	Description/Prompt	Values
`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.	string `{''}`
`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 string `{''}`
`MaskEnables`	Cell array version of `MaskEnableString`.	cell array of strings, each either `'on'` or ’`'off'` `{'[]'}`
`MaskHelp`	Block help. Set by the **Mask help** field on the **Documentation** pane of the Mask Editor dialog box.	`string {''}`
`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 (Opaque is `on`, Transparent is `off`). Set by the **Icon Transparency** option on the **Icon & Ports** pane of the Mask Editor dialog box.	`{'on'} | 'off'`

Mask Parameters (Continued)

Parameter	Description/Prompt	Values
`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 `{'[]'}`
`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 “Change the Appearance of a Block” in the Simulink documentation.	`{'default} | 'physical'`

Mask Parameters (Continued)

Parameter	Description/Prompt	Values
`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 strings `{'[]'}`
`MaskPromptString`	List of dialog parameter prompts (see below). Set inside the **Dialog parameters** area on the **Parameters** pane of the Mask Editor dialog box.	string `{''}`
`MaskPropertyNameString`	Pipe-delimited version of `MaskNames`.	string `{''}`
`MaskRunInitForIconRedraw`	For internal use only.	
`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'}`
`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`.	string `{''}`
`MaskTabNameString`	For internal use only.	
`MaskTabNames`	For internal use only.	

Mask Parameters (Continued)

Parameter	Description/Prompt	Values
`MaskToolTipsDisplay`	Determines which mask dialog parameters to display in the data tip for this masked block (see "Block Data Tips" in the Simulink documentation). 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`.	string `{''}`
`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 strings `{'[]'}`
`MaskTunableValueString`	Comma-delimited string version of `MaskTunableValues`.	delimited string `{''}`
`MaskType`	Mask type. Set by the **Mask type** field on the **Documentation** pane of the Mask Editor dialog box.	string `{'Stateflow'}`
`MaskValues`	Dialog parameter values.	cell array `{'[]'}`
`MaskValueString`	Delimited string version of `MaskValues`.	delimited string `{''}`

Mask Parameters (Continued)

Parameter	Description/Prompt	Values
`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.	string `{''}`
`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 string version of `MaskVisibilities`.	string `{''}`
`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

Simulink Identifier

Simulink Identifier

A Simulink Identifier (SID) is a unique identifier automatically assigned to a Simulink block, model annotation, or a Stateflow object within a Stateflow chart. The SID has the following characteristics:

- Persistent within the lifetime of a Simulink block, model annotation, or Stateflow object.
- Saved in the model file.
- If the name of a block or object changes, the SID remains the same.
- You cannot modify the SID.

The SID format is

```
model_name:sid_number
```

where

- `model_name` is the name of the model where the block, annotation, or object resides
- `sid_number` is a unique number within the model, assigned by Simulink

8

Model Advisor Checks

Simulink Checks

In this section...

"Simulink Check Overview" on page 8-4

"Identify unconnected lines, input ports, and output ports" on page 8-5

"Check root model Inport block specifications" on page 8-6

"Check optimization settings" on page 8-7

"Check for parameter tunability information ignored for referenced models" on page 8-10

"Check for implicit signal resolution" on page 8-11

"Check for optimal bus virtuality" on page 8-12

"Check for Discrete-Time Integrator blocks with initial condition uncertainty" on page 8-13

"Identify disabled library links" on page 8-14

"Identify parameterized library links" on page 8-15

"Identify unresolved library links" on page 8-16

"Identify model reference variants and variant subsystems that override variant choice" on page 8-17

"Identify configurable subsystem blocks for converting to variant subsystem blocks" on page 8-18

"Check usage of function-call connections" on page 8-18

"Check signal logging save format" on page 8-19

"Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues" on page 8-21

"Check if read/write diagnostics are enabled for data store blocks" on page 8-23

"Check data store block sample times for modeling errors" on page 8-25

"Check for potential ordering issues involving data store access" on page 8-26

"Check for partial structure parameter usage with bus signals" on page 8-28

In this section...

"Check for calls to slDataTypeAndScale" on page 8-30

"Check bus usage" on page 8-32

"Check for potentially delayed function-call subsystem return values" on page 8-34

"Identify block output signals with continuous sample time and non-floating point data type" on page 8-35

"Check Merge block usage" on page 8-36

"Check consistency of initialization parameters for Outport and Merge blocks" on page 8-37

"Check for non-continuous signals driving derivative ports" on page 8-55

"Runtime diagnostics for S-functions" on page 8-57

"Check file for foreign characters" on page 8-59

"Check model for known block upgrade issues" on page 8-64

"Check model for known block upgrade issues requiring compile time information" on page 8-65

"Check that the model is saved in SLX format" on page 8-68

"Check Model History properties" on page 8-70

"Analyze model hierarchy for upgrade issues" on page 8-72

"Check and update masked blocks in library to use promoted parameters" on page 8-74

Simulink Check Overview

Use the Simulink Model Advisor checks to configure your model for simulation.

See Also

- "Consult the Model Advisor".
- "Embedded Coder Checks".
- "Simulink Verification and Validation Checks".

Identify unconnected lines, input ports, and output ports

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.

Tips

Use the `PortConnectivity` command to obtain an array of structures describing block input or output ports.

See Also

“Common Block Properties” on page 6-96 for information on the `PortConnectivity` command.

Check root model Inport block specifications

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.

See Also

- “Data Types”.
- “Determine Output Signal Dimensions”.
- “Specify Sample Time”.

Check optimization settings

Check for optimizations that can lead to nonoptimal 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: • **“Block reduction”** • **“Conditional input branch execution”** • **“Implement logic signals as Boolean data (vs. double)”** • **“Remove root level I/O zero initialization”** • **“Remove internal data zero initialization”** • **“Use memset to initialize floats and doubles to 0.0”** • **“Remove code from floating-point to integer conversions that wraps out-of-range values”** (only if you have a Simulink Coder license) Select the following optimization check boxes on the **Optimization > Signals and Parameters** pane in the Configuration Parameters dialog box: • **“Inline parameters”** • **“Enable local block outputs”** • **“Reuse block outputs”**

Condition	Recommended Action
	- **“Eliminate superfluous local variables (Expression folding)”** - **“Inline invariant signals”** (only if you have a Simulink Coder license) Select the following optimization check boxes on the **Optimization > Stateflow** pane in the Configuration Parameters dialog box: - **“Use bitsets for storing state configuration”** - **“Use bitsets for storing Boolean data”**
“Application lifespan (days)” is set as infinite. This could lead to expensive 64-bit counter usage.	Choose a stop time if this is not intended.
The specified diagnostics, which can increase the time it takes to simulate your model, are set to `warning` or `error`.	Select `none` for: - **Diagnostics > Solver > Solver data inconsistency** - **Diagnostics > Data Validity > Array bounds exceeded** - **Diagnostics > Data Validity > Simulation range checking**
The specified Embedded Coder parameters are off.	If you have a Embedded Coder license, and you are using an ERT-based system target file, select the following check boxes: - **Code Generation > Interface > “Single output/update function”** - **Code Generation > General > “Ignore test point signals”** - **Optimization > “Pass reusable subsystem outputs as”**

Tips

If the system contains Model blocks and the referenced model is in Accelerator mode, simulating the model requires generating and compiling code.

See Also

- "Optimization Pane: General".

Check for parameter tunability information ignored for referenced models

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

"Parameters".

Check for implicit signal resolution

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

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.

See Also

- “Signal Basics”.
- “Virtual and Nonvirtual Buses”.

Check for Discrete-Time Integrator blocks with initial condition uncertainty

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.

See Also

- IC block
- Discrete-Time Integrator block
- Constant block

Identify disabled library links

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 known block upgrade issues" on page 8-64.

Results and Recommended Actions

Condition	Recommended Action
Library links are disabled.	Use `Restore Link` from the `Link Options` setting in the context menu.

Capabilities and Limitations

You can run this check on your library models.

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. Choose `Link Options` and click `Restore Link`.

See Also

"Restore Disabled or Parameterized Links".

Identify parameterized library links

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.

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".

Identify unresolved library links

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.

See Also

“Fix Unresolved Library Links”

Identify model reference variants and variant subsystems that override variant choice

Identify model or subsystem for model reference variants and variant subsystems that specify variant choice using the override option instead of using the active variant object.

Results and Recommended Actions

Condition	Recommended Action
Model reference variants or variant subsystems that override variant choice are identified.	Specify variant choice using active variant object.

Capabilities and Limitations

You can run this check on your library models.

See Also

"Select the Active Variant for Simulation"

Identify configurable subsystem blocks for converting to variant subsystem blocks

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 “Convert Subsystem Blocks to Variant Subsystem Blocks”

Capabilities and Limitations

You can run this check on your library models.

See Also

“Set Up Model Variants”

Check usage of function-call connections

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 `none` or `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 signal logging save format

Check signal logging save format.

Description

Check signal logging save format. The signal logging save format `ModelDataLogs` will be removed in a future release. To take advantage of new functionality, update models using `ModelDataLogs` format to use `Dataset` format.

Results and Recommended Actions

Condition	Recommended Action
The model has the signal logging format set to `Dataset`.	No action required.
The model has the signal logging format set to `ModelDataLogs`.	Use the Upgrade Advisor (with the `upgradeadvisor` function) to upgrade a model to use `Dataset` format. Enable the **Check signal logging save format** check, run the check, and click the **Update format** button.
The model contains Model blocks. Models in the model reference hierarchy require the same signal logging save format.	Use the Upgrade Advisor (with the `upgradeadvisor` function) to upgrade a model to use `Dataset` format. Enable the **Check signal logging save format** check, run the check, and click the **Update format** button.

See Also

"Specify the Signal Logging Data Format"

Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues

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.
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.

See Also

- "About Data Stores"
- "Storage Classes for Data Store Memory Blocks"

- Data Store Memory
- Data Store Read
- Data Store Write
- “Duplicate data store names”
- “Multitask data store”

Check if read/write diagnostics are enabled for data store blocks

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.

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

- “About Data Stores”

- Data Store Memory
- Data Store Read
- Data Store Write
- “Detect read before write”
- “Detect write after read”
- “Detect write after write”
- “Check for potential ordering issues involving data store access” on page 8-26

Check data store block sample times for modeling errors

Identify modeling errors due to the sample times of data store blocks.

Description

Check data store blocks for continuous or zero-order hold sample times.

Results and Recommended Actions

Condition	Recommended Action
Data store blocks in your model have continuous or zero-order-hold sample times.	Consider making the listed blocks discrete or replacing them with either Memory or Goto and From blocks.

See Also

- “About Data Stores”
- Data Store Memory
- Data Store Read
- Data Store Write

Check for potential ordering issues involving data store access

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.

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

- “About 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 partial structure parameter usage with bus signals

Identify blocks that use partial structures as parameter values for bus signals.

Description

This check compares structures that provide parameter values for bus signals, to identify partial structures. This check returns a table listing the:

- Paths to blocks that use partial structures as parameter values for bus signals
- Names the block parameters that use the partial structure

For all data stores that you define with a `Simulink.Signal` object that uses a partial structure for its **Initial value** parameter, this check lists the following information, in a second table:

- Name of the signal object
- Workspace (MATLAB or model) of the signal object
- Name of the signal object parameter that uses the partial structure

Results and Recommended Actions

Condition	Recommended Action
Block using partial structure	Consider using the `Simulink.Bus.createMATLABStructure` function to convert a partial structure parameter to a full structure of parameter values for the listed blocks.
Signal objects using partial structure	Consider using the `Simulink.Bus.createMATLABStructure` function to convert a partial structure parameter to a full structure of parameter values for the listed signals.

Tips

- Specifying *partial* structures for block parameter values can be useful during the iterative process of creating a model. You can use partial structures to focus on a subset of signals in a bus.
- Specifying *full* structures for code generation offers these advantages:
  - Generated code is more readable than the code generated for partial structures.
  - Supports a modeling style that explicitly initializes unspecified signals. When you use partial structures, Simulink initializes unspecified signals implicitly.

See Also

- “Specify Initial Conditions for Bus Signals”
- “Data Stores with Signal Objects”
- `Simulink.Bus.createMATLABStruct`
- `Simulink.Signal`

Check for calls to slDataTypeAndScale

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.

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`

Check bus usage

Identify Mux blocks used as a bus creator and bus signal that Simulink treats as a vector.

Description

Models should not contain bus signals that 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
Model uses Mux blocks to create bus signals.	Replace Mux blocks with Bus Creator blocks.
Model is not configured to identify Mux blocks used as bus creators.	In the Configuration Parameters dialog box, on the **Diagnostics > Connectivity** pane, set **Mux blocks used to create bus signals** to `error`.
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 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** causes one of the following:

- Mux blocks are replaced by Bus Creator blocks.
- Bus to Vector blocks are inserted at the inports of blocks that implicitly convert bus signals to vectors.

Tips

- The Bus to Vector conversion block resides in the Simulink/Signal Attributes library.
- Run this check before running **Check consistency of initialization parameters for Outport and Merge blocks**.
- The “Non-bus signals treated as bus signals” diagnostic detects when Simulink implicitly converts a non-bus signal to a bus signal to support connecting the signal to a Bus Assignment or Bus Selector block. This diagnostic is in the Configuration Parameters dialog box, on the **Diagnostics> Connectivity** pane.
- Using a Mux block to create a virtual bus does not support strong type checking and increases the likelihood of run-time errors. In new applications, do not use Mux blocks to create bus signals. Consider upgrading existing applications to that use of Mux blocks.
    - Simulink generates a warning when you load a model created in a release prior to R2010a, if that model contains a Mux block to create a bus signal. For new models, Simulink generates an error.

See Also

- “Avoid Mux/Bus Mixtures”
- Bus to Vector block
- “Mux blocks used to create bus signals”
- “Bus signal treated as vector”
- “Check consistency of initialization parameters for Outport and Merge blocks” on page 8-37
- `Simulink.BlockDiagram.addBusToVector`

Check for potentially delayed function-call subsystem return values

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.

See Also

Signal Conversion block

Unit Delay block

Identify block output signals with continuous sample time and non-floating point data type

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]).

See Also

“What Is Sample Time?”.

Check Merge block usage

Analyze Merge blocks in the same tree as a group, and determine the possibility for them to execute at the same time step.

Description

Blocks that directly drive the same tree of Merge blocks should have mutually exclusive execution in each time step. This check identifies those blocks that drive the same tree of Merge blocks, and so are likely to execute at the same time step.

Input Parameters

Maximum analysis time (seconds)
Provide a maximum analysis time to execute the check.

Results and Recommended Actions

Condition	Recommended Action
Merge blocks can be interconnected to form a tree structure.	Rework your model so that no blocks drive the same tree of Merge blocks.

See Also

- Merge block
- “Check consistency of initialization parameters for Outport and Merge blocks” on page 8-37

Check consistency of initialization parameters for Outport and Merge blocks

Identify Outport and Merge blocks with parameter settings that can lead to unexpected initialization behavior, and migrate your model to the simplified initialization mode.

Description

In R2008b or later versions, you can choose the simplified initialization mode for conditionally executed subsystems, Merge blocks, subsystem elapsed time, and Discrete-Time Integrator blocks. The simplified initialization mode improves the consistency of simulation results. This result is especially true for models that do not specify initial conditions for conditionally executed subsystem output ports, and for models that have conditionally executed subsystem output ports connected to S-functions.

Note Before running this consistency check, verify that your block diagram conforms to the modeling standards set by this diagnostic. For more information, see the Simulink documentation under 'Diagnostics Pane: Connectivity' or complete the following steps: .

1. Run **Check bus usage** in the Model Advisor to check your usage of Mux blocks.
2. In the model window, select **Simulation > Model Configuration Parameters > Diagnostics > Connectivity**.
3. Set **Mux blocks used to create bus signals** to `error`.
4. Set **Bus signal treated as vector** to `error`.

This Model Advisor check identifies settings in your model that can cause problems if you use the 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 the simplified initialization mode. Warning statements identify issues or changes in behavior that may occur after migration.

After running this Model Advisor consistency check, if you select the **Explore Result** button, then the messages will only pertain to blocks that are not library-linked blocks.

Note Because it is difficult to undo these changes, use the **Save Restore Point As** feature to backup your model before migrating to the simplified initialization mode.

Results and Recommended Actions

Condition	Recommended Action
Check the run-time diagnostic setting of the Merge block.	1 In the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane, 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.
Verify that 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.
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.

Condition	Recommended Action
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'.	Specify an explicit value for the **Initial output** parameter of *root Merge blocks*. A root Merge block is a Merge block with an output port that does not connect to another Merge block.
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.

Condition	Recommended Action
Check for Merge blocks with inputs that are combined or reordered outside of conditionally executed subsystems.	Verify that any combination 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.
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 leads 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

Condition	Recommended Action
	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 .
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:

Condition	Recommended Action
	- 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 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.
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.

Condition	Recommended Action
Check for Outport blocks that have an undefined 'Initial output' value.	Specify the **Initial output** parameter for the Outport block, unless you want the block to obtain its **Initial output** value from its input signal. If you want the Outport block to obtain its **Initial output** value from its input signal, verify that its sources are valid. For the simplified initialization mode, valid sources from which an Outport blocks can inherit **Initial output** value are: Constant, Initial Condition, Merge (with initial output), or conditionally executed subsystem blocks. **Note** If you are working with a conditionally executed subsystem Outport block, driven directly by another conditionally executed subsystem Outport block, then resolve this error message for the inner Outport block first. Subsequently, run this Model Advisor check again to determine if you resolved the problem for the outer Outport block.

Condition	**Recommended Action**
Check Outport blocks that have automatic rate transitions.	Simulink has inserted a Rate Transition block at the input of the Outport block. If this rate transition is required, then 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.
Check Outport blocks that have a special signal storage requirement.	Specify the **Initial output** parameter for the Outport block. The Outport block cannot obtain **Initial output** values from its input signals because of its special internal signal storage requirement.
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.

Condition	Recommended Action
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.
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 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 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 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.
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.

Condition	Recommended Action
Check for innermost Outport blocks with variable-size input and an unspecified ’Initial output’.	In classic initialization mode, Simulink assumes that the ’Initial output’ parameter is 0 when it is unspecified ([]) for the innermost Outport block. This block has variable-size input for which the signal size varies only when the parent subsystem is re-enabled. Unless you specify the value of the **Initial output** parameter, it will be set to zero when your model is migrated to the simplified initialization mode.
Check for outer Outport blocks that have an explicit ’Initial output’.	Verify that the following behavior is acceptable. In the classic initialization mode, the **Initial output** and **Output when disabled** parameters of the Outport block must match those of their source Outport blocks. Consequently, Simulink will set the ’Source of initial output value’ parameter to ’Input signal’ in the simplified initialization mode.
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 Discrete-Time Integrator blocks.	Verify that the following behavior is acceptable. Simulink has changed the initialization behavior of the Discrete-Time Integrator block to be more consistent and robust in the simplified initialization mode. These changes include: • Elimination of the **Use initial condition as initial and reset value for** parameter • Application of the **Initial condition** parameter only to the integrator output • Simplified enable/disable behavior. See the Discrete-Time Integrator block documentation for more information on the simplified initialization for the Discrete-Time Integrator.
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

Condition	Recommended Action
	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 library blocks with instances that cannot be migrated.	Examine the failed subcheck results for each block to determine the corrective actions.
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 a migration conflict for Outport blocks that will 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 will use an '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.`

Condition	Recommended Action
	`Simulink will maintain these settings` because their parent conditionally executed subsystems are connected to SimEvents blocks.
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 merged Outport blocks without explicit specification of 'Initial output'.	Review the results from subcheck **Check for library blocks with instances that cannot be migrated** to learn about different migration paths for other instances of each Outport block

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 string , [] , 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

- “Consult the Model Advisor”
- “Simulink Checks” on page 8-2
- “Check bus usage” on page 8-32
- Merge block
- Discrete-Time Integrator block
- Outport block
- “Mux blocks used to create bus signals”
- “Bus signal treated as vector”
- “Detect multiple driving blocks executing at the same time step”
- “Underspecified initialization detection”

Check for non-continuous signals driving derivative ports

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.

See Also

- “Modeling Dynamic Systems”
- “Simulating Dynamic Systems”

Runtime diagnostics for S-functions

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, on the **Diagnostics** pane, set **Solver data inconsistency** to `warning` or `error`.
Array bounds exceeded is set to `none`.	In the Configuration Parameters dialog box, on the **Diagnostics> Data Validity** pane, set **Array bounds exceeded** to `warning` or `error`

See Also

- "What Is an S-Function?"
- "How S-Functions Work"

Check file for foreign characters

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`
- "Models with Different Character Encodings"
- "Saving Models with Different Character Encodings"

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check model for known block upgrade issues

Uses the `slupdate` function to 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

- `slupdate`.
- “Write Level-2 MATLAB S-Functions”.
- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check model for known block upgrade issues requiring compile time information

Uses the `slupdate` function to 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.

Action Results

Clicking **Modify** replaces blocks from a previous release of Simulink software with the latest versions.

See Also

- `slupdate`.
- n-D Lookup Table.
- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check that the model is saved in SLX format

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.

Tips

Simulink Projects can help you upgrade models to SLX format and preserve file revision history in source control. See "Simulink Projects".

See Also

- "Saving Models in the SLX File Format"
- "Consult the Upgrade Advisor".
- "Model Upgrades"

Check Model History properties

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.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Analyze model hierarchy for upgrade issues

Check for child models and guide you through upgrading a model hierarchy.

Description

This check identifies child models of this model, and offers to run the Upgrade Advisor upon each child model in turn. If you have a model hierarchy you need to check and update each child model. The Advisor provides tools to help with these tasks.

Results and Recommended Actions

Condition	Recommended Action
Child models found	Click **Analyze Next Model** to close the current Upgrade Advisor session, and open Upgrade Advisor for the next model in the hierarchy.

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 **Analyze Next Model**, 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 **Analyze Next Model** 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 **Analyze Next Model** 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 **Analyze Next Model** to open the next model. Continue until you reach the end of the hierarchy.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check and update masked blocks in library to use promoted parameters

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

See Also

- "Consult the Upgrade Advisor".
- "Model Upgrades"

9

Performance Advisor Checks

Simulink Performance Advisor Checks

In this section...

"Simulink Performance Advisor Check Overview" on page 9-2

"Baseline" on page 9-3

"Checks that require Update Diagram" on page 9-3

"Checks that require simulation to run" on page 9-3

"Check Accelerator settings" on page 9-3

"Create Baseline" on page 9-3

"Identify resource intensive diagnostic settings" on page 9-4

"Check optimization settings" on page 9-4

"Identify inefficient lookup table blocks" on page 9-5

"Identify Interpreted MATLAB Function blocks" on page 9-5

"Check MATLAB Function block debug settings" on page 9-6

"Check Stateflow block debug settings" on page 9-6

"Identify simulation target settings" on page 9-6

"Check model reference rebuild setting" on page 9-7

"Check Model Reference Parallel Build" on page 9-7

"Check Delay block circular buffer setting" on page 9-9

"Check solver type selection" on page 9-9

"Select normal or accelerator simulation mode" on page 9-11

Simulink Performance Advisor Check Overview

Use Performance Advisor checks to improve model simulation time.

See Also

"How Performance Advisor Improves Simulation Performance"

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

"How Performance Advisor Improves Simulation Performance"

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

"How Performance Advisor Improves Simulation Performance"

Check Accelerator settings

These checks evaluate whether simulating in Accelerator mode can improve performance.

See Also

"What Is Acceleration?"

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"
- "How Performance Advisor Improves Simulation Performance"

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.

When you enable Inline Parameters optimization, Simulink uses the numerical values of model parameters instead of their symbolic names. This substitution can improve performance by reducing the parameter tuning computations performed during simulations.

See Also

- “Optimization Pane: General”
- “How Performance Advisor Improves Simulation Performance”

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”
- “How Performance Advisor Improves Simulation Performance”

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
- “How Performance Advisor Improves Simulation Performance”

Check MATLAB Function block debug settings

After verifying that your MATLAB code works correctly, disable debugging support for MATLAB Function blocks where possible. In the Simulation Target pane of the Configuration Parameters dialog box, disable debugging/animation, overflow detection, and echoing expressions without semicolons.

See Also

"How Performance Advisor Improves Simulation Performance"

Check Stateflow block debug settings

By default, Stateflow charts highlight the current active states and animate the state transitions that take place as the model runs. This feature is useful for debugging, but it slows the simulation. To accelerate simulations, either close all Stateflow charts or disable the animation. Similarly, if you're using Simulink 3D Animation™, SimMechanics™ visualization, FlightGear, or another 3D animation package, consider disabling the animation or reducing scene fidelity to improve performance.

See Also

- "Speed Up Simulation"
- "Diagnostics Pane: Stateflow"
- "How Performance Advisor Improves Simulation Performance"

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

- "Simulation Target Pane: General"
- "How Performance Advisor Improves Simulation Performance"

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

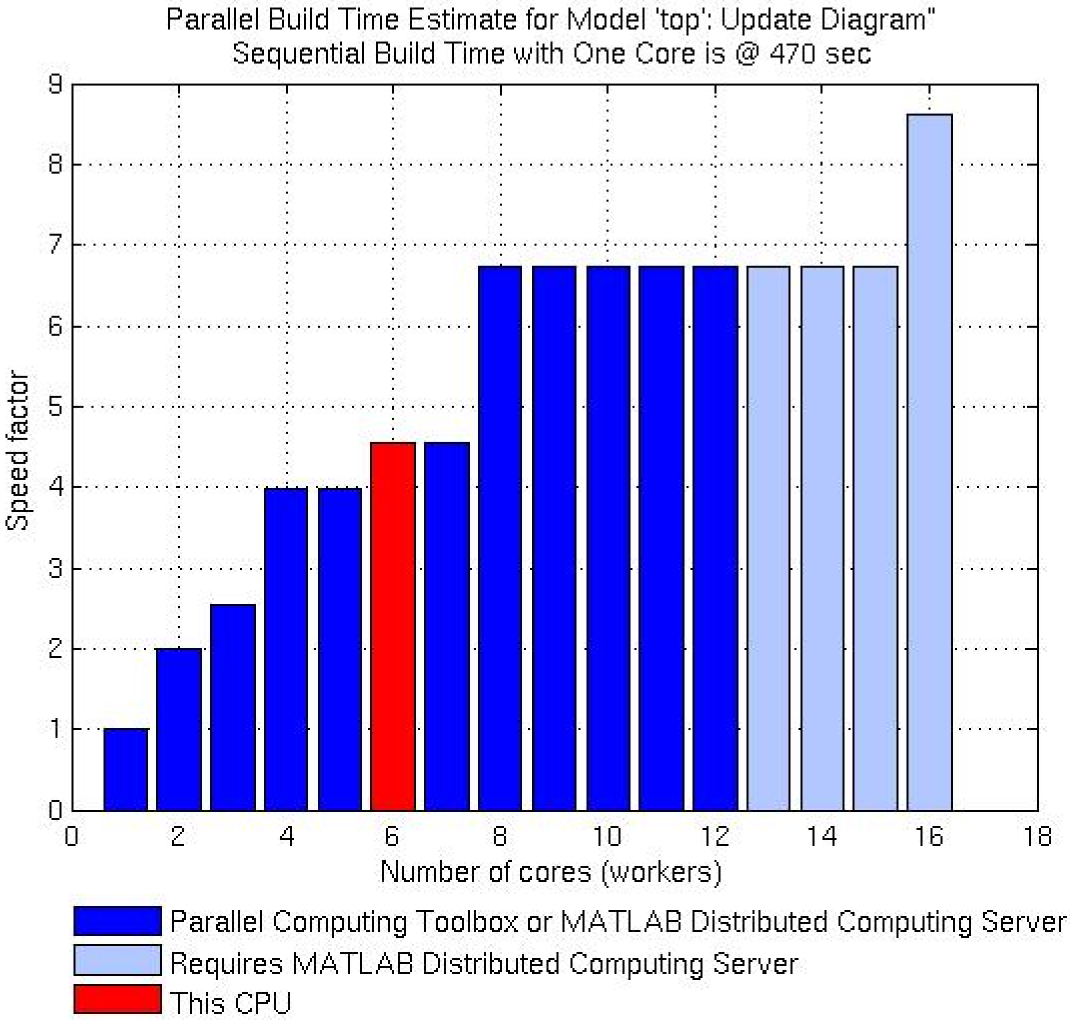
- "Rebuild"
- "How Performance Advisor Improves Simulation Performance"

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 using different numbers of cores. It also estimates the parallel build time for the model using more cores, if Parallel Computing Toolbox™ or MATLAB Distributed Computing Server™ software is 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.



See Also

- "Enable parallel model reference builds"
- "How Performance Advisor Improves Simulation Performance"

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
- "How Performance Advisor Improves Simulation 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

- “Choose a Solver”
- “Speed Up Simulation”
- “How Performance Advisor Improves Simulation Performance”

Select normal or accelerator simulation mode

To improve simulation, check to see if running the model in Accelerator mode can improve simulation speed.

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.

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.

See Also

- “How Acceleration Modes Work”
- “Choosing a Simulation Mode”
- “Comparing Performance”
- “How Performance Advisor Improves Simulation Performance”

10

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 string 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

Index

Symbols and Numerics

1-D Lookup Table block 1-887
2-D Lookup Table block 1-888

A

Abs block 1-2
absolute tolerance
  specifying for a block state 1-823 1-1483
absolute value
  generating 1-2
Action Port block 1-10
Action subsystems
  creating 1-10
  with If block 1-759
  with SwitchCase block 1-1743
ActionPort
  Propagate sizes of variable-size signals 1-14
  States when execution is resumed 1-12
Add block 1-1667
`add_block` command 2-2
`add_line` command 2-7
`add_param` command 2-9
Additional Discrete block library
  block parameters 6-255
Additional Math: Increment - Decrement block library
  block parameters 6-259
`addterms` command 2-12
Algebraic Constraint block 1-15
algebraic equations
  modeling 1-15
algebraic loops
  integrator block reset or IC port 1-459
analysis functions
  perturbing model 1-777
`animate` 4-4
annotations
  annotation block. *See* Model Info block
`ashow` debug command 4-5
Assertion block 1-18
Assignment block 1-21
Atomic Subsystem block 1-1633
`atrace` debug command 4-6
`attachConfigSet` command 2-13
`attachConfigSetCopy` command 2-15
automatic data typing 2-178
  safety margin 2-222
automatic scaling
  autoscale safety margin 2-221
  fixptbestprec 2-173
autoscaling
  fixptbestprec 2-173

B

Backlash block 1-28
Backward Euler method 1-457
Backward Rectangular method 1-457
Bad Link block 1-36
Band-Limited White Noise block 1-37
`bdclose` command 2-22
`bdIsLoaded` command 2-23
`bdroot` command 2-24
Bias block 1-41
bits
  clear 1-50
  mask 1-50
  set 1-50
Bitwise Operator block 1-48
block dialog boxes
  closing 2-31
  opening 2-310
block parameters
  Additional Discrete library 6-255
  Additional Math: Increment - Decrement library 6-259
  changing during simulation 2-333
  Continuous library 6-110
  Discontinuities library 6-115

    Discrete library 6-119
    Logic and Bit Operations library 6-137
    Lookup Tables library 6-141
    Math library 6-158
    Model Verification block library 6-183
    Model-Wide Utilities library 6-187
    Ports & Subsystems library 6-189
    Signal Attributes library 6-224
    Signal Routing library 6-232
    Sinks library 6-241
    Sources library 6-245
    User-defined functions library 6-254
block properties
    common 6-96
Block Support Table block 1-56
blocks 6-109
    adding to model 2-2
    Compare To Zero 1-150
    Counter Limited 1-200
    current 2-243
    Data Type Propagation 1-262
    Decrement Stored Integer 1-285
    Decrement Time To Zero 1-287
    Decrement To Zero 1-289
    deleting
        `delete_block` command 2-107
    Detect Decrease 1-330
    Detect Fall Negative 1-334
    Detect Fall Nonpositive 1-338
    Detect Increase 1-342
    Detect Rise Nonnegative 1-346
    Detect Rise Positive 1-350
    Filter Direct Form II 1-1879
    handle of current 2-244
    Increment Stored Integer 1-773
    Index Vector 1-775
    Interval Test Dynamic 1-861
    Repeating Sequence Interpolated 1-1410
    Repeating Sequence Stair 1-1415
    Transfer Fcn Direct Form II Time Varying 1-1882
    Transfer Fcn First Order 1-1886
    Transfer Fcn Lead or Lag 1-1889
    Transfer Fcn Real Zero 1-1892
    Unit Delay Enabled External IC 1-1986
    Unit Delay Enabled Resettable 1-1989
    Unit Delay Enabled Resettable External IC 1-1993
    Unit Delay External IC 1-1996
    Unit Delay Resettable 1-1999
    Unit Delay Resettable External IC 1-2002
    Unit Delay With Preview Enabled 1-2005
    Unit Delay With Preview Enabled Resettable 1-2009
    Unit Delay With Preview Enabled Resettable External RV 1-2013
    Unit Delay With Preview Resettable 1-2017
    Unit Delay With Preview Resettable External RV 1-2021
    Weighted Sample Time 1-2062
    See also block parameters
`bode` function 2-29
Boolean expressions
    modeling 1-140
`break` debug command 4-9
`bshow` debug command 4-11
Bus Assignment block 1-57
Bus Creator
    Data type 1-77
    Number of inputs 1-71
    Output as nonvirtual bus 1-80
    Rename selected signal 1-76
    Signal naming options 1-70
    Signals in bus 1-72
Bus Creator block 1-63
Bus Selector
    Output as bus 1-90
    Selected signals 1-89
    Signals in the bus 1-85

Bus Selector block 1-82
Bus to Vector block 1-92
buses 1-778

C

capping unconnected blocks
    using the Terminator block 1-1754
character encoding, model 2-537
Check Discrete Gradient block 1-95
Check Dynamic Gap block 1-99
Check Dynamic Lower Bound block 1-103
Check Dynamic Range block 1-107
Check Dynamic Upper Bound block 1-111
Check Input Resolution block 1-115
Check Static Gap block 1-118
Check Static Lower Bound block 1-122
Check Static Range block 1-126
Check Static Upper Bound block 1-131
Chirp Signal block 1-135
`clear` debug command 4-12
clearing bits 1-50
Clock block 1-138
`close_system` command 2-31
`closeDialog` command 2-34
code generation
    scaling 1-256
code generation from MATLAB
    resolving extrinsic function calls in generated code 2-56
coder.inline function 2-58
coder.nullcopy function 2-65
coder.ref function 2-73
coder.rref function 2-75
coder.target function 2-86
coder.wref function 2-99
`color` command 3-2
Combinatorial Logic block 1-140
combining input lines into vector line 1-1067
commands, simulation
    Simulink.Block.getSampleTimes 2-381
    Simulink.BlockDiagram.getChecksum 2-395
    Simulink.BlockDiagram.getInitialState 2-398
    Simulink.BlockDiagram.getSampleTimes 2-400
    Simulink.SubSystem.getChecksum 2-530
Compare To Constant block 1-147
Compare To Zero block 1-150
Complex to Magnitude-Angle block 1-153
Complex to Real-Imag block 1-156
Configurable Subsystem block 1-159
configuration parameters
    closing dialog 2-34
    opening dialog 2-314
configuration reference
    activating 2-336
    attaching 2-13
    copying and attaching 2-15
    detaching 2-117
    obtaining 2-253
        active 2-251
        list of names 2-254
configuration set
    activating 2-336
    attaching 2-13
    copying and attaching 2-15
    detaching 2-117
    obtaining 2-253
        active 2-251
        list of names 2-254
connecting
    buses to root-level inports 1-778
Constant
    Constant value 1-169
    Data type override 1-182 1-248 1-510 1-613 1-723 1-800 1-878 1-1054 1-1096 1-1172 1-1257 1-1331 1-1392 1-1568 1-1688 1-1733 1-1926
    Frame period 1-181
    Sample time 1-172
    Sampling mode 1-171 1-812

Constant block 1-165
constant value
    generating 1-165
`continue` debug command 4-13
Continuous block library
    block parameters 6-110
control flow diagrams
    Action subsystem 1-10
    `do-while`
        While Iterator block 1-2069
    `for`
        For Iterator block 1-650
    `if-else`
        `If` block 1-759
    `switch`
        Switch Case block 1-1743
    `while`
        While Iterator block 1-2069
Controlling inlining in generated code 2-58
Cosine block 1-1583
Coulomb and Viscous Friction block 1-191
Coulomb friction 1-191
Counter Free-Running block 1-195
Counter Limited block 1-200
Create Subsystem menu item 1-1633
current block
    getting pathname 2-243
    handle 2-244
current system
    getting pathname 2-245

D

Data Store Memory block 1-203
Data Store Read block 1-218
Data Store Write block 1-224
Data Type Conversion block 1-230
Data Type Conversion Inherited block 1-257
Data Type Duplicate block 1-260
Data Type Propagation block 1-262
Data Type Scaling Strip block 1-274
data types
    propagation 1-262
DataTypeConversion
    Input and output to have equal 1-236
Dead Zone block 1-276
Dead Zone Dynamic block 1-281
deadband 1-28
debug commands
    `ashow` 4-5
    `atrace` 4-6
    `break` 4-9
    `bshow` 4-11
    `clear` 4-12
    `continue` 4-13
    `disp` 4-14
    `emode` 4-20
    `etrace` 4-21
    `help` 4-22
    `nanbreak` 4-23
    `next` 4-24
    `probe` 4-25
    `quit` 4-27
    `status` 4-33
    `step` 4-35
    `stop` 4-38
    `strace` 4-39
    `systems` 4-42
    `tbreak` 4-43
    `trace` 4-44
    `undisp` 4-45
    `untrace` 4-46
    `xbreak` 4-49
    `zcbreak` 4-50
    `zclist` 4-51
decision tables
    modeling 1-140
Declaring uninitialized variables 2-65
Decrement Real World block 1-283
Decrement Stored Integer block 1-285

Decrement Time To Zero block 1-287
Decrement To Zero block 1-289
Delay block 1-291
delaying input by variable amount 1-2026
`delete_block` command 2-107
`delete_line` command 2-108
`delete_param` command 2-110
Demux
  Bus selection mode 1-319
  Display option 1-318
  Number of outputs 1-317
Demux block 1-310
Derivative block 1-320
  accuracy of 1-320
  linearization 1-321
  Linearization Time Constant 1-324
derivatives
  calculating 1-320
  limiting 1-1361
`detachConfigSet` command 2-117
Detect Change block 1-326
Detect Decrease block 1-330
Detect Fall Negative block 1-334
Detect Fall Nonpositive block 1-338
Detect Increase block 1-342
Detect Rise Nonnegative block 1-346
Detect Rise Positive block 1-350
Determining code generation target 2-86
Difference block 1-354
differential-algebraic systems
  modeling 1-15
Digital Clock block 1-360
Direct Lookup Table (n-D) block 1-363
Discontinuities block library
  block parameters 6-115
Discrete block library
  block parameters 6-119
Discrete Derivative block 1-377
Discrete Filter block 1-384
Discrete FIR Filter block 1-426
Discrete Integrator
  External reset 1-480
  Gain value 1-479
  Integrator method 1-478
  Use initial condition as initial and reset value for 1-483
Discrete State-Space block 1-448
discrete state-space model 2-28
Discrete Transfer Fcn block 1-518
Discrete Zero-Pole block 1-539
Discrete-Time Integrator block 1-456
discrete-time systems
  linearization 2-28
`disp` command 3-4
`disp` debug command 4-14
Display block 1-545
  as floating display 1-547
displaying
  signals graphically 1-1459
Divide block 1-551
DocBlock block 1-554
Dot Product block 1-557

E

eigenvalues of linearized matrix 2-28
`emode` debug command 4-20
Enable block 1-563
Enabled and Triggered Subsystem block 1-581
Enabled Subsystem block 1-582
enabled subsystems
  Enable block 1-563
EnablePort
  Enable zero-crossing detection 1-570
  Propagate sizes of variable-size signals 1-568
  Show output port 1-569
  States when enabling 1-567
Enumerated Constant block 1-583
enumerated constant value
  generating 1-583

Environment Controller block 1-587
`etrace` debug command 4-21
expressions
  applying to block inputs 1-595
    Interpreted MATLAB Function block 1-855
external inputs
  `flag` 2-301
  from workspace 1-777
Extract Bits block 1-590

F

Fcn block 1-595
  compared to Math Function block 1-930
  compared to Rounding Function block 1-1428
`fig` files
  annotating for printing 2-175
files
  reading from 1-662
  writing to
    To File block 1-1858
Filter Direct Form II block 1-1879
Find
  Index mode 1-606
  Index output format 1-604
  Mode 1-611
  Number of input dimensions 1-605
  Output data type 1-609
  Sample time (-1 for inherited) 1-608
  Scaling 1-616
  Show output port for nonzero input values 1-607
  Signedness 1-614
  Word length 1-615
Find block 1-599
`find_system` command 2-143
finding objects 2-143
finite-state machines
  implementing 1-140
First-Order Hold block 1-617
fixdt function 2-150
Fixed-Point State-Space block 1-619
fixed-point tool
  dialog pane
    Derived min/max 2-219
    Simulation min/max 2-220
Fixed-Point Tool 2-178
  dialog pane
    Apply accepted fraction lengths 2-223
    Apply accepted word lengths 2-224
    Data type override 2-203
    Data type override applies to 2-206
    Derive min/max values for selected system 2-211
    Fixed-point instrumentation mode 2-201
    Highlight results with potential issues 2-213
    Merge instrumentation results from multiple simulations 2-210
    Prepare for conversion using the Fixed-Point Advisor 2-197
    Propose fraction lengths 2-214
    Propose word lengths 2-215
    Run simulation and storeresults 2-209
    Safety margin for design and derived min/max 2-221
    Safety margin for simulation min/max 2-222
    Show details for selected result 2-225
  dialog pane overview 2-193
  main toolbar 2-180
  Show option 2-180
fixpt_evenspace_cleanup function 2-154
fixpt_interp1 function 2-156
fixpt_look1_func_approx function 2-159
fixpt_look1_func_plot function 2-164
fixptbestprec function 2-172
  autoscaling 2-173
flip-flops

- implementing 1-140
- float function 2-174
- Floating Scope block 1-1459
- `for` control flow diagram
  - creating 1-650
- For Each
  - Input Partition 1-629
- For Each block 1-625 1-630 to 1-635
- For Each Subsystem block 1-637
- For Iterator block 1-650
- For Iterator Subsystem block 1-657
- For subsystems
  - creating 1-650
- Forward Euler method 1-456
- Forward Rectangular method 1-456
- `fprintf` command 3-8
- frames for printing 2-175
- From block 1-658
- From File block 1-662
- From Workspace block 1-680
- Function-Call Feedback Latch block 1-696
- Function-Call Generator block 1-702
- Function-Call Split block 1-705
- Function-Call Subsystem block 1-708
- functions
  - fixdt 2-150
  - fixpt_evenspace_cleanup 2-154
  - fixpt_interp1 2-156
  - fixpt_look1_func_approx 2-159
  - fixpt_look1_func_plot 2-164
  - fixptbestprec 2-172
  - float 2-174
  - fxptdlg 2-178
  - num2fixpt 2-307
  - sfix 2-339
  - sfrac 2-340
  - sint 2-533
  - sldiscmdl 2-543
  - slmdldiscui 2-548
  - ufix 2-576
  - ufrac 2-577
  - uint 2-578
- fxptdlg function 2-178

G

- gain
  - varying during simulation 1-1598
- Gain block 1-709
- `gcb` command 2-243
- `gcbh` command 2-244
- `gcs` command 2-245
- `getActiveConfigSet` command 2-251
- `getConfigSet` command 2-253
- `getConfigSets` command 2-254
- `getvariable` command 2-265
- `global` Goto tag visibility 1-742
- Goto block 1-742
- Goto Tag Visibility block 1-746
- graphics
  - displaying on mask icon 3-12
- Greek letters
  - displaying on mask icons 3-4
  - using the `text` function 3-16
- Ground block 1-748
- GUI
  - Fixed-Point Tool 2-178

H

- handle of current block 2-244
- `hasvariable` command 2-266
- `help` debug command 4-22
- Hide Name menu item
  - suppressing display of port label 1-1075
- Hit Crossing block 1-751
- hybrid systems
  - linearization 2-28

I

IC block 1-756
If Action Subsystem block 1-770
If block 1-759
`if-else` control flow diagram
  creating 1-759
image
  displaying on mask icon 3-9
  drawing on mask icon using `patch` 3-11
`image` command 3-9
Increment Real World block 1-771
Increment Stored Integer block 1-773
Index Vector block 1-775
`inf` values
  in mask plotting commands 3-12
inherited
  data types
    by backpropagation 1-262
  scaling
    by backpropagation 1-262
initial conditions
  setting 1-756
Inport
  Data type 1-575 1-795 1-1922
  Icon display 1-787
  Interpolate data 1-580 1-791 1-1933
  Latch input by delaying outside signal 1-788
  Latch input for feedback signals of function-call subsystem outputs 1-789
  Output as nonvirtual bus 1-806
  Output function-call 1-792
  Port dimensions 1-571 1-1918
  Port dimensions (-1 for inherited) 1-807
  Signal type 1-811
  Variable-size signal 1-808
Inport block 1-776
Inport blocks
  in subsystem 1-1634
inports
  root-level
    connecting to buses 1-778
input ports
  unconnected 1-748
inputs
  applying expressions to 1-595
  applying MATLAB function to
    Fcn block 1-595
    Interpreted MATLAB Function block 1-855
  combining into vector line 1-1067
  delaying by variable amount 1-2026
  from outside system 1-776
  from previous time step 1-942
  from workspace 1-777
  generating step between two levels 1-1627
  interpolated mapping 1-889
  logical operations on 1-866
  multiplying block inputs during simulation 1-1598
  outputting minimum or maximum 1-980
  passing through stair-step function 1-1352
  plotting 1-2081
  reading from file 1-662
  width of 1-2076
integration
  block input 1-814 1-1473
Integrator
  Enable zero-crossing detection 1-836
  External reset 1-826
  Ignore limit and reset when linearizing 1-835
Integrator block 1-814
Integrator Limited block 1-814
interpolated mapping 1-889
Interpolation Using Prelookup block 1-840
Interpreted MATLAB Function block 1-855
Interval Test block 1-858
Interval Test Dynamic block 1-861

J

Jacobians 2-28

L

left-hand approximation 1-456
legacy_code function 2-276
limiting
    signals 1-1430
limiting derivative of signal 1-1361
limiting integral 1-816 1-1475
linearization
    discrete-time systems 2-28
linearized matrix
    eigenvalues 2-28
lines
    adding 2-7
    deleting 2-108
`linmod` function 2-285
    Transport Delay block 1-1896
`linmod2` function 2-290
`linmodv5` function 2-294
`local` Goto tag visibility 1-742
Logic
    Icon shape 1-873
    Number of input ports 1-872
    Operator 1-871
    Require all inputs and output to have the same data type 1-875
Logic and Bit Operations block library
    block parameters 6-137
logic circuits
    modeling 1-140
Logical Operator block 1-866
Lookup Table Dynamic block 1-909
Lookup Tables block library
    block parameters 6-141

M

MACs
    propagating data type information for 1-267
Magnitude-Angle to Complex block 1-917
Manual Switch block 1-924
mask icon
    displaying graphics on 3-12
    displaying image on 3-9
    displaying port label on 3-14
    displaying symbols and Greek letters on 3-16
    displaying text on 3-4
    displaying text using `fprintf` 3-8
    displaying text using `text` 3-16
    specifying color of 3-2
    using the `patch` function 3-11
mask icons
    changing plot colors on 3-2
    displaying symbols and Greek letters on 3-4
    question marks in 3-12
masked blocks
    parameters 6-260
masked subsystems
    question marks in icon 3-12
masking bits 1-50
Math block library
    block parameters 6-158
Math Function block 1-928
mathematical functions
    performing
        Math Function block 1-928
        Rounding Function block 1-1427
        Trigonometric Function block 1-1940
mathematical symbols
    displaying on mask icons 3-4
    displaying on mask icons using `text` 3-16
MATLAB character encoding, changing 2-537
MATLAB functions
    applying to block input
        Fcn block 1-595

        Interpreted MATLAB Function block 1-855
- matrices
    - writing to 1-1863
- Matrix Concatenate
    - Concatenate dimension 1-2061
    - Mode 1-2060
    - Number of inputs 1-2059
- Matrix Concatenate block 1-2054
- Memory block 1-942
- memory region
    - shared
        - Data Store Memory block 1-203
        - Data Store Read block 1-218
        - Data Store Write block 1-224
- Merge block 1-957
- MinMax block 1-980
- MinMax Running Resettable block 1-988
- Model block 1-990
- Model Info block 1-1019
- model parameters
    - table 6-2
- Model Reference 1-2049
    - Condition 1-1005 1-2050
    - Enable variants 1-999 1-1014
    - Generate preprocessor conditionals 1-1014
    - Model argument values (for this instance) 1-996 1-1009
    - Model arguments 1-995 1-1008
    - Model name 1-994 1-1006 to 1-1007
    - Override variant conditions 1-1012
    - Simulation mode 1-997 1-1010
    - Variant 1-1013
    - Variant choices table 1-1001
    - Variant object 1-1004
- Model Verification block library
    - block parameters 6-183
- Model-Wide Utilities block library
    - block parameters 6-187
- models
    - closing 2-22
    - creating
        - `new_system` command 2-304
    - getting name 2-24
    - parameters 6-2
    - replacing blocks 2-322
- multiplying block inputs
    - during simulation 1-1598
- Multiport Switch block 1-1023
- MultiPortSwitch
    - Allow different data input sizes (Results in variable-size output signal) 1-1047
    - Bias 1-1060
    - Data port for default case 1-1038
    - Data port indices 1-1036
    - Data port order 1-1033
    - Diagnostic for default case 1-1039
    - Fraction length 1-1058
    - Mode 1-1052
    - Number of data ports 1-1035
    - Output data type 1-1050
    - Output maximum 1-1049
    - Output minimum 1-1048
    - Require all data port inputs to have the same data type 1-1041
    - Signedness 1-1055
    - Slope 1-1059
    - Word length 1-1056
- multirate systems
    - linearization 2-28
- Mux
    - Display option 1-1072
    - Number of inputs 1-1071
- Mux block 1-1067

N

- n-D Lookup Table block 1-889
- `NaN` values
    - in mask plotting commands 3-12

`nanbreak` debug command 4-23
`new_system` command 2-304
`next` debug command 4-24
nonlinear systems
  spectral analysis of 1-135
normally distributed random numbers 1-1358
num2fixpt function 2-307

O

objects
  finding 2-143
obsolete blocks, replacing 2-562
`ode113` solver
  Memory block 1-942
`ode15s` solver
  Memory block 1-942
`open_system` command 2-310
`openDialog` command 2-314
opening
  block dialog boxes 2-310
  Simulink Library Browser 2-375
  system windows 2-310
operating point 2-285
Outport
  Data type 1-1091
  Initial output 1-1089
  Output as nonvirtual bus in parent model 1-1102
  Output when disabled 1-1088
  Port dimensions (-1 for inherited) 1-1104
  Sampling mode 1-1109
  Signal type 1-1108
  Source of initial output value 1-1087
  Variable-size signal 1-1105
Outport block 1-1075
Outport blocks
  in subsystem 1-1634
output
  outside system 1-1075
  selected elements of input vector 1-1508
  selected information about the signal on input 1-1297
  switching between two inputs 1-924
  values
    displaying 1-545
  writing to file
    To File block 1-1858
  writing to workspace
    To Workspace block 1-1863
  zero within range 1-276
output ports
  capping unconnected 1-1754

P

parameters
  adding 2-9
  deleting 2-110
  masked blocks 6-260
  model 6-2
Passing arguments by reference as read input or write output 2-73
Passing arguments by reference as read-only input 2-75
Passing arguments by reference as write-only output 2-99
`patch` command 3-11
Permute Dimensions block 1-1111
phase-shifted wave 1-1539
PID Controller
  1 degree-of-freedom 1-1114
  2 degree-of-freedom 1-1195
piecewise linear signal
  generating
    Signal Builder block 1-1539
`plot` command 3-12
plotting input signals
  Scope block 1-1459
  XY Graph block 1-2081

- Polynomial block 1-1280
- port label
  - displaying on mask icon 3-14
- port labels
  - suppressing display 1-1075
- `port_label` command 3-14
- Ports & Subsystems block library
  - block parameters 6-189
- precision
  - maximum 2-172
- Prelookup block 1-1283
- print frames 2-175
- `printframe` 2-175
- PrintFrame Editor 2-175
- printing
  - borders 2-175
  - `fig` files with frames 2-175
  - with print frames 2-177
- Probe block 1-1297
- `probe` debug command 4-25
- Product
  - Dimension 1-1317
  - Multiplication 1-1315
  - Multiply over 1-1316
  - Number of inputs 1-1313
- Product block 1-1303
- Product of Elements block 1-1338
- programmable logic arrays
  - modeling 1-140
- propagation of data types 1-262
- properties
  - block
    - list 6-96
- Pulse Generator block 1-1346

Q

- Quantizer block 1-1352
- question marks in mask icon 3-12
- `quit` debug command 4-27

R

- Ramp block 1-1355
- random noise
  - generating 1-1358
- Random Number block 1-1358
  - and Band-Limited White Noise block 1-37
  - compared to Band-Limited White Noise block 1-1358
- random numbers
  - generating normally distributed 1-37
  - normally distributed 1-1358
  - uniformly distributed 1-1959
- Rate Limiter block 1-1361
- Rate Limiter Dynamic block 1-1365
- Rate Transition block 1-1367
- reading data
  - from data store 1-218
  - from file 1-662
  - from workspace 1-680
- Real-Imag to Complex block 1-1377
- Reciprocal Sqrt block 1-1600
- region of zero output 1-276
- regular expressions 2-147
- Relational Operator
  - Relational operator 1-1386
- Relational Operator block 1-1380
- Relay block 1-1398
- Repeating Sequence block 1-1406
- Repeating Sequence Interpolated block 1-1410
- Repeating Sequence Stair block 1-1415
- repeating signals 1-1406
- replace obsolete blocks 2-562
- `replace_block` command 2-322
- replacing blocks in model 2-322
- Resettable Delay block 1-1421
- Reshape block 1-1422
- right-hand approximation 1-457
- root-level inports
  - connecting to buses 1-778
- Rounding Function block 1-1427

S

S-Function block 1-1516
S-Function Builder block 1-1519
sample-and-hold
  applying to block input 1-942
sampling interval
  generating simulation time 1-360
Saturate
  Lower limit 1-1435
  Treat as gain when linearizing 1-1436
  Upper limit 1-1434
Saturation block 1-1430
Saturation Dynamic block 1-1454
`save_system` command 2-324
sawtooth wave
  generating 1-1549
Scope block 1-1459
`scoped` Goto tag visibility 1-742
Second-Order Integrator block 1-1473
Second-Order Integrator Limited block 1-1473
SecondOrderIntegrator
  Absolute tolerance dx/dt 1-1501
  Absolute tolerance x 1-1494
  External reset 1-1503
  Ignore state limits and the reset for linearization 1-1506
  Initial condition dx/dt 1-1497
  Initial condition source dx/dt 1-1496
  Initial condition source x 1-1489
  Initial condition x 1-1490
  Limit dx/dt 1-1498
  Limit x 1-1491
  Lower limit dx/dt 1-1500
  Lower limit x 1-1493
  Reinitialize dx/dt when x reaches saturation 1-1505
  Show output 1-1507
  State name dx/dt 1-1502
  State name x 1-1495
  Upper limit dx/dt 1-1499
  Upper limit x 1-1492
Selector block 1-1508
separating vector signal 1-310
sequence of signals 1-1346
sequential circuits
  implementing 1-142
`set_param` command 2-333
`setActiveConfigSet` command 2-336
setting bits 1-50
setting parameter values 2-333
`sfix` function 2-339
`sfrac` function 2-340
shared data store
  Data Store Memory block 1-203
  Data Store Read block 1-218
  Data Store Write block 1-224
Shift Arithmetic block 1-1520
Sign block 1-1535
Signal Attributes block library
  block parameters 6-224
Signal Builder block 1-1539
Signal Conversion block 1-1541
Signal Generator block 1-1548
Signal Routing block library
  block parameters 6-232
Signal Specification
  Require nonvirtual bus 1-1575
Signal Specification block 1-1557
  Bias 1-1574
  Data type 1-1563
  Dimensions (-1 for inherited) 1-1577
  Fraction length 1-1572
  Maximum 1-1562
  Mode 1-1566
  Sample time (-1 for inherited) 1-1579
  Sampling mode 1-1581
  Signal type 1-1580
  Signedness 1-1569
  Slope 1-1573
  Variable-size signal 1-1578

- Word length 1-1570

signals
- displaying graphically 1-1459
- displaying X-Y plot of 1-2081
- generating pulses 1-1346
- limiting 1-1430
- limiting derivative of 1-1361
- passed from Goto block 1-658
- passing to From block 1-742
- plotting
  - Scope block 1-1459
  - XY Graph block 1-2081
- repeating 1-1406

SignalSpecification
- Minimum 1-1561

Signed Sqrt block 1-1600

simulation
- stopping
  - Stop Simulation block 1-1630

simulation commands
- Simulink.Block.getSampleTimes 2-381
- Simulink.BlockDiagram.getChecksum 2-395
- Simulink.BlockDiagram.getInitialState 2-398
- Simulink.BlockDiagram.getSampleTimes 2-400
- Simulink.SubSystem.getChecksum 2-530

simulation time
- generating at sampling interval 1-360
- outputting 1-138

Simulink
- printing diagram with frames 2-175

simulink classes
- Simulink.AliasType 5-7
- Simulink.Bus 5-44
- Simulink.BusElement 5-49
- Simulink.ModelDataLogs 5-116
- Simulink.ModelWorkspace 5-125
- Simulink.NumericType 5-147
- Simulink.ParamCoderInfo 5-168
- Simulink.Parameter 5-161
- Simulink.Signal 5-222
- Simulink.SignalCoderInfo 5-234
- Simulink.SubsysDataLogs 5-317
- Simulink.TimeInfo 5-319
- Simulink.TsArray 5-322

`simulink` command 2-375

Simulink Library Browser
- opening 2-375

Simulink.AliasType 5-7

Simulink.Block.getSampleTimes command
- description 2-381

`Simulink.BlockDiagram.addBusToVector` command 2-383

`Simulink.BlockDiagram.copyContentsToSubSystem` command 2-389

`Simulink.BlockDiagram.createSubSystem` command 2-391

`Simulink.BlockDiagram.deleteContents` command 2-394

Simulink.BlockDiagram.getChecksum command
- description 2-395

Simulink.BlockDiagram.getInitialState command
- description 2-398

Simulink.BlockDiagram.getSampleTimes command
- description 2-400

Simulink.Bus 5-44

`Simulink.Bus.cellToObject` command 2-405

`Simulink.Bus.createMATLABStruct` command 2-406

`Simulink.Bus.createObject` command 2-419

`Simulink.Bus.objectToCell` command 2-422

`Simulink.Bus.save` command 2-423

Simulink.BusElement 5-49

`Simulink.defineIntEnumType` command 2-425

Simulink.ModelDataLogs 5-116

Simulink.ModelWorkspace 5-125

Simulink.NumericType 5-147

Simulink.ParamCoderInfo 5-168

Simulink.Parameter 5-161

Simulink.Signal 5-222
Simulink.SignalCoderInfo 5-234
`Simulink.SimulationData.createStructOfTimeseries`
  command 2-501
`Simulink.SimulationData.signalLoggingSelector`
  command 2-511
`Simulink.SimulationData.updateDatasetFormatLogging`
  command 2-512
Simulink.SubsysDataLogs 5-317
`Simulink.SubSystem.convertToModelReference`
  command 2-523
`Simulink.SubSystem.copyContentsToBlockDiagram`
  command 2-527
`Simulink.SubSystem.deleteContents`
  command 2-529
Simulink.SubSystem.getChecksum command
  description 2-530
`Simulink.SubSystemModelDataLlogs.convertToDataset`
  command 2-434
Simulink.TimeInfo 5-319
Simulink.TsArray 5-322
Sine and Cosine block 1-1583
Sine block 1-1583
sine wave
  generating
    Signal Generator block 1-1549
    Sine Wave block 1-1589
  generating with increasing frequency
    Chirp Signal block 1-135
Sine Wave block 1-1589
Sinks block library
  block parameters 6-241
`sint` function 2-533
sldiscmdl function 2-543
Slider Gain block 1-1598
slmdldiscui function 2-548
`slreplace_mux` command 2-560
`slupdate` command 2-562
Sources block library
  block parameters 6-245
spectral analysis of nonlinear systems 1-135
Sqrt block 1-1600
square root functions
  performing 1-1600
square wave
  generating 1-1549
Squeeze block 1-1615
`ss2tf` function 2-29 2-288
`ss2zp` function 2-29 2-288
stair-step function
  passing signal through 1-1352
state derivatives
  setting to zero 2-568
state space in discrete system 1-448
State-Space block 1-1617
  A 1-1619
  B 1-1620
  C 1-1621
  D 1-1622
  Initial conditions 1-1623
Stateflow
  printing diagram with frames 2-175
states
  resetting 1-817 1-1481
  specifying absolute tolerance for 1-823 1-1483
`status` debug command 4-33
Step block 1-1627
`step` debug command 4-35
`stop` debug command 4-38
Stop Simulation block 1-1630
stopping simulation 1-1630
`strace` debug command 4-39
Subsystem
  Code generation file name (no extension) 1-1655
  Code generation file name options 1-1653
  Code generation function name 1-1652
  Code generation function name options 1-1650

  - Code generation function packaging 1-1647
  - Function with separate data 1-1656
  - Memory section for constants 1-1663
  - Memory section for execution functions 1-1661
  - Memory section for initialize/terminate functions 1-1659
  - Memory section for internal data 1-1664
  - Memory section for parameters 1-1665
  - Minimize algebraic loop occurrences 1-1643
  - Name of error callback function 1-1640
  - Permit hierarchical resolution 1-1641
  - Propagate execution context across subsystem boundary 1-1644
  - Read/Write permissions 1-1639
  - Sample time (-1 for inherited) 1-1646
  - Show port labels 1-1638
  - Treat as atomic unit 1-1642
  - Warn if function-call inputs are context-specific 1-1645
- SubSystem
  - Allow different input sizes (Results in variable-size output signal) 1-926
  - Generate preprocessor conditionals 1-2053
  - Name (read-only) 1-2048
  - Override variant conditions and use following variant 1-2051
  - Variant 1-2052
  - Variant choices (list of child subsystems) 1-2046
- Subsystem block 1-1633
- subsystems
  - and Inport blocks 1-777
  - enabled 1-563
- Subtract block 1-1667
- Sum
  - Dimension 1-1676
  - Icon shape 1-1673
  - List of signs 1-1674
  - Sum over 1-1675
- Sum block 1-1667
- Sum of Elements block 1-1667
- Switch
  - Allow different data input sizes (Results in variable-size output signal) 1-1726
  - Criteria for passing first input 1-1715
  - Threshold 1-1717
- Switch block 1-1710
- Switch Case Action Subsystem block 1-1750
- Switch Case block 1-1743
- `switch` control flow diagram
  - creating 1-1743
- switching output between inputs
  - Manual Switch block 1-924
- switching output between two inputs 1-924
- system windows
  - closing 2-31
- systems
  - current 2-245
  - saving 2-324
- `systems` debug command 4-42

T

- Tapped Delay block 1-1751
- `tbreak` debug command 4-43
- Terminator block 1-1754
- terminators
  - adding 2-12
- TeX formatting commands
  - using in mask icon text 3-16
  - using with `disp` 3-4
- `text` command 3-16
- `tf2ss` utility
  - converting Transfer Fcn to state-space form 1-1871
- time delay
  - simulating 1-1896
- Time Scope block 1-1760
- Time-Based Linearization block 1-1756

To File block 1-1858
To Workspace block 1-1863
`trace` debug command 4-44
Transfer Fcn block 1-1870
  Denominator coefficients 1-1875
  Numerator coefficients 1-1874
Transfer Fcn Direct Form II Time Varying block 1-1882
Transfer Fcn First Order block 1-1886
Transfer Fcn Lead or Lag block 1-1889
Transfer Fcn Real Zero block 1-1892
transfer function form
  converting to 2-29 2-288
transfer functions
  discrete 1-518
  linear 1-1870
  poles and zeros 1-2093
    discrete 1-539
Transport Delay block 1-1896
  Direct feedthrough of input during linearization 1-1903
  Initial buffer size 1-1901
  Initial output 1-1900
  Pade order (for linearization) 1-1904
  Time delay 1-1899
  Use fixed buffer size 1-1902
Trapezoidal method 1-458
Trigger block 1-1905
Trigger-Based Linearization block 1-1935
Triggered Subsystem block 1-1939
triggered subsystems
  Trigger block 1-1905
TriggerPort
  Enable zero-crossing detection 1-1917
  Output data type 1-1914
  Propagate sizes of variable-size signals 1-1912
  Sample time 1-1916
  Sample time type 1-1915
  Show output port 1-1913
  States when enabling 1-1910
  Trigger type 1-1909
Trigonometric Function block 1-1940
`trim` function 2-568
truth tables
  implementing 1-140

U

`ufix` function 2-576
`ufrac` function 2-577
`uint` function 2-578
Unary Minus block 1-1956
unconnected input ports 1-748
unconnected output ports
  using the Terminator block 1-1754
`undisp` debug command 4-45
Uniform Random Number block 1-1959
  compared to Band-Limited White Noise block 1-1959
uniformly distributed random numbers 1-1959
Unit Delay
  Initial conditions 1-1971
  Input processing 1-1972
Unit Delay block 1-1962
  compared to Transport Delay block 1-1896
Unit Delay Enabled block 1-1983
Unit Delay Enabled External IC block 1-1986
Unit Delay Enabled Resettable block 1-1989
Unit Delay Enabled Resettable External IC block 1-1993
Unit Delay External IC block 1-1996
Unit Delay Resettable block 1-1999
Unit Delay Resettable External IC block 1-2002
Unit Delay With Preview Enabled block 1-2005
Unit Delay With Preview Enabled Resettable block 1-2009
Unit Delay With Preview Enabled Resettable External RV block 1-2013
Unit Delay With Preview Resettable block 1-2017

Unit Delay With Preview Resettable External
  RV block 1-2021
`untrace` debug command 4-46
Update Diagram menu item
  changing block parameters during
    simulation 2-333
User-defined functions block library
  block parameters 6-254

V

Variable Integer Delay block 1-2025
variable time delay 1-2026
Variable Time Delay block 1-2026
  Direct feedthrough of input during
    linearization 1-2038
  Handle zero delay 1-2037
  Initial buffer size 1-2035
  Initial output 1-2034
  Maximum delay 1-2033
  Pade order (for linearization) 1-2039
  Select delay type 1-2032
  Use fixed buffer size 1-2036
Variable Transport Delay block 1-2026
  Absolute tolerance 1-2040
  Direct feedthrough of input during
    linearization 1-2038
  Initial buffer size 1-2035
  Initial output 1-2034
  Maximum delay 1-2033
  Pade order (for linearization) 1-2039
  Select delay type 1-2032
  State Name 1-2040
  Use fixed buffer size 1-2036
`vdp` model
  Scope block 1-1470
Vector Concatenate
  Concatenate dimension 1-2061
  Mode 1-2060
  Number of inputs 1-2059
Vector Concatenate block 1-2054
vector signals
  generating from inputs 1-1067
  separating 1-310
viscous friction 1-191
visibility of Goto tag 1-746

W

Weighted Sample Time block 1-2062
Weighted Sample Time Math block 1-2063
`while` control flow diagram
  creating 1-2069
While Iterator block 1-2069
While Iterator Subsystem block 1-2075
While subsystems
  creating 1-2069
white noise
  generating 1-37
Width block 1-2076
workspace
  reading data from 1-680
  writing output to 1-1863
workspace variables
  determine if variable exists 2-266
  getting values of 2-265
Wrap To Zero block 1-2079
writing data to data store 1-224
writing output to file 1-1858
writing output to workspace 1-1863

X

`xbreak` debug command 4-49
XY Graph block 1-2081

Z

`zcbreak` debug command 4-50
`zclist` debug command 4-51
zero crossings

detecting
    Hit Crossing block 1-751
zero output in region
    generating 1-276
Zero-Order Hold block 1-2086
Zero-Pole block 1-2093
    Gain 1-2099
    Poles 1-2098
    Zeros 1-2097
zero-pole form
    converting to 2-29 2-288