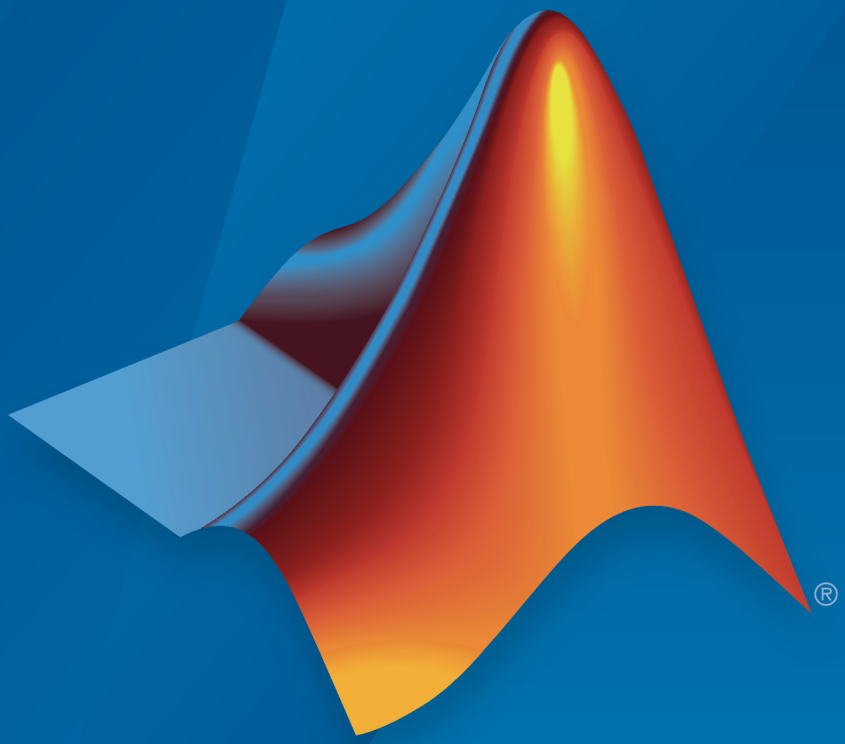
Simulink®

Reference



MATLAB® & SIMULINK®

R2016a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Simulink® Reference

Revision History

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

Contents

7 Fixed-Point Tool

8 Model Advisor Checks

9 Model Reference Conversion Advisor

10 Performance Advisor Checks

11 Simulink Limits

Blocks — Alphabetical List

Abs

Output absolute value of input

Library

Math Operations

Description

The Abs block outputs the absolute value of the input.

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

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

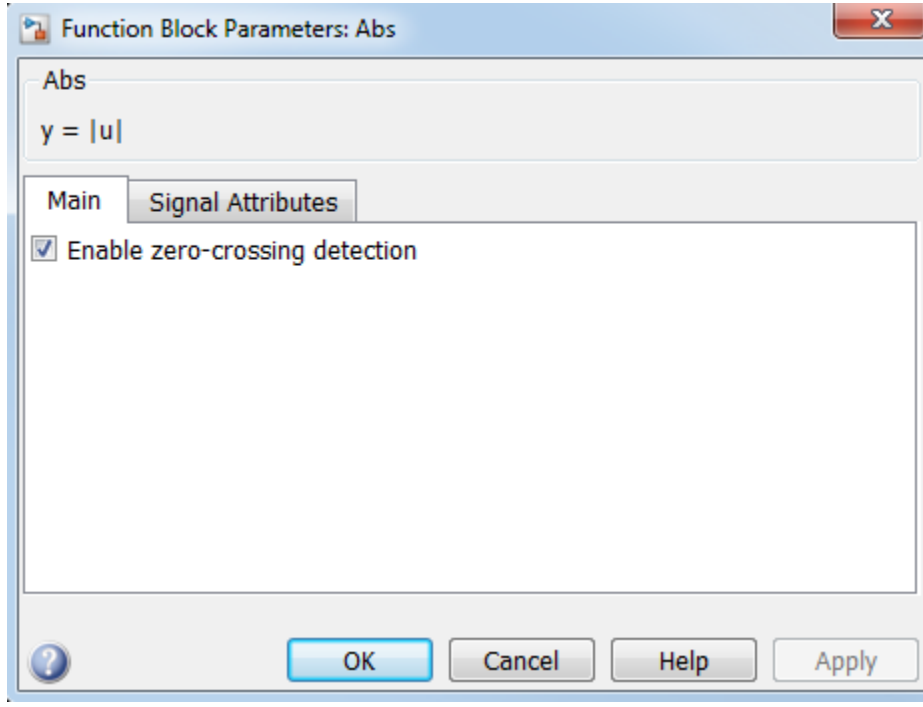
The `Abs` block accepts real signals of the following data types:

- 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:



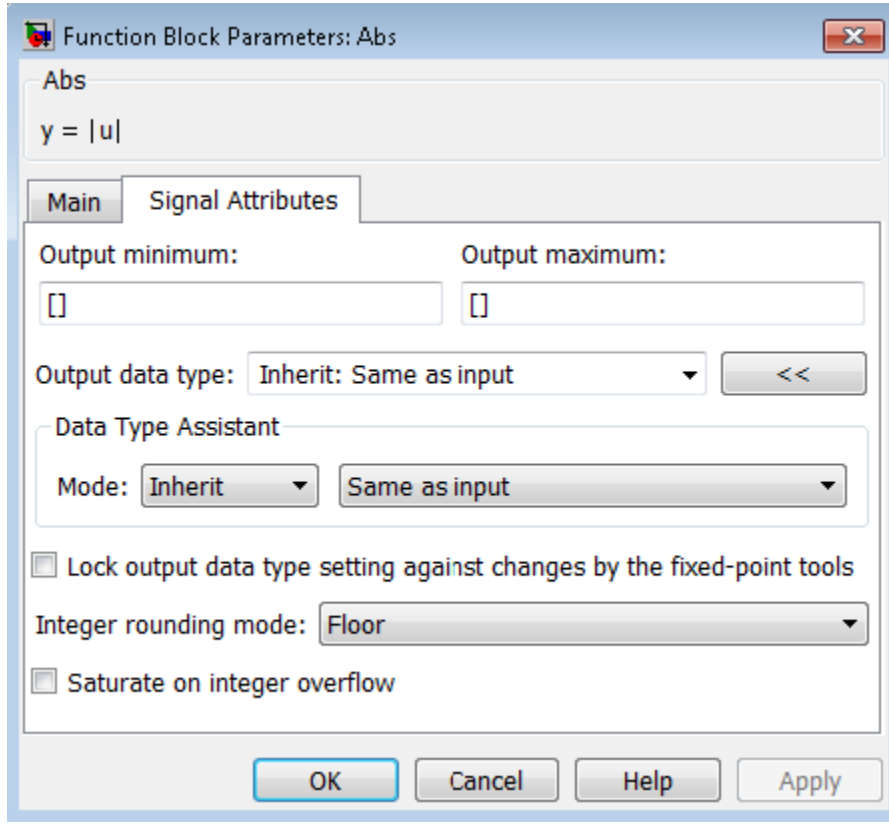
Enable zero-crossing detection

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

Sample time

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

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



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

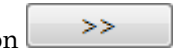
Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- 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 "Control Signal Data Types" in Simulink User's Guide for more information.

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

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

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

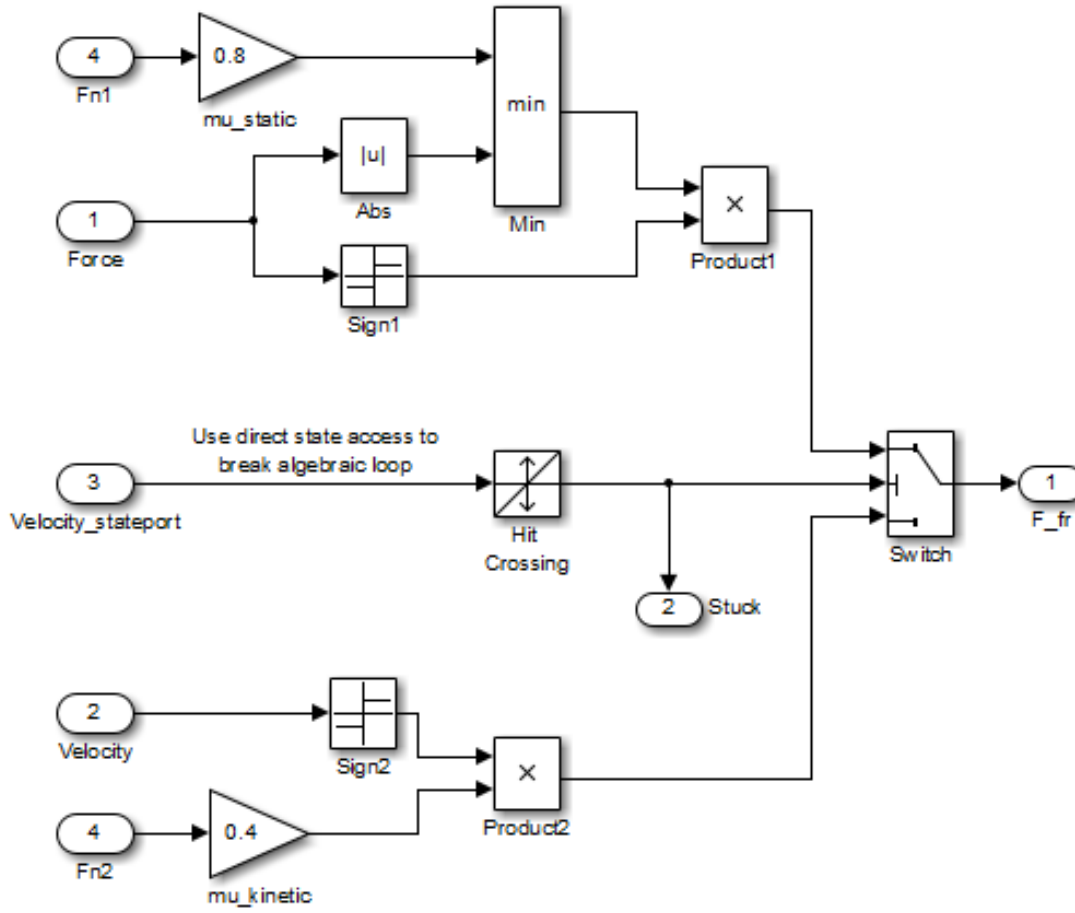
detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

Usage as an Input to a MinMax Block

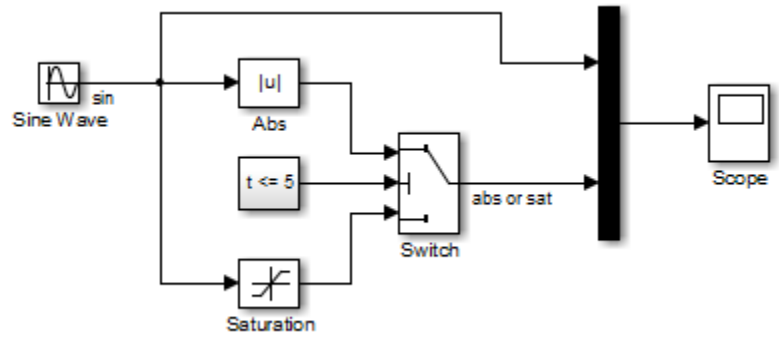
The `sldemo_hardstop` model shows how you can use the `Abs` block as an input to the `MinMax` block.

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



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.



Characteristics

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

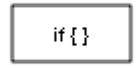
Introduced before R2006a

Action Port

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

Library

Ports & Subsystems



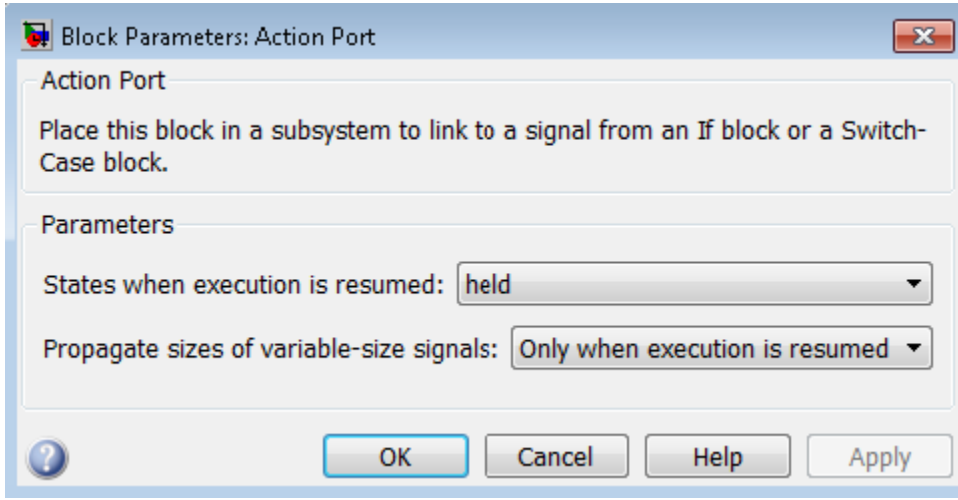
Description

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

Data Type Support

Action Port blocks do not have data inputs or outputs.

Parameters and Dialog Box



States when execution is resumed

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

Settings

Default: `held`

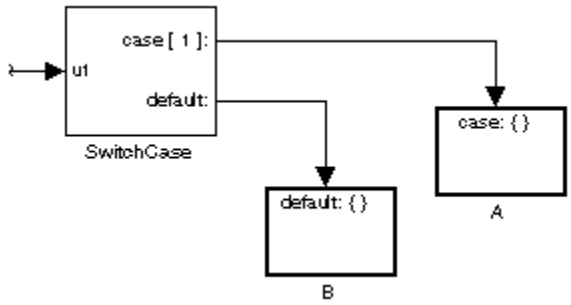
`held`

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`, `Switch Case Action Subsystem`

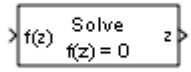
Introduced before R2006a

Algebraic Constraint

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).

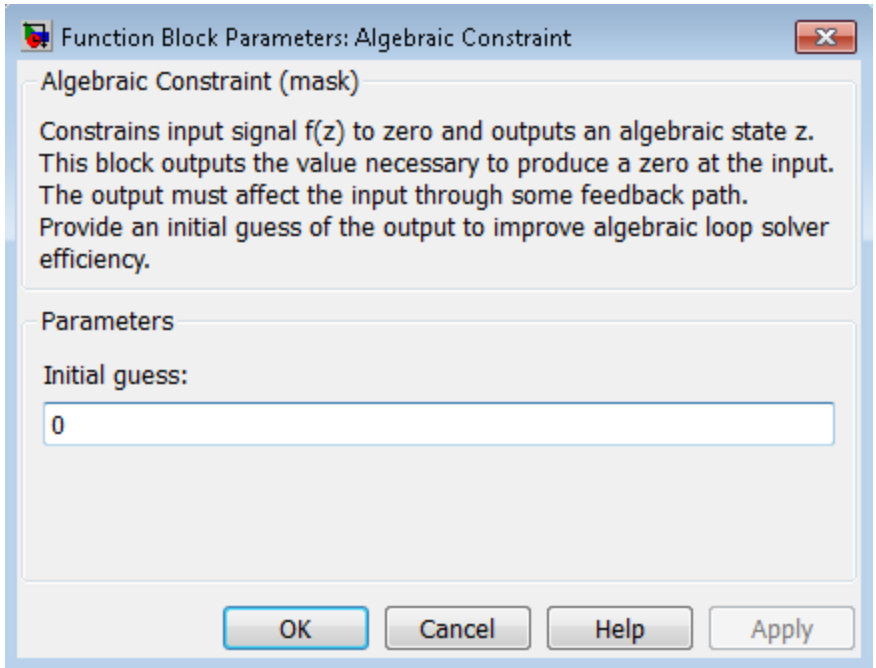
Algorithm

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`.

Parameters and Dialog Box



Initial guess

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

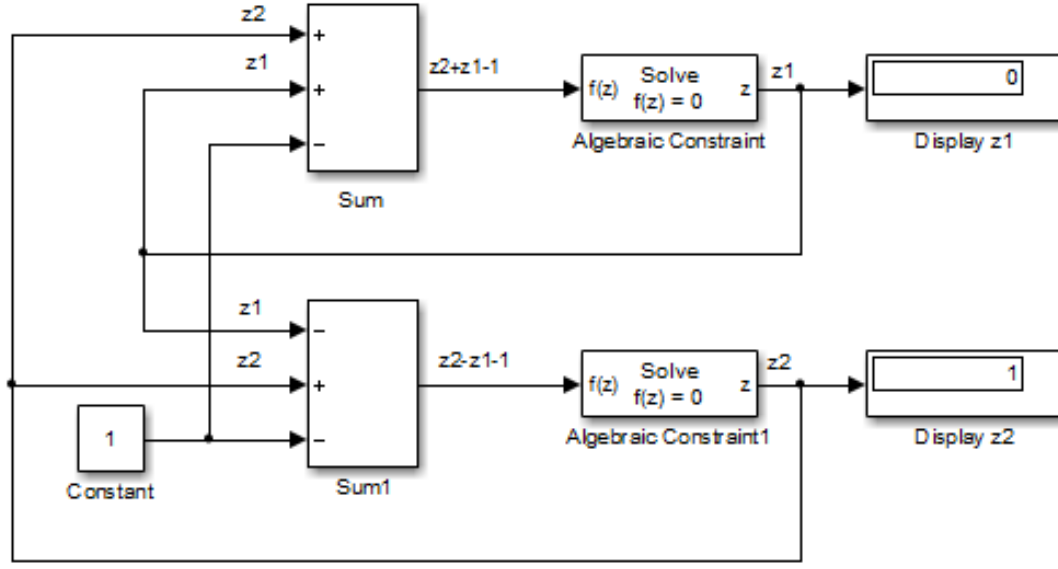
Example

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.



Characteristics

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

References

[1] Garbow, B. S., K. E. Hillstrom, and J. J. Moré. *User Guide for MINPACK-1*. Argonne, IL: Argonne National Laboratory, 1980.

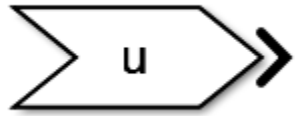
[2] Rabinowitz, P. H. *Numerical Methods for Nonlinear Algebraic Equations*. New York, NY: Gordon and Breach, 1970.

Introduced before R2006a

Argument Inport

Argument input port for Simulink Function block

Description



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

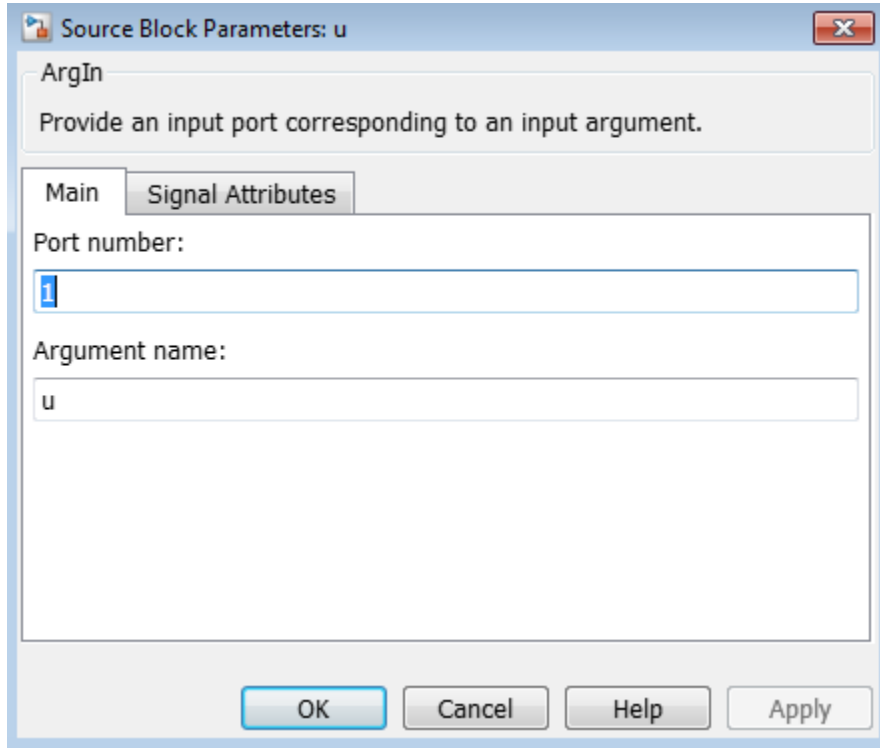
Data Type Support

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

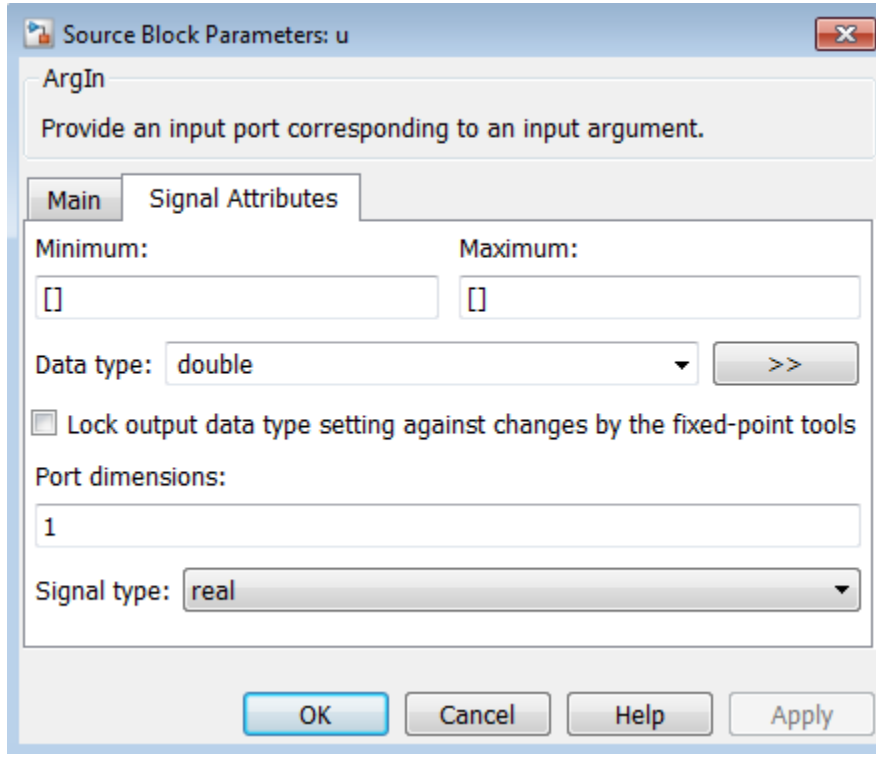
The numeric and data types of the block output are the same as those of its input. You can specify the signal type and data type of an input argument to an Argument Inport block using the **Signal type** and **Data type** parameters. For more information, see "Data Types Supported by Simulink".

Parameters and Dialog Box

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



The **Signal Attributes** pane of the `Argument Inport` block dialog box appears as follows:



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 in the parent subsystem or model block.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Argument Name

Settings

Default: u

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

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Minimum

Specify the minimum value for the block to 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 6-101 for the command-line information.

Maximum

Specify the maximum value for the block to 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 6-101 for the command-line information.

Data type

Specify the output data type of the argument 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)`.

`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 6-101 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 “Control Signal Data Types”.

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 6-101 for the command-line information.

See Also

For more information, 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.

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.

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**

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.

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.

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.

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 6-101 for the command-line information.

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

Signal type

Specify the numeric type of the argument input.

Settings

Default: `real`

`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 6-101 for the command-line information.

Characteristics

Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Argument Outport

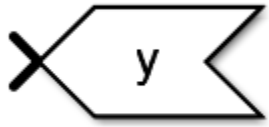
Simulink Function

Introduced in R2014b

Argument Outport

Argument output port for Simulink Function block

Description



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

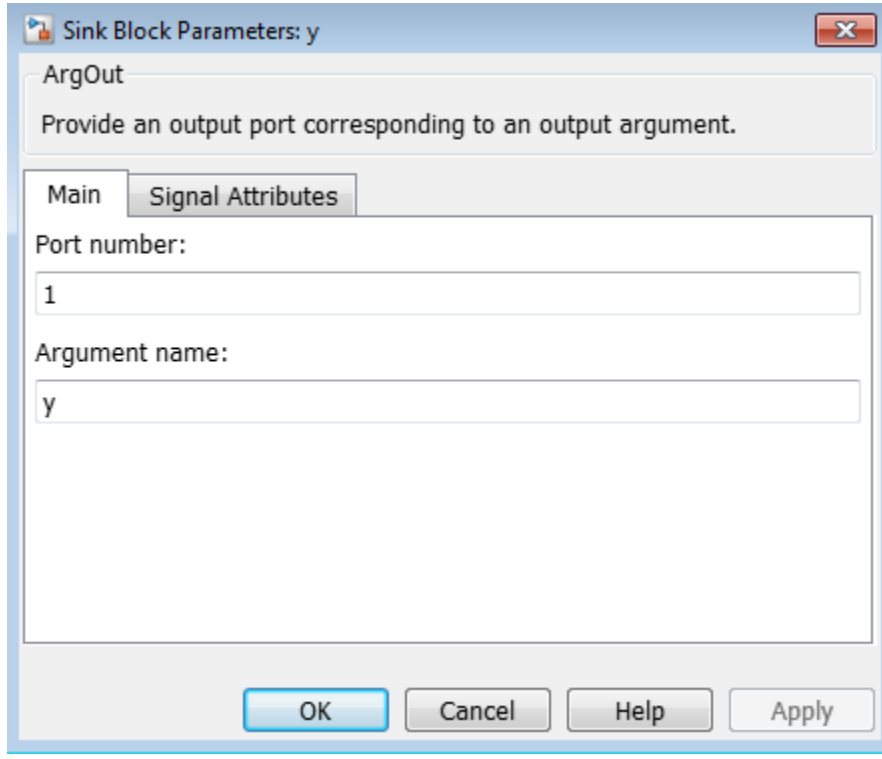
Data Type Support

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

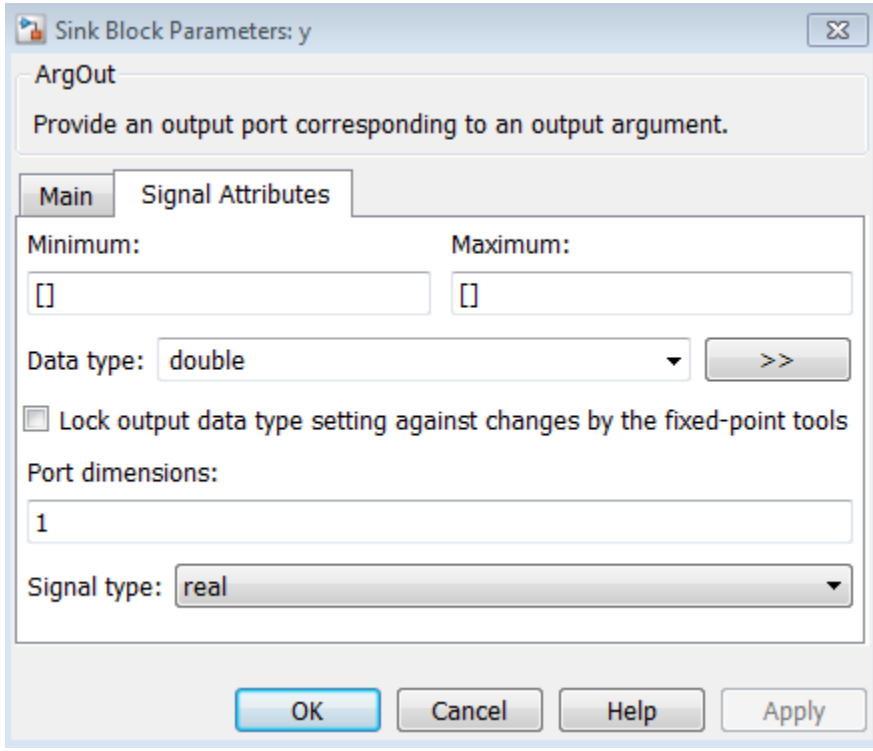
For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box

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



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



- "Port number" on page 1-42
- "Argument Name" on page 1-42
- "Minimum" on page 1-43
- "Maximum" on page 1-44
- "Data type" on page 1-44
- "Show data type assistant" on page 1-46
- "Mode" on page 1-46
- "Data type override" on page 1-47
- "Signedness" on page 1-49
- "Word length" on page 1-49
- "Scaling" on page 1-50
- "Fraction length" on page 1-50

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 6-101 for the command-line information.

Argument Name

Settings

Specify the name of the output argument.

Default: y

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

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Minimum

Specify the minimum value for the block to 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 6-101 for the command-line information.

Maximum

Specify the maximum value for the block to 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 6-101 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)`.

`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 6-101 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 "Control Signal Data Types".

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 6-101 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.

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.

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**

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.

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.

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.

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

See Also

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

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.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Port dimensions

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

Settings

Default: 1

Valid values are:

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

Dependency

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

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Signal type

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

Settings

Default: `real`

`real`

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

`complex`

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

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Characteristics

Dimensionalized	Yes

Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Argument Inport

Simulink Function

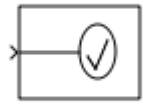
Introduced in R2014b

Assertion

Check whether signal is zero

Library

Model Verification



Description

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

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

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

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

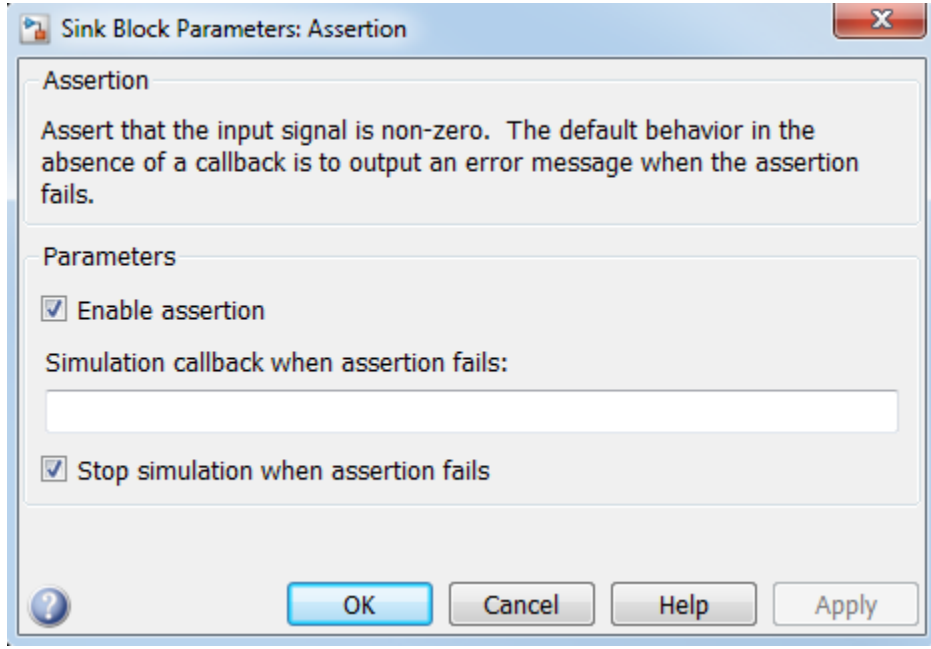
Note: For information about how Simulink Coder™ generated code handles Model Verification blocks, see "Debug".

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.

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 on the **All Parameters** tab in the Configuration Parameters dialog box lets you enable or disable all Assertion blocks in a model regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

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

Sample time

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

Characteristics

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

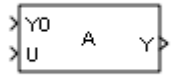
Introduced before R2006a

Assignment

Assign values to specified elements of signal

Library

Math Operations



Description

The Assignment block assigns values to specified elements of the signal. You can specify the indices of the elements to be assigned values either by entering the indices in the block'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.

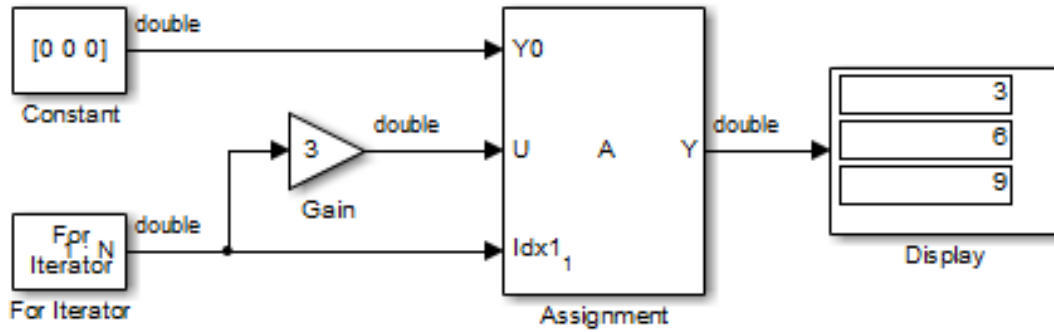
When using the Assignment block in Normal mode, Simulink initializes block outputs to zero even if the model does not explicitly initialize them. In Accelerator mode, Simulink converts the model into an S-Function. This involves code generation. The code generated may not do implicit initialization of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

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

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

Iterated Assignment

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



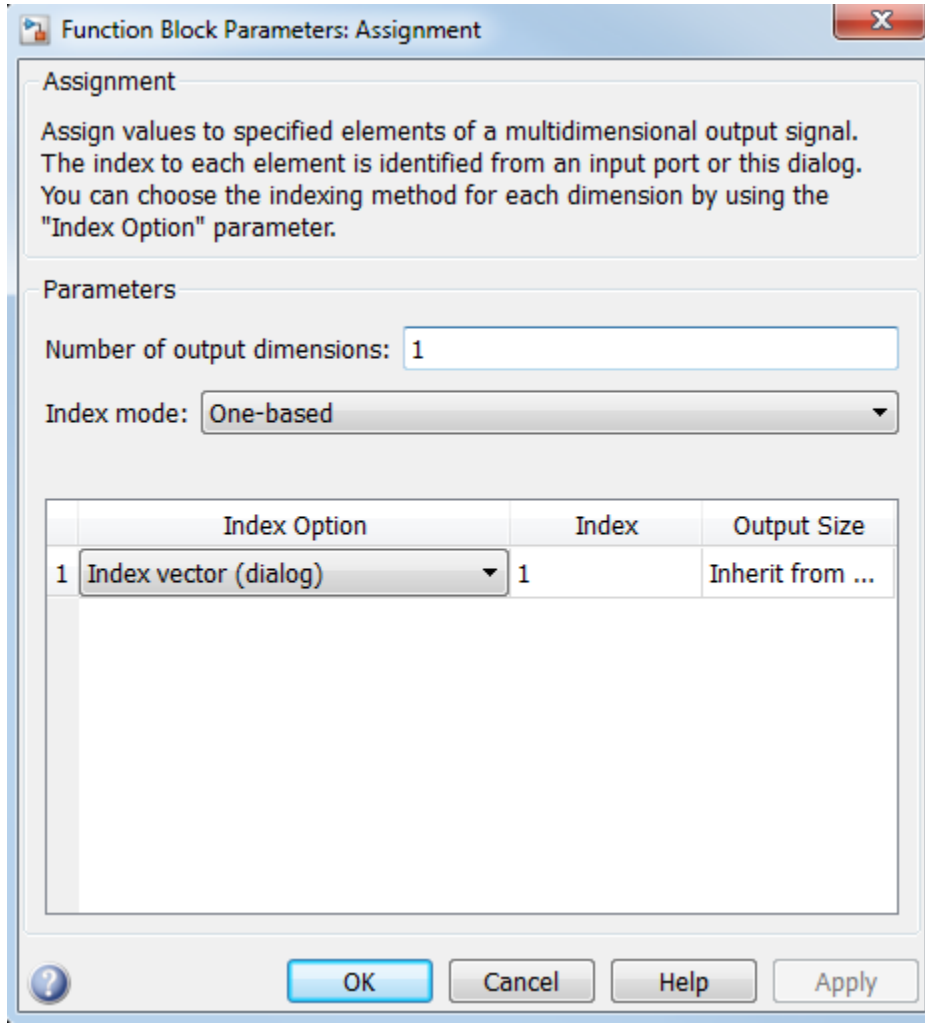
Iterated assignment uses an iterator (For or While) block to generate indices for the Assignment block. On the first iteration of an iterated assignment, the Assignment block copies the first input (Y0) to the output (Y) and assigns the second input (U) to the output $Y(E_1)$. On successive iterations, 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, enumerated, and nonvirtual bus data types. The external indices port accepts any built-in data type, except Boolean data types.

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

Parameters 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

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

Characteristics

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

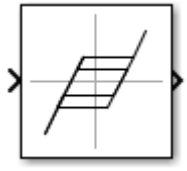
Introduced before R2006a

Backlash

Model behavior of system with play

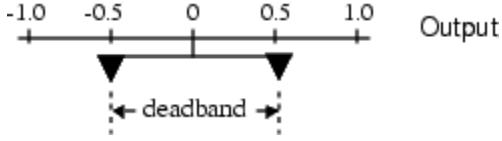
Library

Discontinuities



Description

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



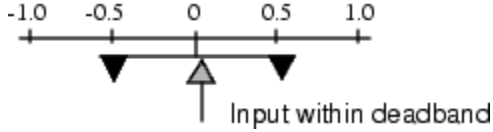
A system with play can be in one of three modes:

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

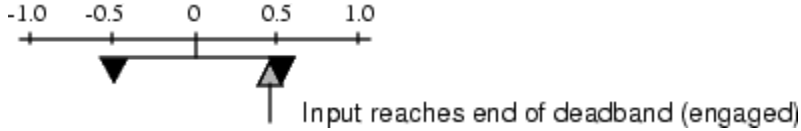
If the initial input is outside the deadband, the **Initial output** parameter value determines whether the block is engaged in a positive or negative direction, and the output at the start of the simulation is the input plus or minus half the deadband width.

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

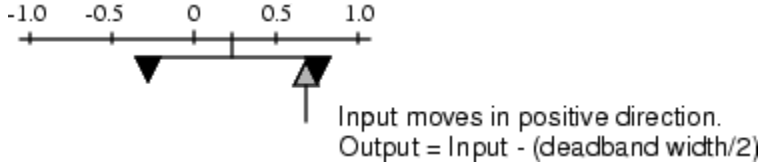
The following figures illustrate the block's operation when the initial input is within the deadband. The first figure shows the relationship between the input and the output while the system is in disengaged mode (and the default parameter values are not changed).



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



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



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

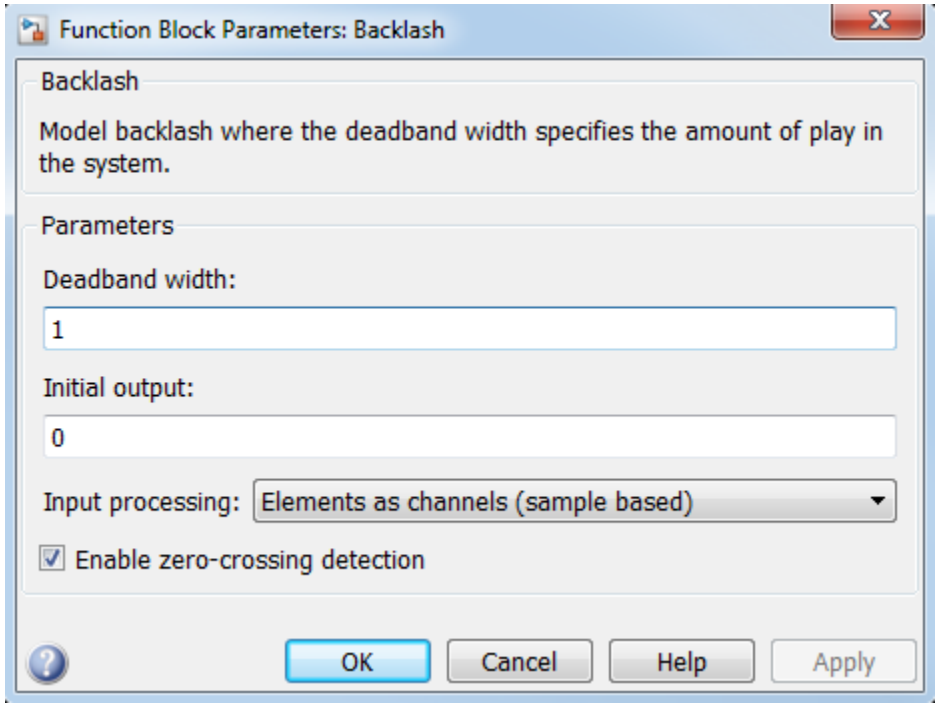
For example, if the deadband width is 2 and the initial output is 5, the output, y, at the start of the simulation is as follows:

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

Data Type Support

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

Parameters 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

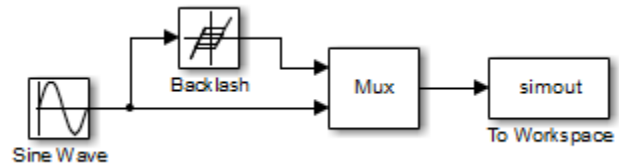
Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Sample time

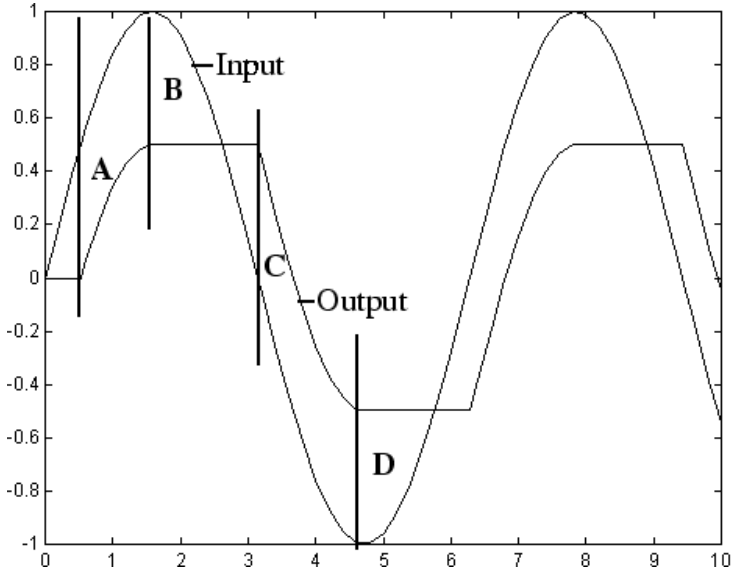
> **Note:** This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see "Blocks for Which Sample Time Is Not Recommended".

Examples

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



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



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

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

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

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

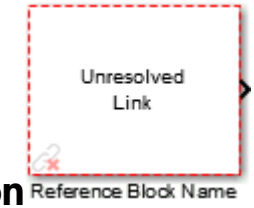
Characteristics

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

Introduced before R2006a

Unresolved Link

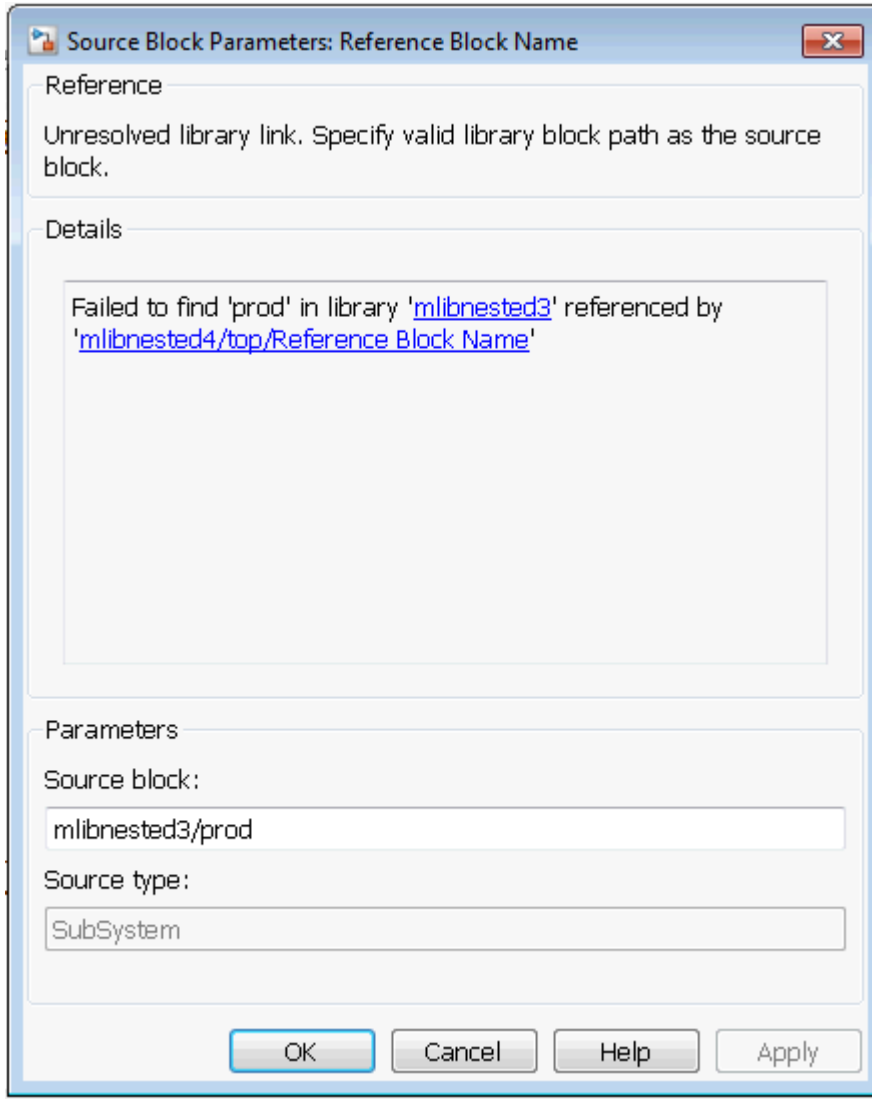
Indicate unresolved reference to library block



Description

This block indicates an unresolved reference to a library block (see “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



Details

Description of the cause of the unresolved link. You can customize this description to include URLs as follows:

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

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

Source block

Path of the library block that this link represents. To fix a bad link, 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.

Introduced in R2014a

Band-Limited White Noise

Introduce white noise into continuous system

Library

Sources

Description

Simulation of White Noise

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

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

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

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

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

Comparison with the Random Number Block

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

Usage with the Averaging Power Spectral Density Block

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

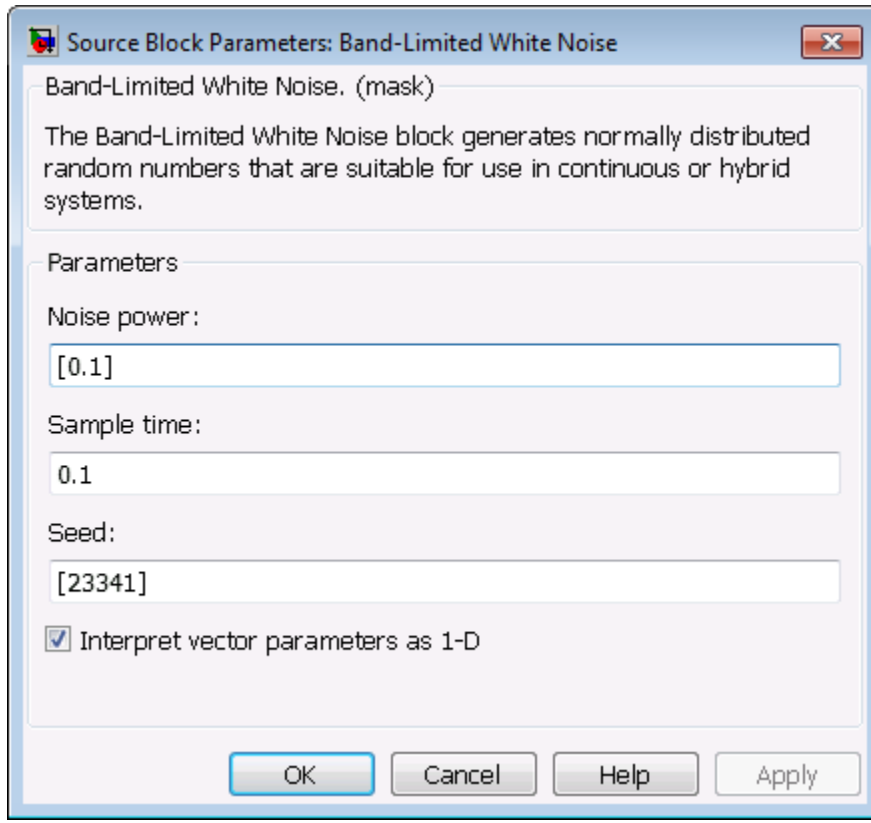
Algorithm

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`.

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

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

See Also

`Random Number`

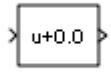
Introduced before R2006a

Bias

Add bias to input

Library

Math Operations



Description

The Bias block adds a bias, or offset, to the input signal according to
$Y = U + bias$,

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

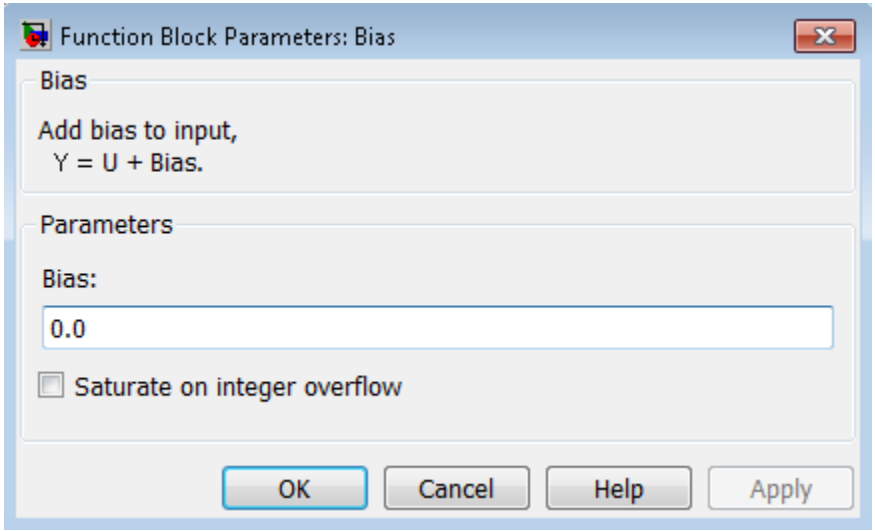
Data Type Support

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

- Floating point
- Built-in integer
- Fixed point

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

Parameters 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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.

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

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

Characteristics

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

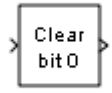
Introduced before R2006a

Bit Clear

Set specified bit of stored integer to zero

Library

Logic and Bit Operations



Description

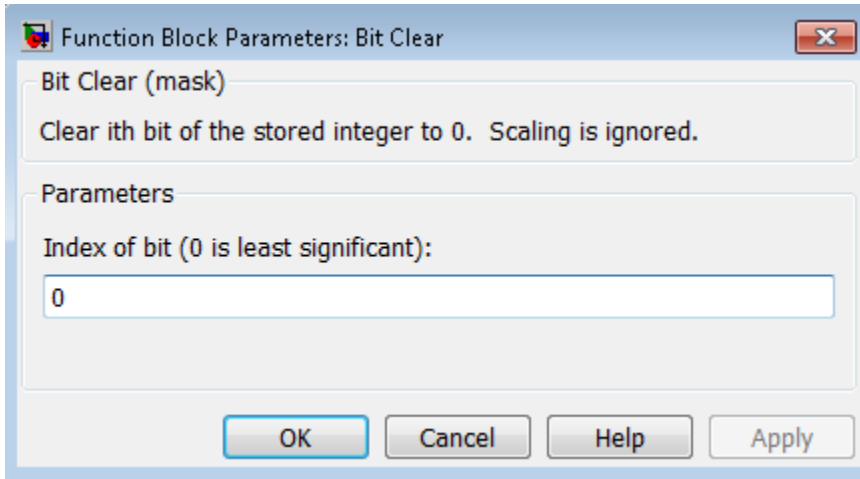
The Bit Clear block sets the specified bit, given by its index, of the stored integer to zero. Scaling is ignored.

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

Data Type Support

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

Parameters and Dialog Box



Index of bit

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

Examples

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

Characteristics

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

See Also

`Bit Set`

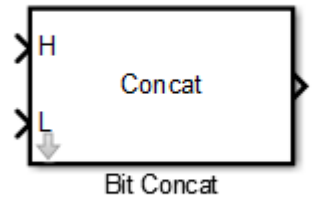
Introduced before R2006a

Bit Concat

Concatenates up to 128 input words into single output

Library

HDL Coder / HDL Operations



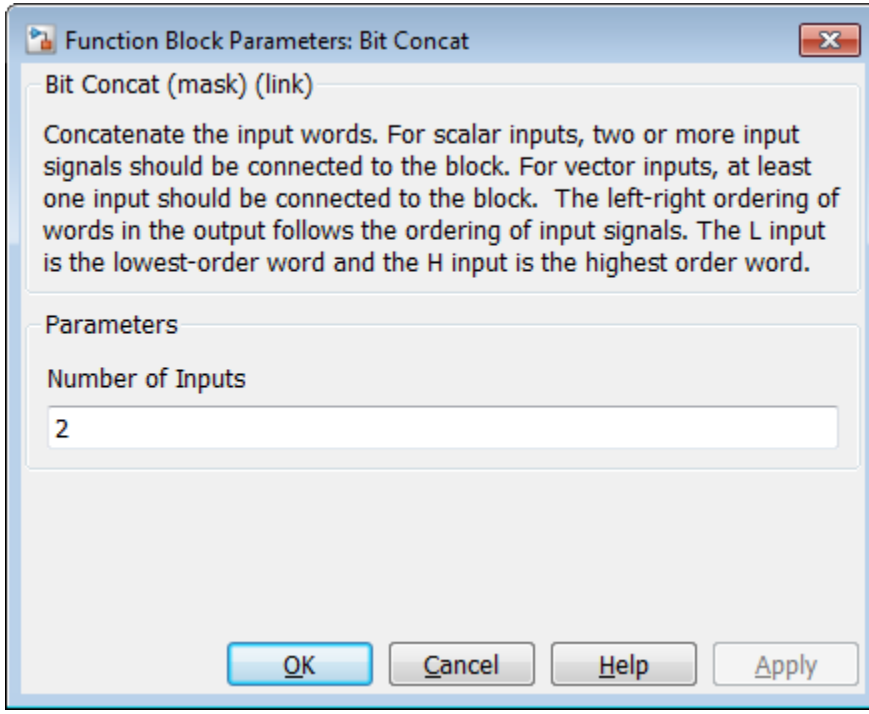
Description

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

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

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

Dialog Box and Parameters



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

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

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

Ports

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

Supported Data Types

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

See Also

`Bit Shift` | `Bit Reduce` | `Bit Rotate` | `Bit Slice`

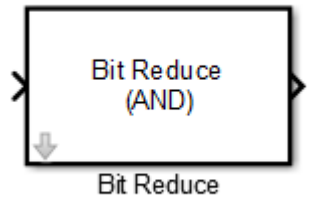
Introduced in R2014a

Bit Reduce

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

Library

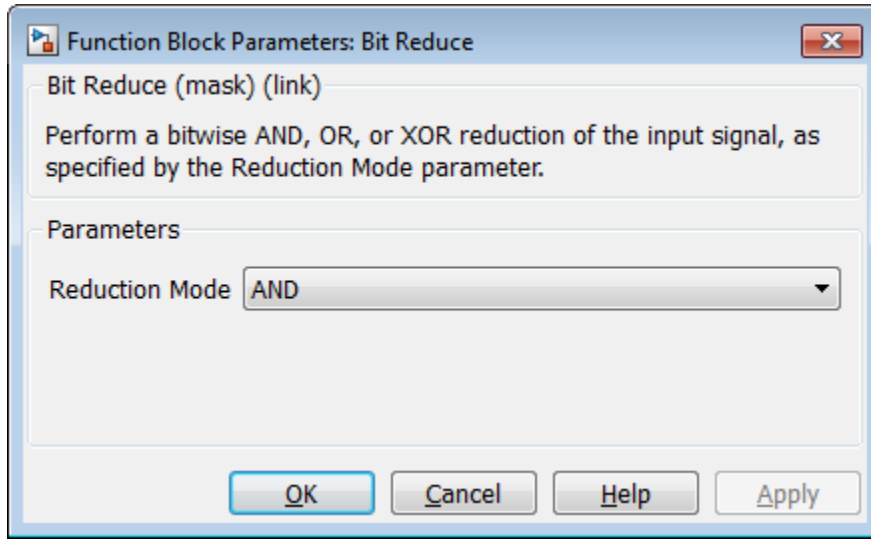
HDL Coder / HDL Operations



Description

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

Dialog Box and Parameters



Reduction Mode

Specifies the reduction operation:

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

Ports

The block has the following ports:

Input

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

Output

Supported data type: `ufix1`

See Also

`Bit Shift` | `Bit Concat` | `Bit Rotate` | `Bit Slice`

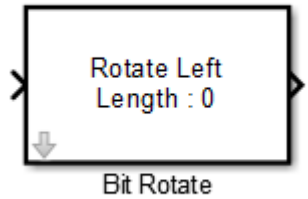
Introduced in R2014a

Bit Rotate

Rotate input signal by bit positions

Library

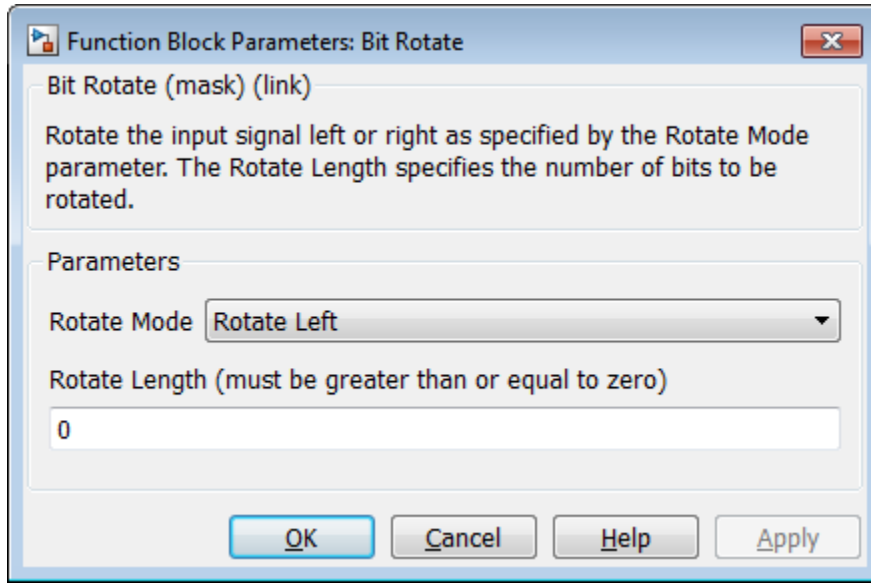
HDL Coder / HDL Operations



Description

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

Dialog Box and Parameters



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

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

Ports

The block has the following ports:

Input

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

Output

Has the same data type as the input signal.

See Also

`Bit Shift` | `Bit Concat` | `Bit Reduce` | `Bit Slice`

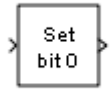
Introduced in R2014a

Bit Set

Set specified bit of stored integer to one

Library

Logic and Bit Operations



Description

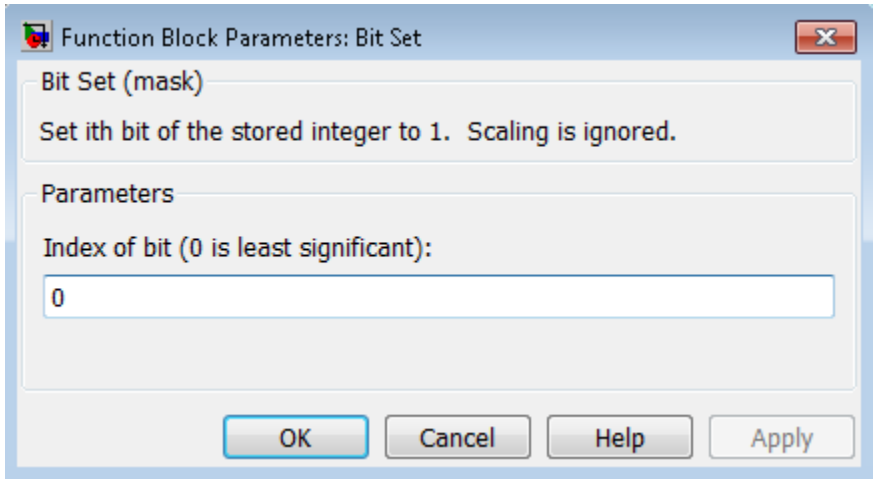
The Bit Set block sets the specified bit of the stored integer to one. Scaling is ignored.

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

Data Type Support

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

Parameters and Dialog Box



Index of bit

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

Examples

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

Characteristics

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

See Also

Bit Clear

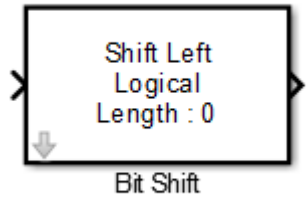
Introduced before R2006a

Bit Shift

Logical or arithmetic shift of input signal

Library

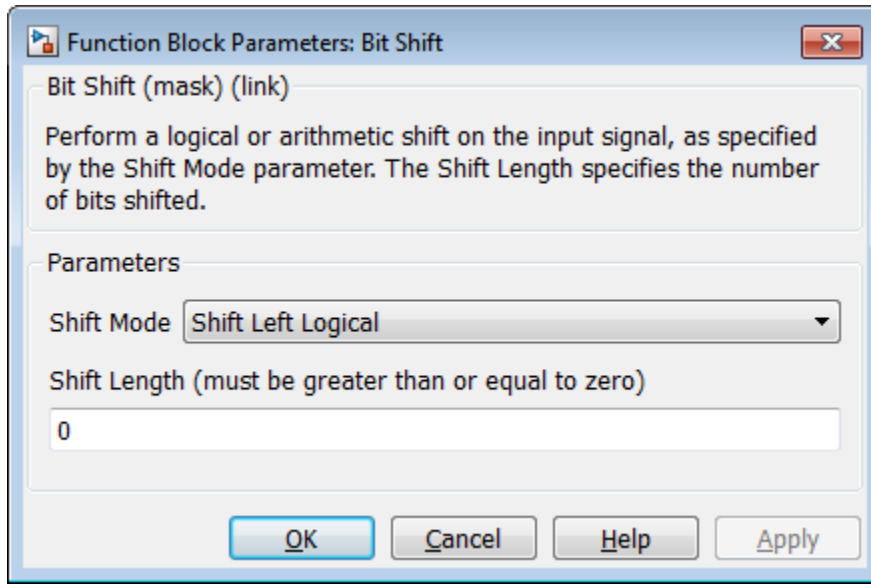
HDL Coder / HDL Operations



Description

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

Dialog Box and Parameters



Shift Mode

Default: `Shift Left Logical`

Specifies the type and direction of shift:

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

Shift Length

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

Ports

The block has the following ports:

Input

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

Output

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

See Also

`Bit Rotate` | `Bit Concat` | `Bit Reduce` | `Bit Slice`

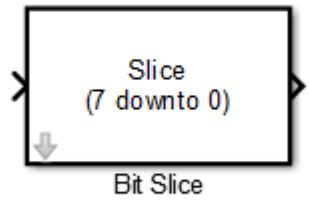
Introduced in R2014a

Bit Slice

Return field of consecutive bits from input signal

Library

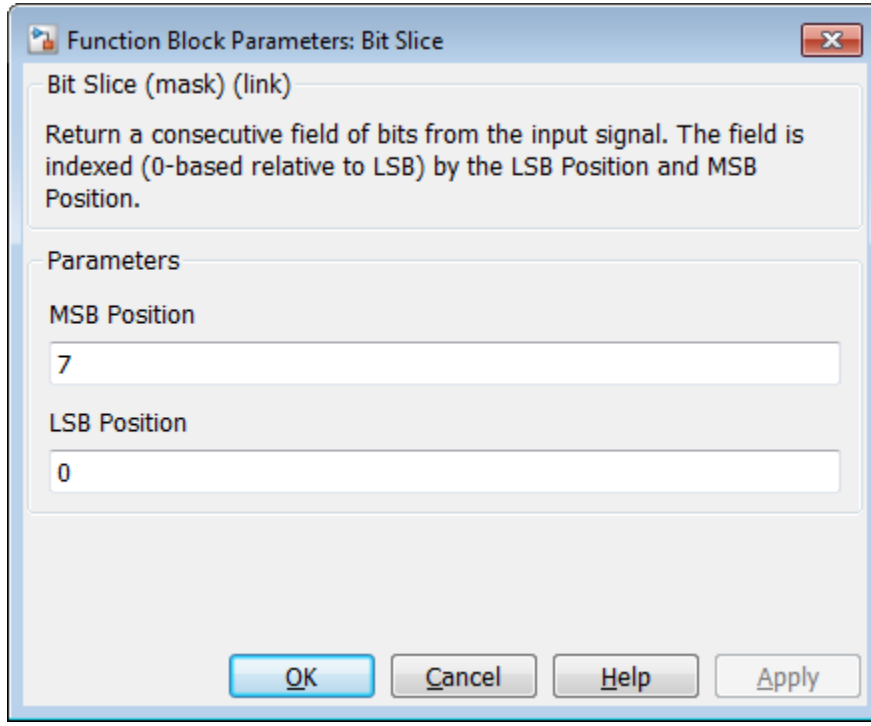
HDL Coder / HDL Operations



Description

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

Dialog Box and Parameters



MSB Position

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

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

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

The word length of the output is computed as `(MSB Position - LSB Position) + 1`.

LSB Position

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

Ports

The block has the following ports:

Input

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

Output

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

See Also

`Bit Rotate` | `Bit Concat` | `Bit Reduce` | `Bit Shift`

Introduced in R2014a

Bitwise Operator

Specified bitwise operation on inputs

Library

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.

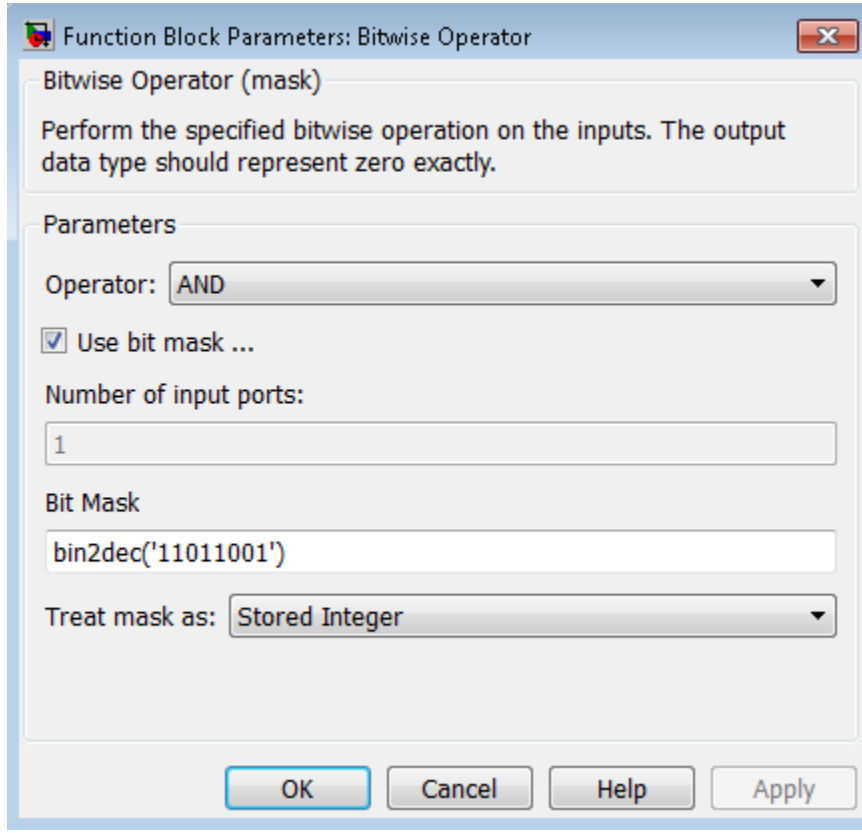
Data Type Support

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



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

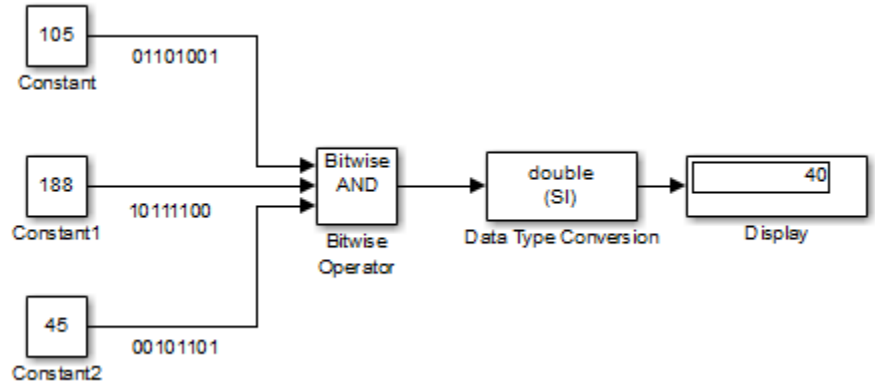
Specify whether to treat the mask as a real-world value or a stored integer. This parameter is available only when you select **Use bit mask**.

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.

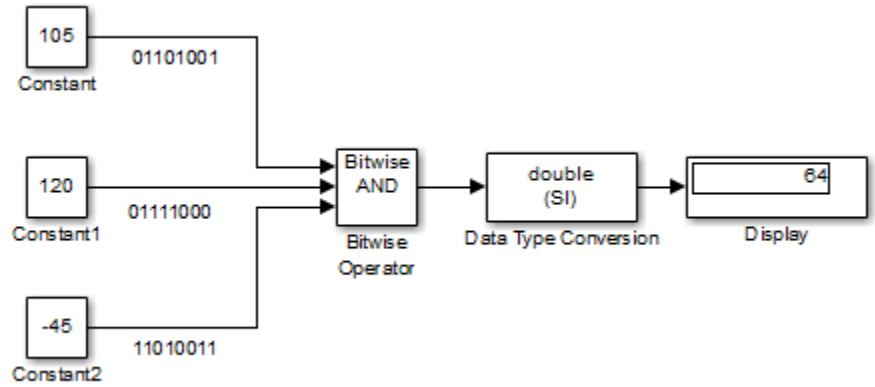


Each Constant block outputs an 8-bit unsigned integer (`uint8`). To determine the binary value of each Constant block output, use the `dec2bin` function. The results for all logic operations appear in the next table.

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

Signed Inputs for the Bitwise Operator Block

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



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

Operation	Binary Value	Decimal Value
AND	01000000	64

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

Characteristics

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

See Also

`Logical Operator`

Introduced before R2006a

Block Support Table

View data type support for Simulink blocks

Library

Model-Wide Utilities

Description

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

Data Type Support

Not applicable

Parameters and Dialog Box

Not applicable

Characteristics

Data Types	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Alternatives

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

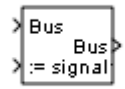
Introduced in R2007b

Bus Assignment

Replace specified bus elements

Library

Signal Routing



Description

The Bus Assignment block assigns signals (including buses and arrays of buses) 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. You cannot use the Bus Assignment block to replace a bus that is nested within an array of buses.

For information about buses, see:

- “Composite Signals”
- “Create Bus Signals”

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.

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`.

For information about using this block in a library block, see "Buses and Libraries".

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 can assign or replace a sub-bus that is an array of buses. However, the nested bus cannot be nested inside of an array of buses.
- To replace a signal in an array of buses, use a Selector block to select the index for the bus element that you want to use with the Bus Assignment block. Then use that selected bus element with the Bus Assignment block.

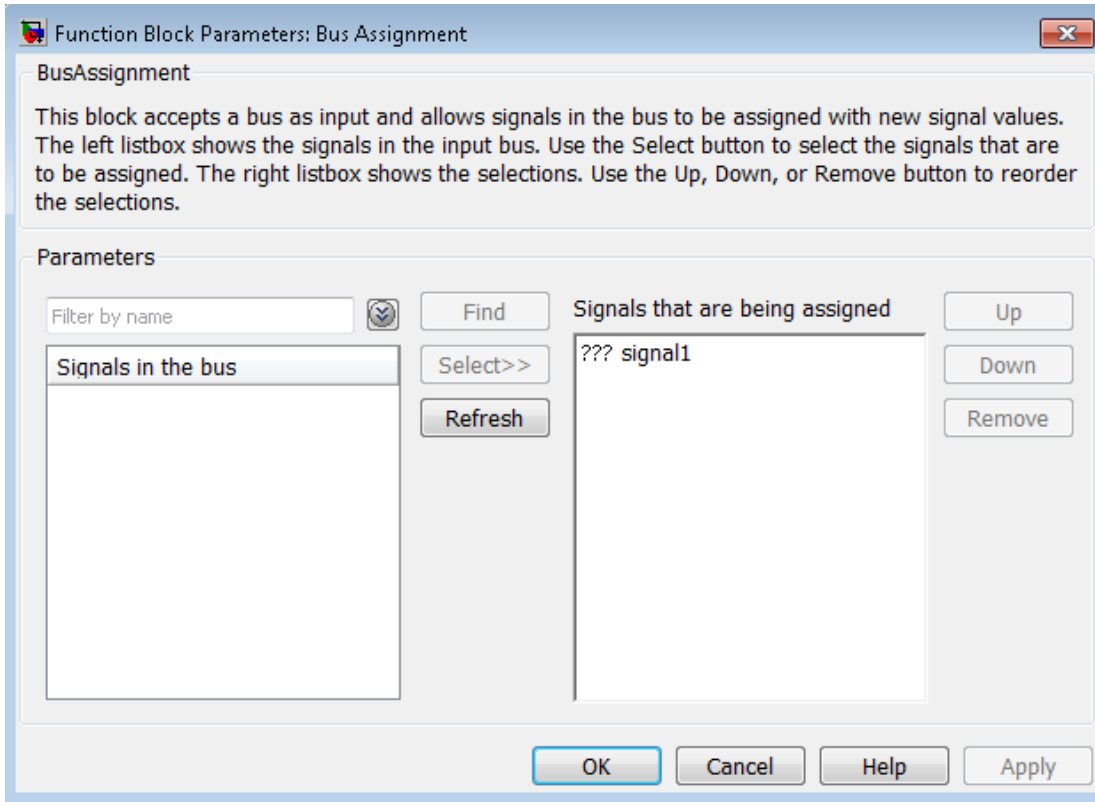
Data Type Support

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:



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 ().

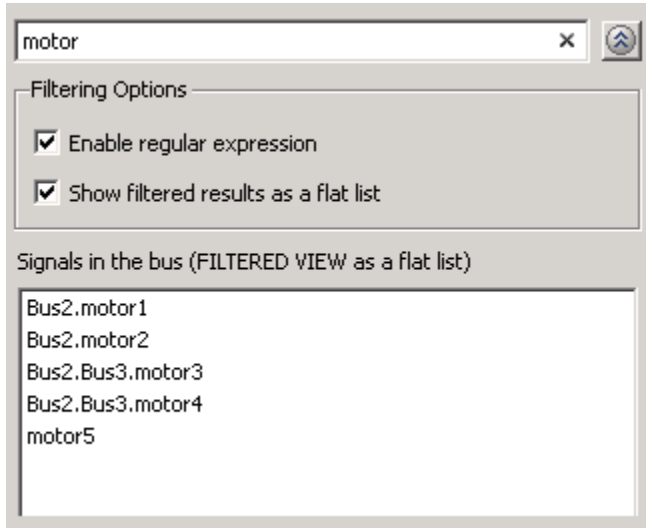
Enables the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see "Regular Expressions".

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.



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

Characteristics

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

See Also

- "Composite Signals"
- "Create Bus Signals"
- `Bus Creator`
- `Bus Selector`
- `Bus to Vector`

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

Introduced before R2006a

Bus Creator

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.

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 bus 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.

For information about using this block in a library block, see “Buses and Libraries”.

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”.

Bus Signal Naming

The Bus Creator block assigns a name to each signal on the bus that it creates. You can then refer to signals by name when you are searching for their sources (see “Browse Bus Signals” on page 1-120) or selecting signals for connection to other blocks.

Specify one of the following signal naming options:

- 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.

  The Bus Creator block generates names for bus signals whose corresponding inputs do not have names. The names are in the form `signaln`, where `n` is the number of the port the input signal connects to.
- Each input signal must have a specific name.
- If the bus output data type is a bus object, bus signal names use the corresponding bus object element names.

You can change the name of any signal by editing the 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, to see the updated name in the dialog box, click the **Refresh** button next to the **Signals in the bus** list.

Bus Object as the Output Data Type

You can use a bus object as the bus output data type for a Bus Creator block. Using a bus object can provide strong data typing with an explicit signal interface. Model referencing requires using bus objects for bus signals that cross model reference boundaries. For more information, see “Bus Objects”.

To create a nonvirtual bus using a Bus Creator block, use the following settings.

- For the **Output data type** parameter, use a bus object.
- Select **Output as nonvirtual bus**.

To use a bus object to enforce strong data typing, clear the **Override bus signal names from inputs** check box.

Browse 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. An arrow next to a signal indicates that the signal is itself a bus. To display the contents of the bus, click the arrow. 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.

Reorder, Add, or Remove Signals

To rearrange the signals that the Bus Creator block includes in the bus signal that it produces, use buttons such as **Add**.

You can select multiple contiguous signals in the **Signals in the bus** list to reorder or remove. You cannot rearrange leaf signals within a bus. For example, you can move bus signal `Bus1` up or down in the list, but you cannot reorder any of the bus elements of `Bus1`.

After making your edits, click **Apply**.

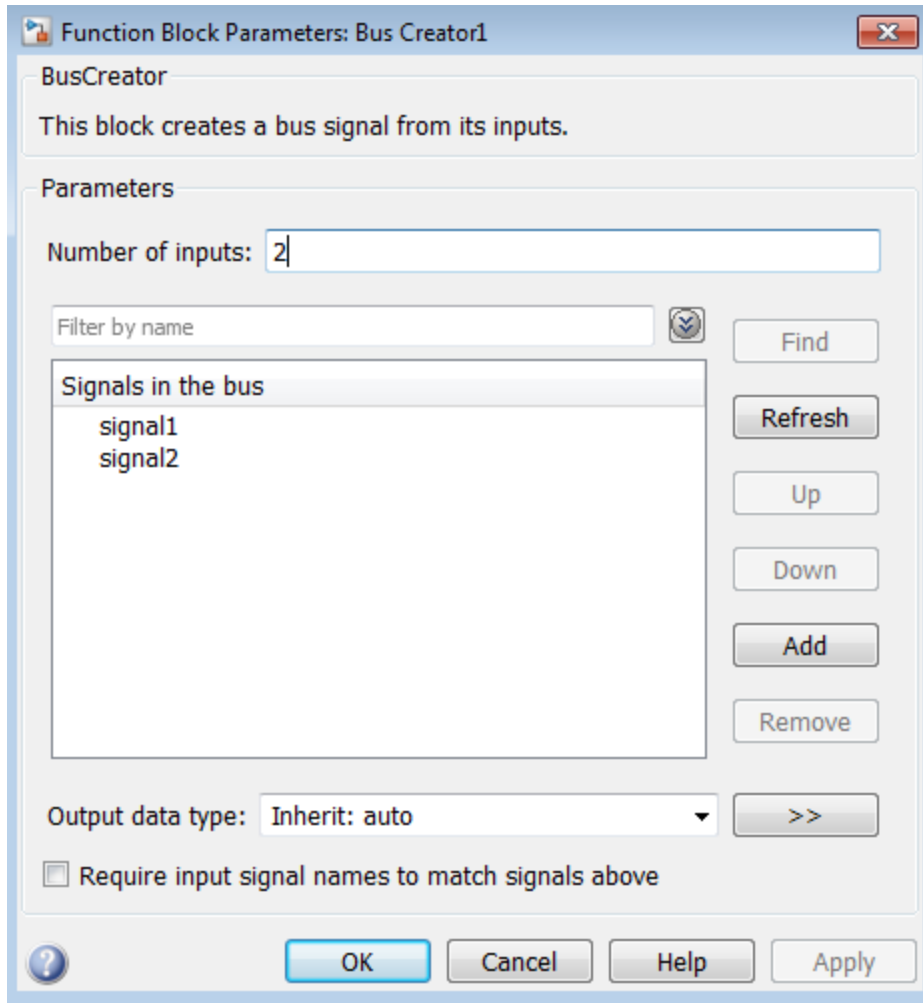
Data Type Support

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”.

If you change elements or the order of elements in the Bus Creator and the incoming bus is a nonvirtual bus, Simulink reports any inconsistency errors when you compile the model.

Parameters and Dialog Box



- "Number of inputs" on page 1-123
- "Signals in the bus" on page 1-124
- "Enable regular expression" on page 1-125
- "Show filtered results as a flat list" on page 1-126

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 6-101 for the command-line information.

Signals in the bus

Show the input signals for the bus.

Settings

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

Tips

- An arrow next to a signal name indicates that the signal is itself a bus. Click the arrow 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.
- To rearrange signals in the bus, see “Reorder, Add, or Remove Signals” on page 1-120.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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

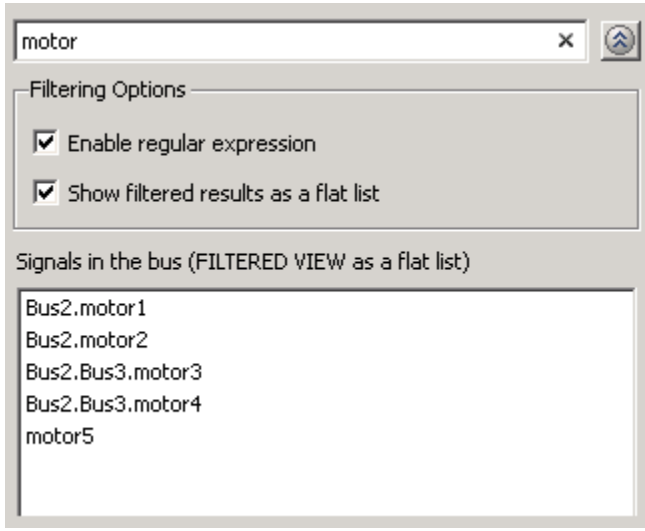
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

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.



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.

Output 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.

Tips

- Determine whether you want the Bus Creator block to output a virtual or nonvirtual bus.
  - For a virtual bus, use the **Output data type** parameter default (`Inherit: auto`) or set the parameter to specify a bus object using `Bus: <object name>`.
  - For a nonvirtual bus, set the **Output data type** parameter to specify a bus object using `Bus: <object name>` and click **Output as nonvirtual bus**.
- If you specify a bus object as the output data type, to have bus signal names match the corresponding bus object element names, clear the **Override bus signal names from inputs** check box (which is selected by default).

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 “Control Signal Data Types”.

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.

Tip

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 6-101 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 6-101 for the command-line information.

Override bus signal names from inputs

Override bus signal names from input signals or inherit names from the bus object elements.

Settings

Default: On

 On

Override bus element names from input signal names.

 Off

Inherit bus signal names from the corresponding element names in the bus object.

Tips

- To inherit signal names from bus element names, clear the **Override bus signal names from inputs** check box. This approach:
  - Enforces strong data typing.
  - Avoids having to enter a signal name multiple times. Without this option, you need to enter the signal names in the bus object and in the model, which can lead to accidentally creating signal name mismatches.
  - Supports the array of buses requirement to have consistent signal names across array elements.
- Alternatively, you can enforce strong data typing and also check that input signal names match the bus object element names.
  - Select the **Override bus signal names from inputs** check box.
  - Set the **Configuration Parameters** > **Diagnostics** > **Connectivity** > **Element name mismatch** parameter to `error`.

Dependencies

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

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Require input signal names to match signals above

Require that input signals have the names listed in the **Signals in the bus** list.

Settings

Default: Off

 On

Check that the input signal names match the signal names in the Bus Creator block parameters dialog boxs.

 Off

Does not check that the input signal names match the signal names in the Bus Creator block parameters dialog box.

Tips

- The **Require input signal names to match signals above** option might be removed in a future release. To enforce strong data typing, see the Bus Creator block parameter "Override bus signal names from inputs" on page 1-131.
- If you select **Override bus signal names from inputs**, the **Require input signal names to match signals above** setting is ignored.

Rename selected signal

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

Settings

Default: `''`

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

Dependencies

Selecting **Require input signal names to match signals above** enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Examples

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

Characteristics

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

See Also

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

Introduced before R2006a

Bus Selector

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 Bus Signals"

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.

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`.

For information about using this block in a library block, see "Buses and Libraries".

Reorder or Remove Signals

To reorder the selected signals that the Bus Selector block includes in the bus signal that it produces, click **Up** or **Down**.

You can select multiple contiguous signals in the **Signals in the bus** list to remove or reorder.

You cannot rearrange leaf signals within a bus. For example, you can move bus signal `Bus1` up or down in the list, but you cannot reorder any of the bus elements of `Bus1`.

After you click a button, click **Apply**.

Array of Buses Support

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.

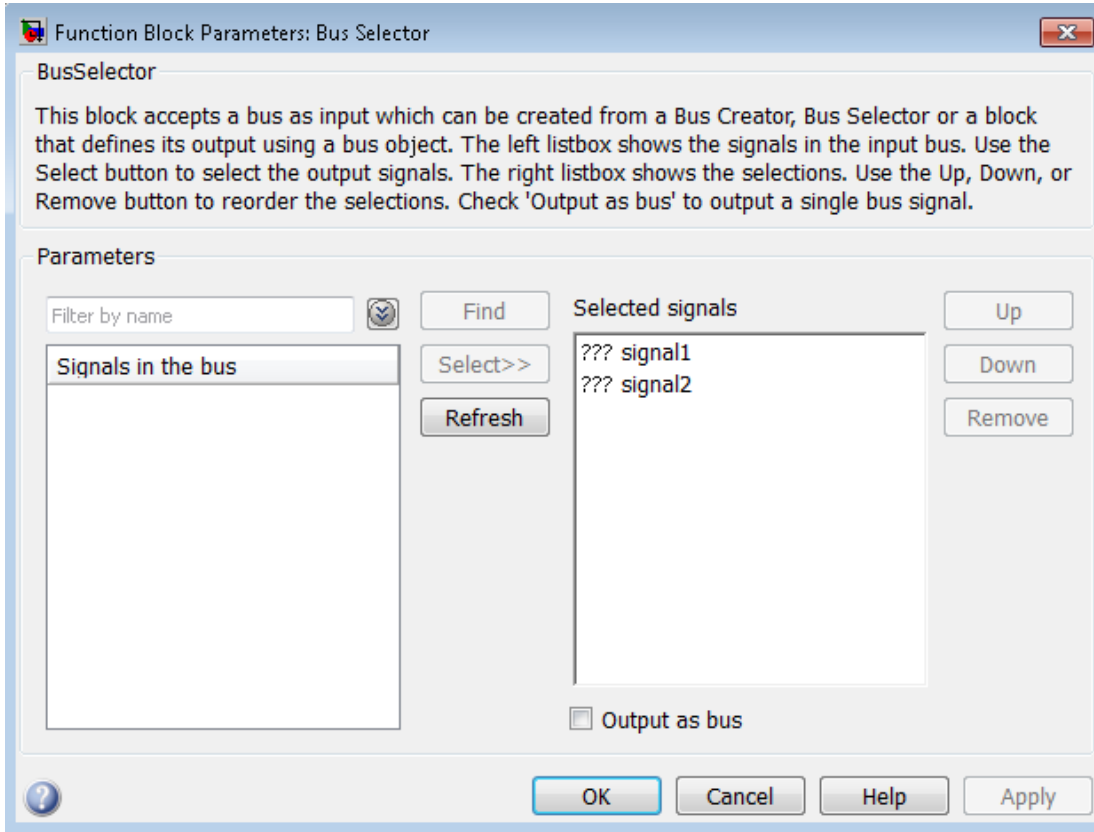
Data Type Support

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
Signals in the bus
Find
Select>>
Refresh
Selected signals
??? signal1
??? signal2
Up
Down
Remove
Output as bus
OK
Cancel
Help
Apply

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 6-101 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

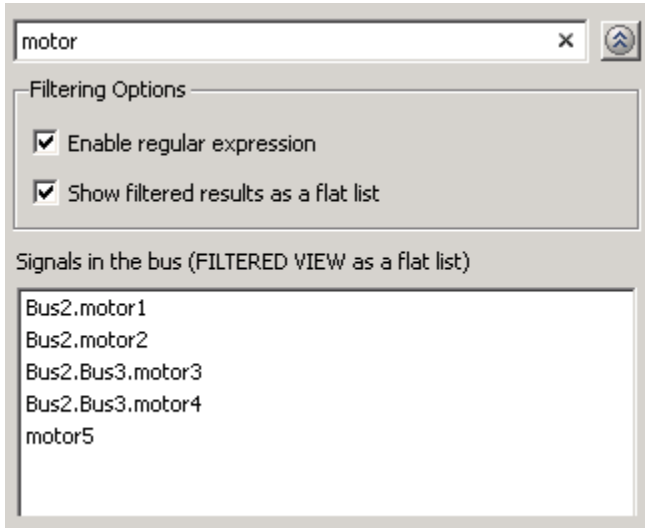
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

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.



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 6-101 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.
- If the **Selected signals** list box includes only one signal and you enable **Output as bus**, then:
  - If the selected signal is a non-bus signal, it is treated as a non-bus signal (it is not wrapped in a bus).
  - If the selected signal is a bus signal, then the output is that bus signal.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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`

Characteristics

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

See Also

- "Composite Signals"
- "Create Bus Signals"
- `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`

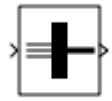
Introduced before R2006a

Bus to Vector

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 “Correct Buses Used as Muxes”.

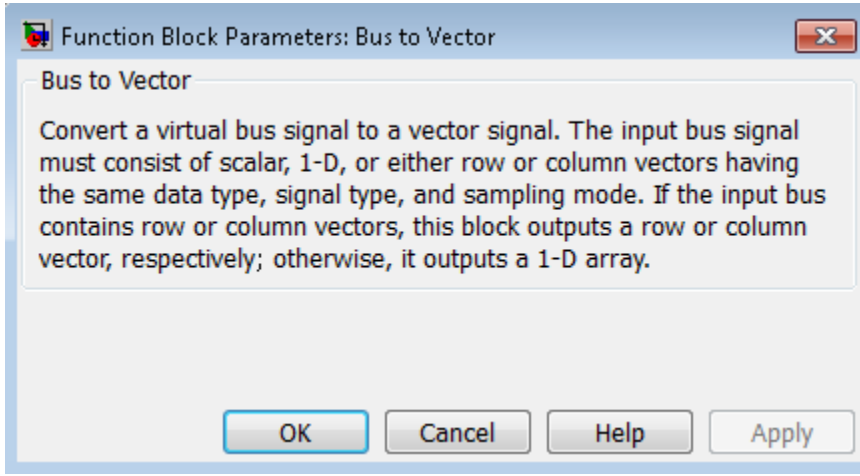
Note Simulink hides the name of a Bus to Vector block when you copy it from the Simulink library to a model.

Data Type Support

The Bus to Vector block accepts virtual bus signals and nonbus signals. If the input is a nonbus signal, this block has no effect.

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

Parameters and Dialog Box



This block has no user-accessible parameters.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

- “Composite Signals”
- “Create Bus Signals”
- 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`

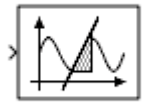
Introduced in R2007a

Check Discrete Gradient

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

Library

Model Verification



Description

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

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

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

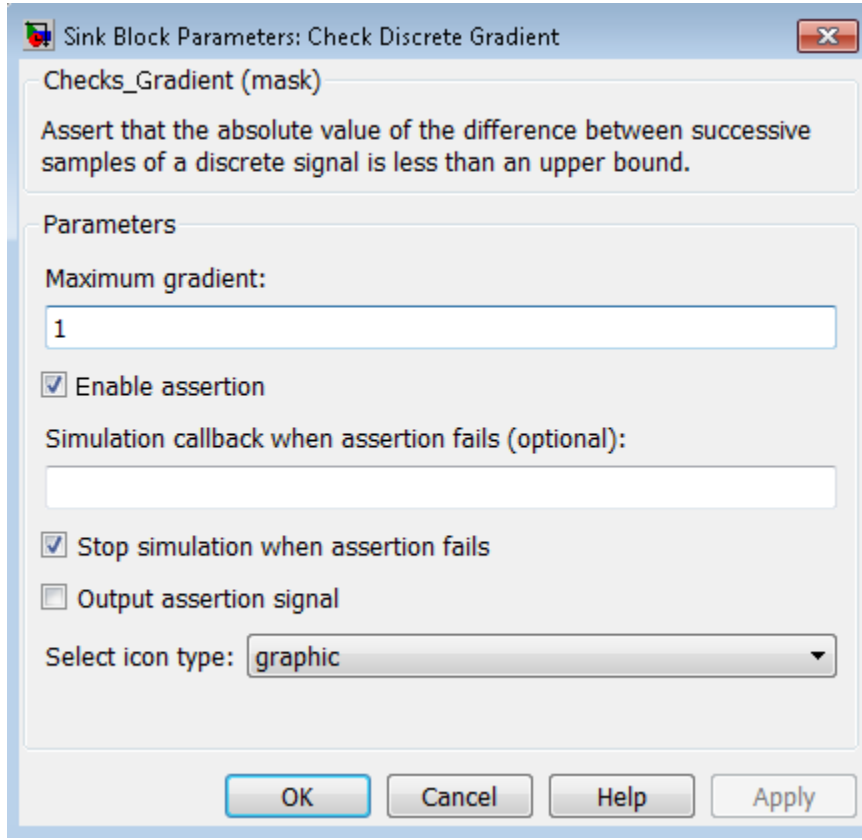
Note: For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug”.

Data Type Support

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

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

Parameters 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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Discrete Gradient blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Discrete Gradient block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Inherited from driving block

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

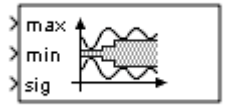
Introduced before R2006a

Check Dynamic Gap

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

Library

Model Verification



Description

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

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

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

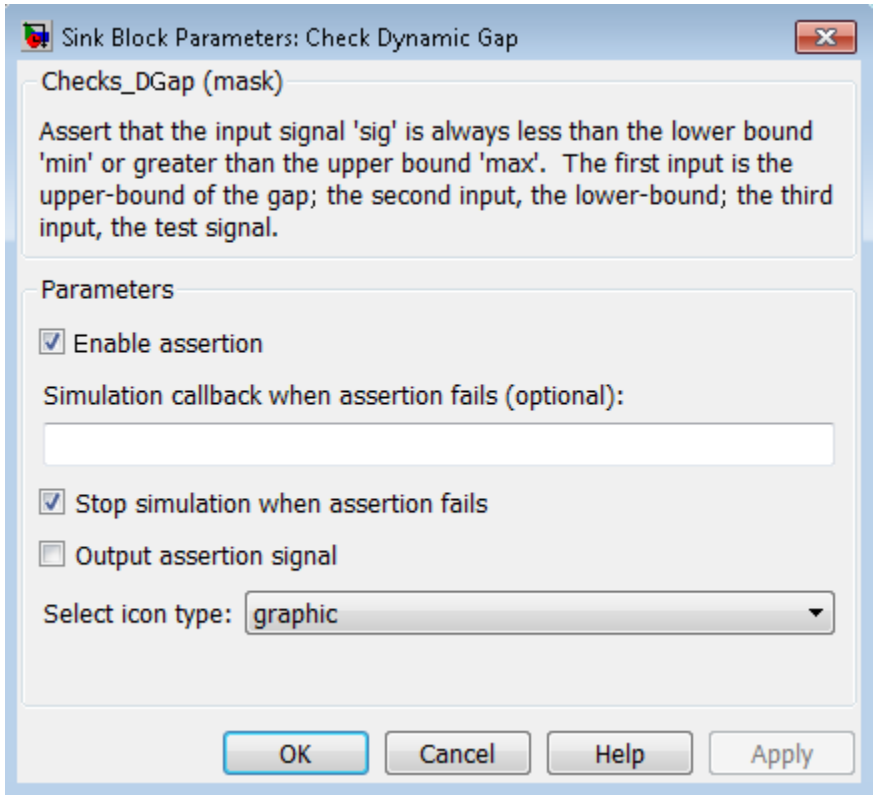
Data Type Support

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



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 on the **All Parameters** tab in the Configuration Parameters dialog

box allows you to enable or disable all model verification blocks in a model, including Check Dynamic Gap blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Gap block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parmeters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

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

Code Generation	Yes

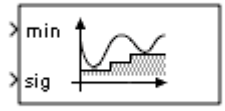
Introduced before R2006a

Check Dynamic Lower Bound

Check that one signal is always less than another signal

Library

Model Verification



Description

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

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

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

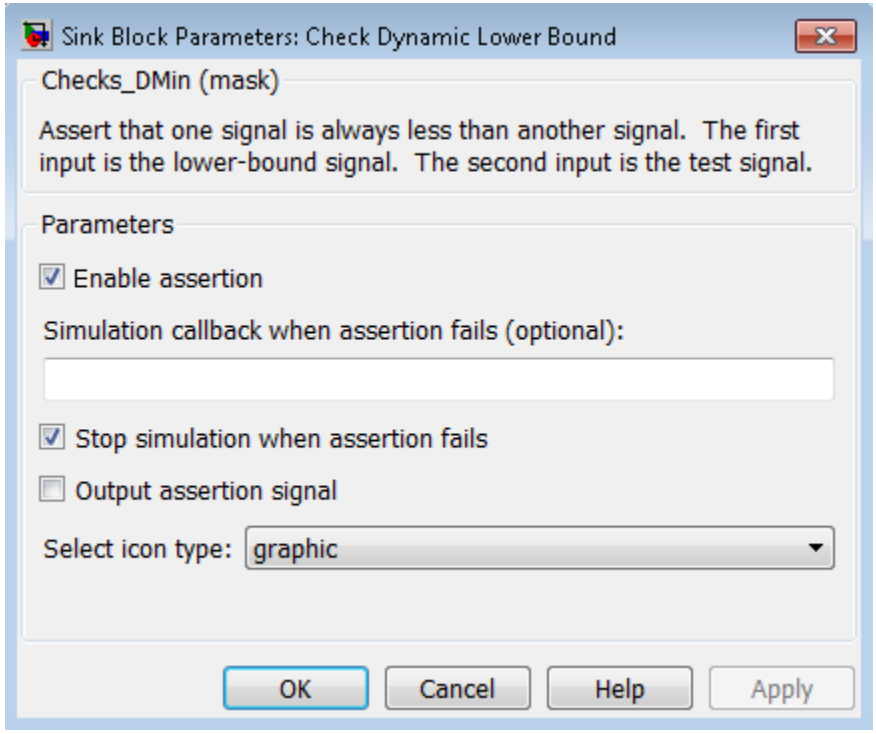
Data Type Support

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

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

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

Parameters 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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks, including Check Dynamic Lower Bound blocks, in a model regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Lower Bound block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

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

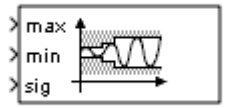
Introduced before R2006a

Check Dynamic Range

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

Library

Model Verification



Description

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

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

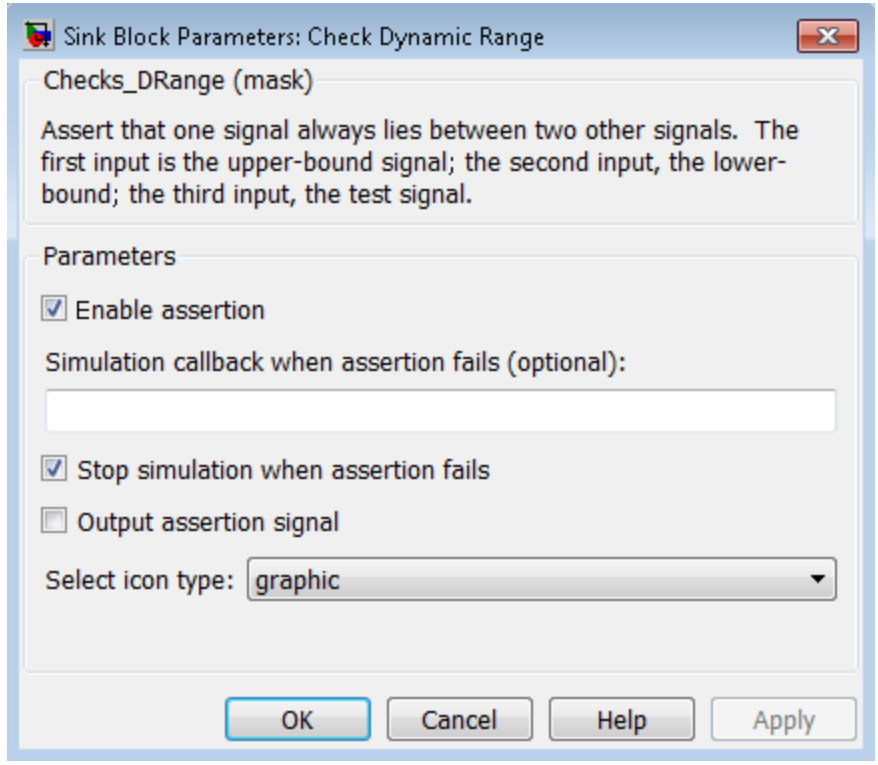
Note: For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug”.

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



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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Dynamic Range blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Range block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes

Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

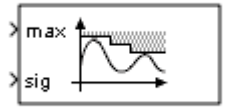
Introduced before R2006a

Check Dynamic Upper Bound

Check that one signal is always greater than another signal

Library

Model Verification



Description

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

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

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

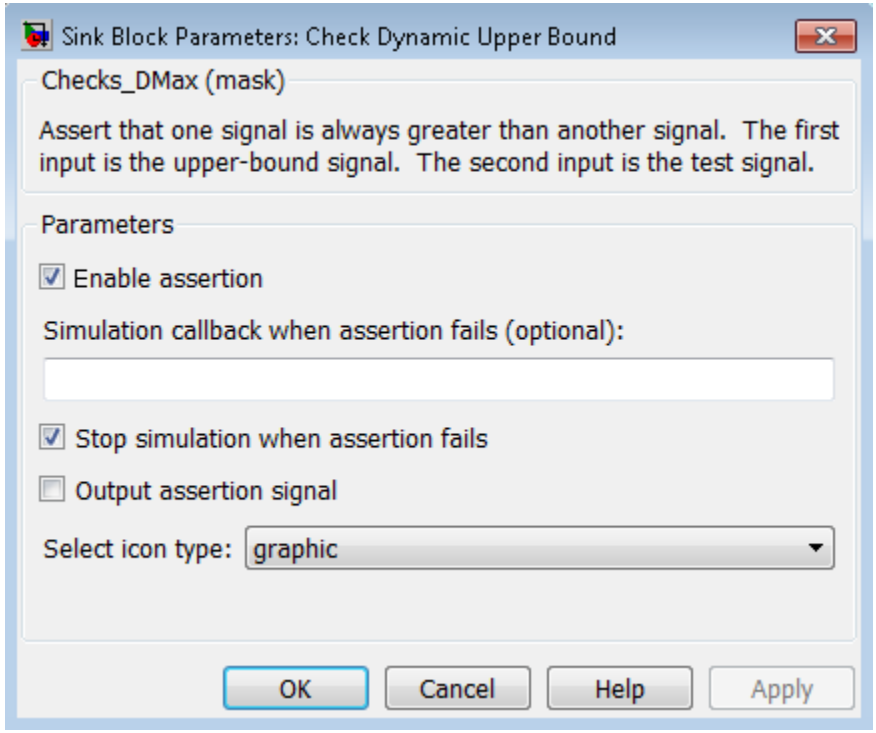
Data Type Support

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

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

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

Parameters 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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks, including

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

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Upper Bound block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

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

Code Generation	Yes

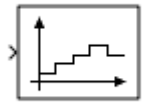
Introduced before R2006a

Check Input Resolution

Check that input signal has specified resolution

Library

Model Verification



Description

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

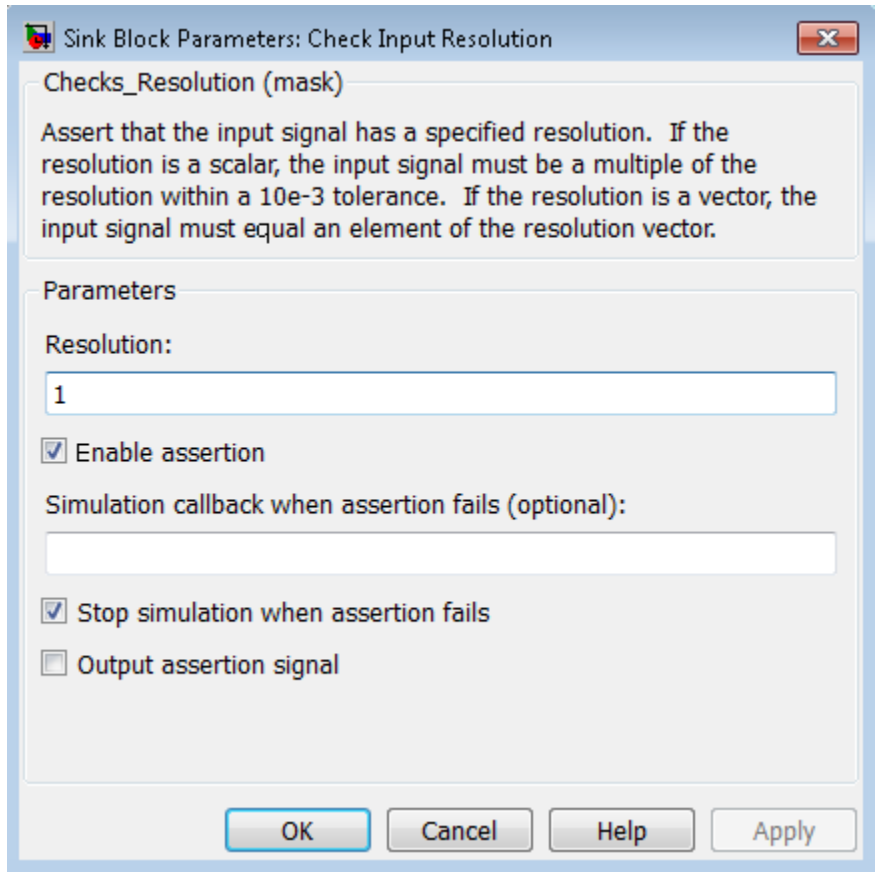
Note: For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug”.

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.

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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Input Resolution blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Input Resolution block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Characteristics

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

Code Generation	Yes

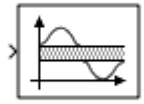
Introduced before R2006a

Check Static Gap

Check that gap exists in signal's range of amplitudes

Library

Model Verification



Description

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

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

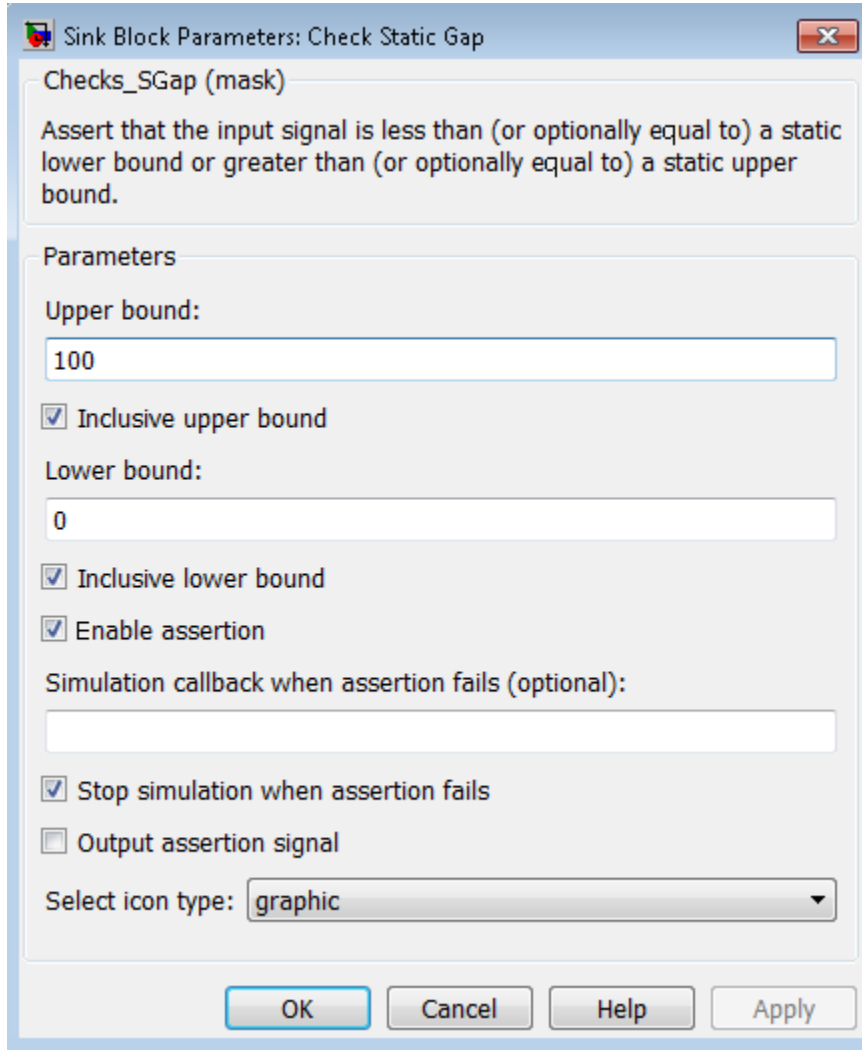
Note: For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug”.

Data Type Support

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

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

Parameters 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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Gap blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Static Gap block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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

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

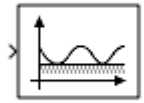
Introduced before R2006a

Check Static Lower Bound

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

Library

Model Verification



Description

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

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

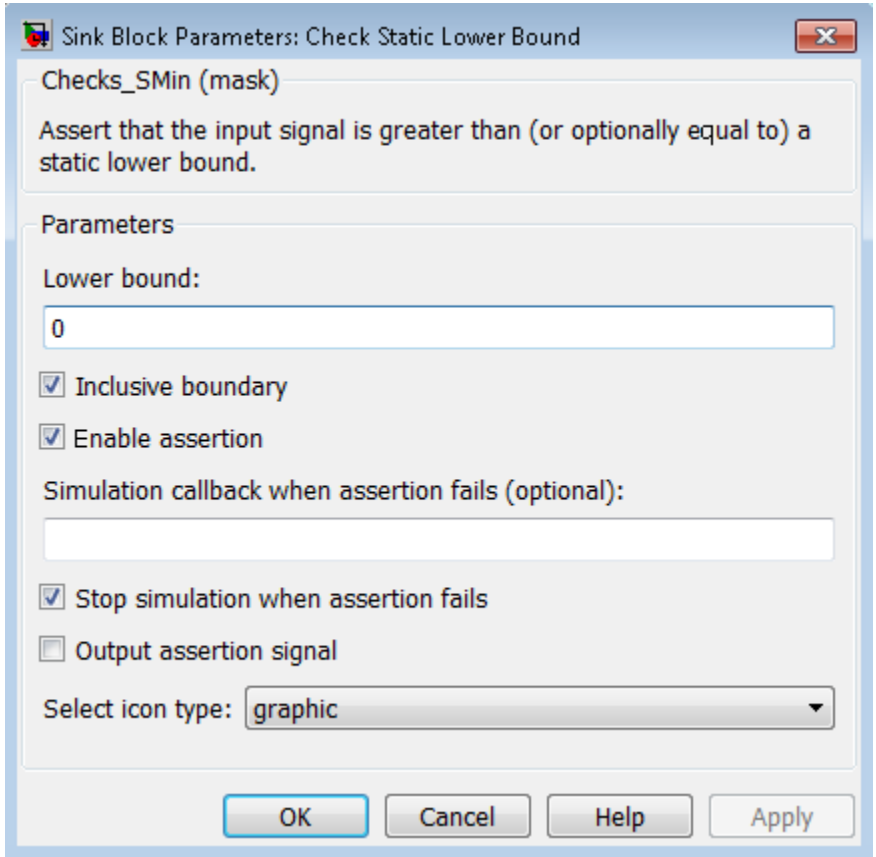
Note: For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug”.

Data Type Support

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

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

Parameters 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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Lower Bound blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Static Lower Bound block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No

Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

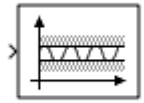
Introduced before R2006a

Check Static Range

Check that signal falls inside fixed range of amplitudes

Library

Model Verification



Description

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

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

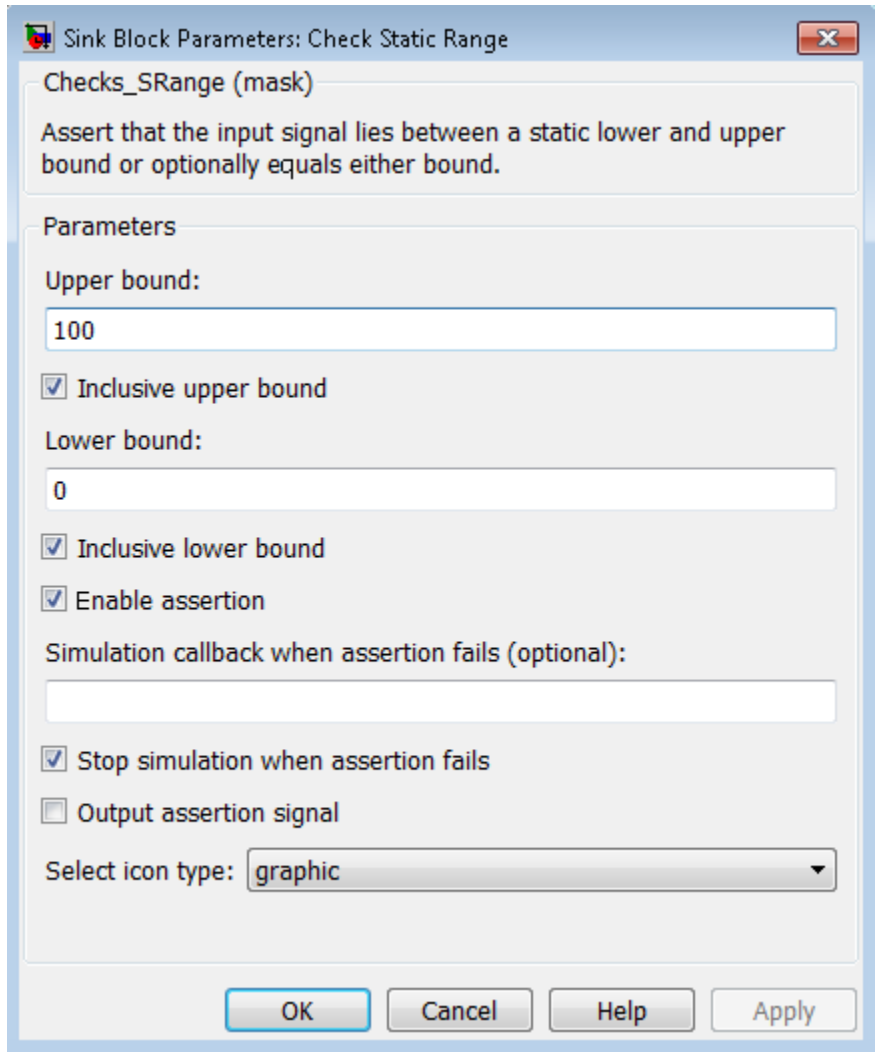
Note: For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug”.

Data Type Support

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

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

Parameters 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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Range blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Static Range block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

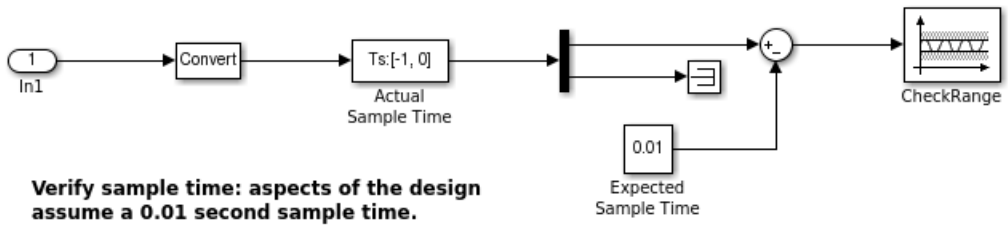
Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion

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

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

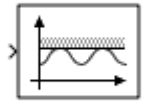
Introduced before R2006a

Check Static Upper Bound

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

Library

Model Verification



Description

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

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

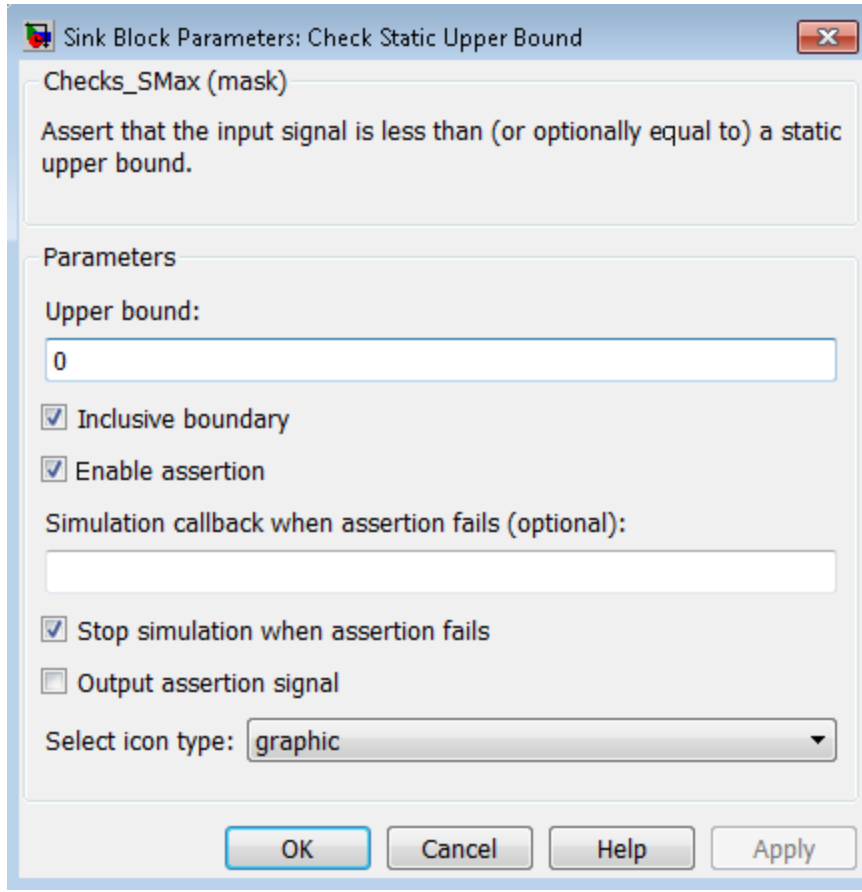
Note: For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug”.

Data Type Support

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

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

Parameters 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 on the **All Parameters** tab in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Upper Bound blocks, regardless of the setting of this option.

Simulation callback when assertion fails

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

Stop simulation when assertion fails

Selecting this check box causes the Check Static Upper Bound block to halt the simulation when the block's output is zero and display an error in the Diagnostic 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 **All Parameters** tab of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

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

Characteristics

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

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

Introduced before R2006a

Chirp Signal

Generate sine wave with increasing frequency

Library

Sources

Description

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

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

The following limitations apply to the Chirp Signal block:

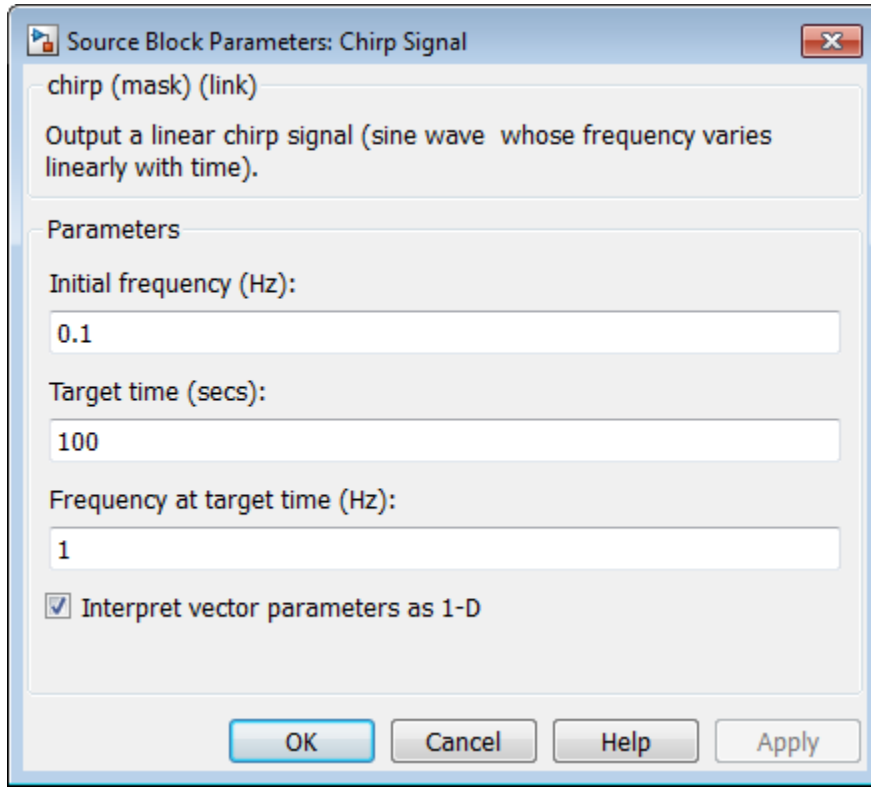
- The start time of the simulation must be 0. To confirm this value, go to the **Solver** pane in the Configuration Parameters dialog box and view the **Start time** field.
- Suppose that you use a Chirp Signal block in an enabled subsystem. Whenever the subsystem is enabled, the block output matches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

Data Type Support

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

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

Parameters 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

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

Introduced before R2006a

Clock

Display and provide simulation time

Library

Sources

Description

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

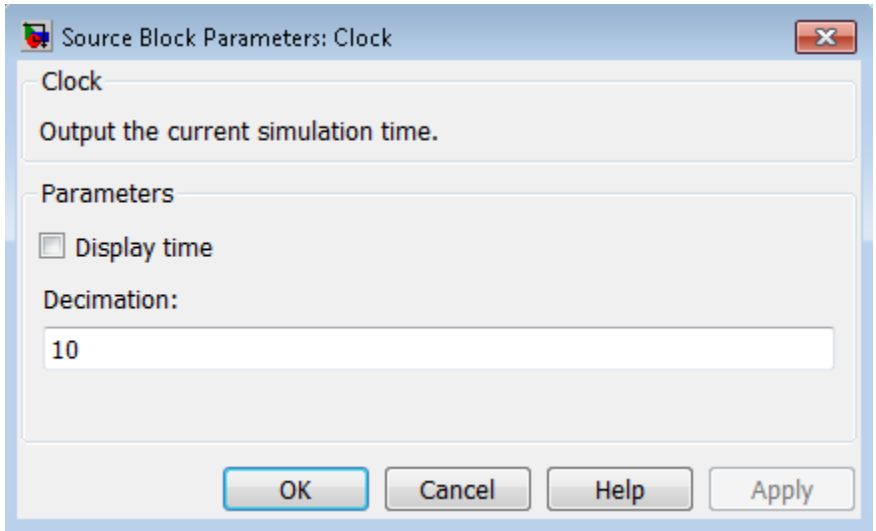
When you need the current time within a discrete system, use the `Digital Clock` block.

Data Type Support

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

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

Parameters and Dialog Box



Display time

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

Decimation

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

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

Examples

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

- `sldemo_tonegen_fixpt`
- `penddemo`

Characteristics

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

See Also

`Digital Clock`

Introduced before R2006a

Combinatorial Logic

Implement truth table

Library

Logic and Bit Operations

Description

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

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

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

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

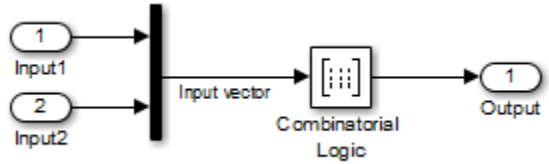
Simulink returns a row of the matrix by computing the row's index from the input vector elements. Simulink computes the index by building a binary number where input vector elements having zero values are 0 and elements having nonzero values are 1, then adding 1 to the result. For an input vector, `u`,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

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

Two-Input AND Logic

This example builds a two-input AND function, which returns 1 when both input elements are 1, and 0 otherwise. To implement this function, specify the **Truth table**

parameter value as `[0; 0; 0; 1]`. The portion of the model that provides the inputs to and the output from the Combinatorial Logic block might look like this:



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

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

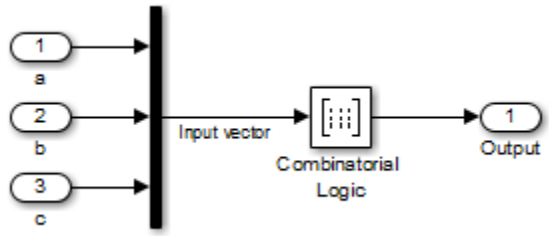
```
(2^1*1 + 1)
```

The output value is 0.

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

Circuit Logic

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



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

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

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

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

Data Type Support

The type of signals accepted by a `Combinatorial Logic` block depends on whether you selected the Boolean logic signals option (see “Implement logic signals as Boolean data

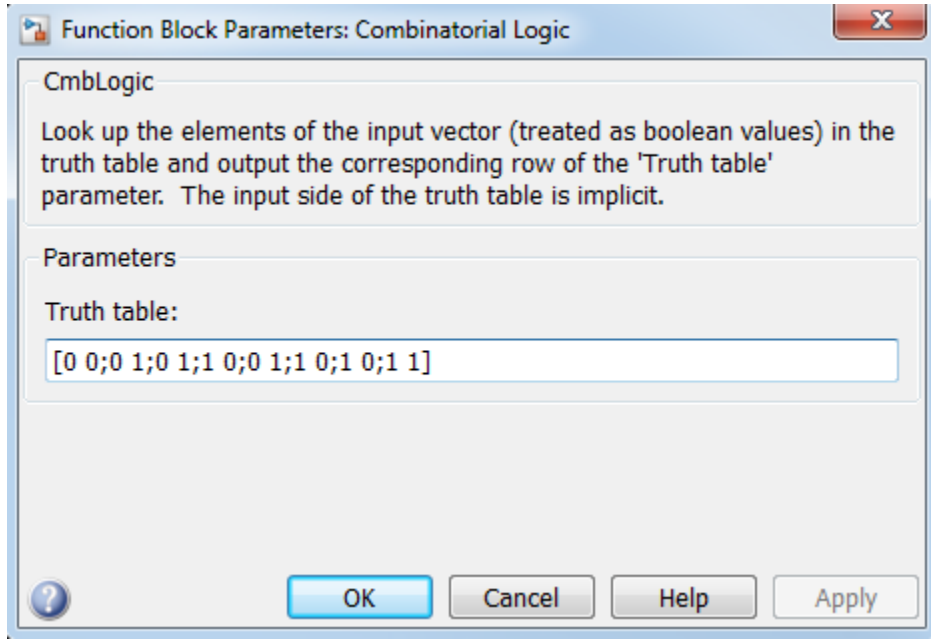
(vs. double)"). If this option is enabled, the block accepts real signals of type `Boolean` or `double`. The **Truth table** parameter can have Boolean values (0 or 1) of any data type, including fixed-point data types. If the table contains non-Boolean values, the data type of the table must be `double`.

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

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

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

Parameters 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

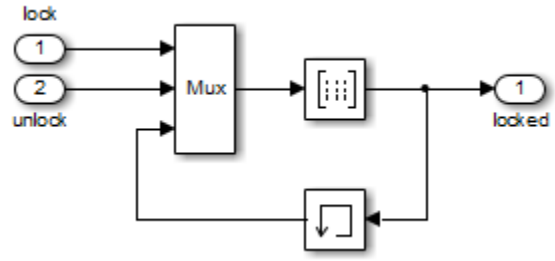
Note: This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see "Blocks for Which Sample Time Is Not Recommended".

Examples

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

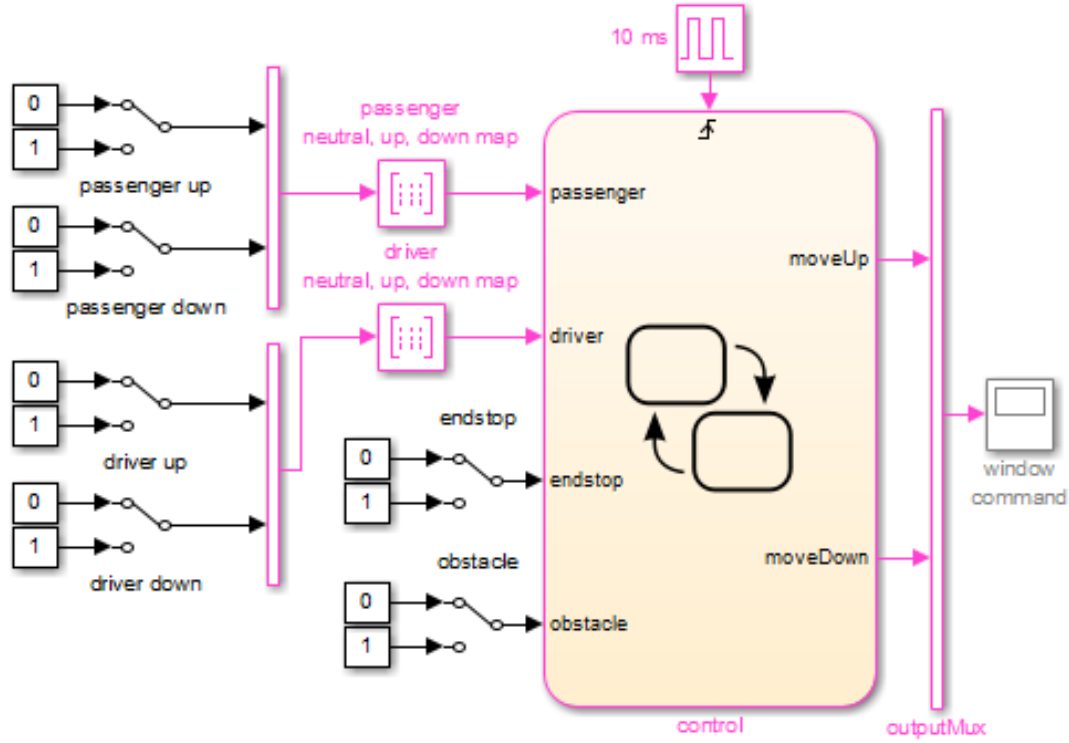
The `sldemo_clutch` model shows how you can use the `Combinatorial Logic` block with the `Memory` block to implement a finite-state machine.

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



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

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



Characteristics

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

Introduced before R2006a

Compare To Constant

Determine how signal compares to specified constant

Library

Logic and Bit Operations

Description

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

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

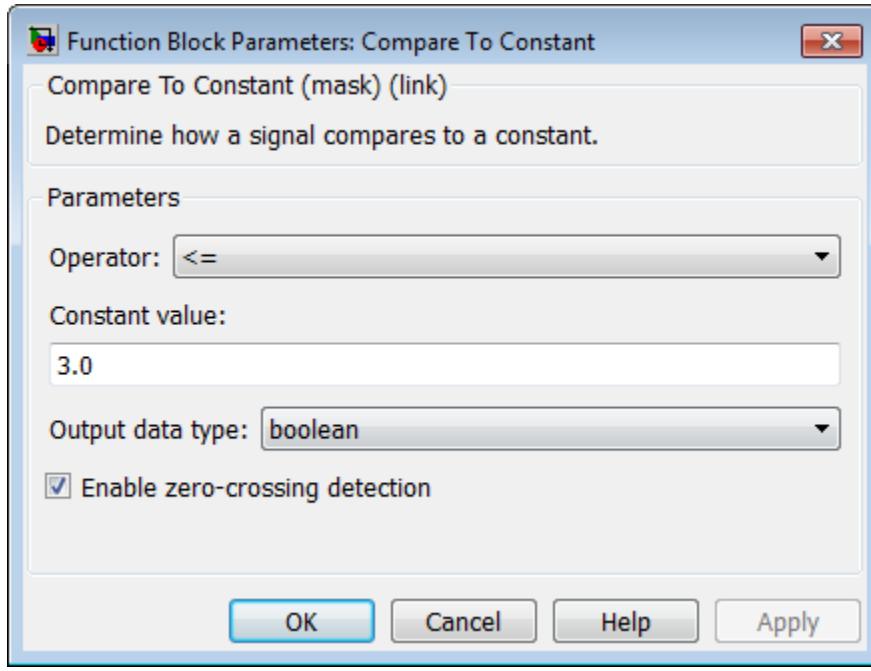
The output is `0` if the comparison is false, and `1` if it is true.

Data Type Support

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

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

Parameters 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

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

See Also

`Compare To Zero`

Introduced before R2006a

Compare To Zero

Determine how signal compares to zero

Library

Logic and Bit Operations

Description

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

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

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

Data Type Support

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

- Floating point
- Built-in integer
- Fixed point

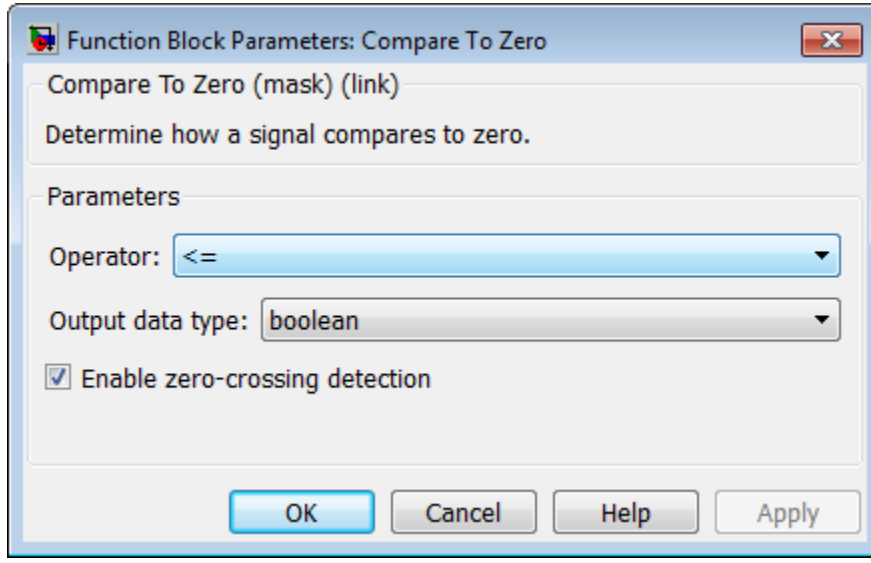
- Boolean

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

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

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

Parameters 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

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

See Also

`Compare To Constant`

Introduced before R2006a

Complex to Magnitude-Angle

Compute magnitude and/or phase angle of complex signal

Library

Math Operations

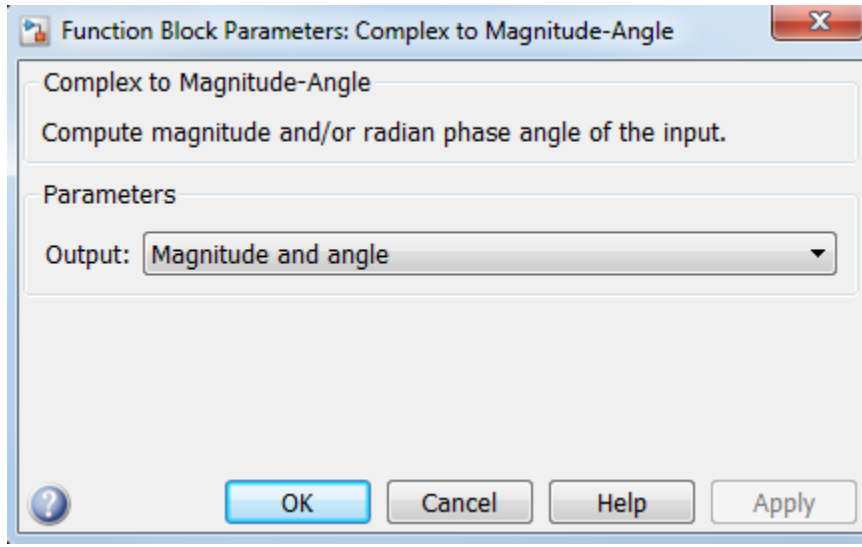
Description

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

Data Type Support

See the preceding description.

Parameters and Dialog Box



Output

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

Sample time

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

Characteristics

Data Types	Double | Single
Sample Time	Inherited from driving block
Direct Feedthrough	Yes

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Complex to Real-Imag

Output real and imaginary parts of complex input signal

Library

Math Operations

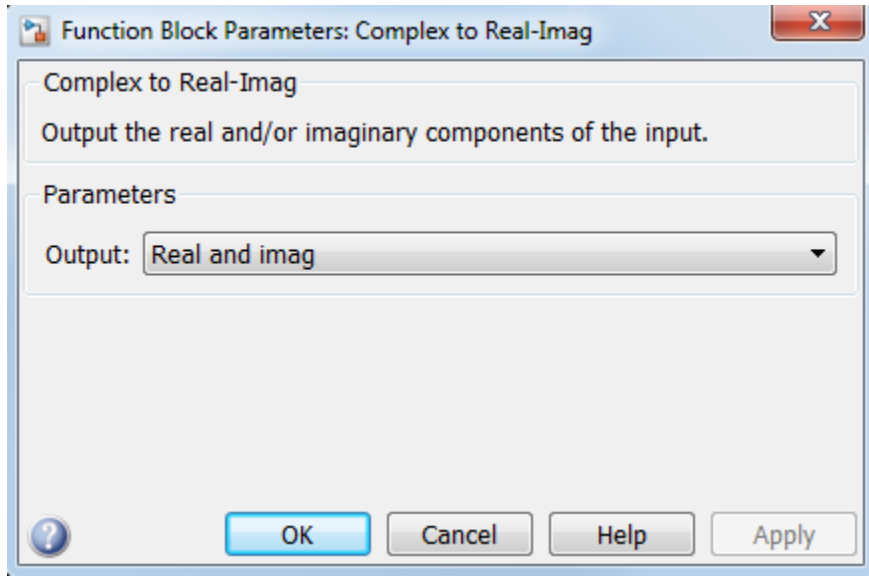
Description

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

Data Type Support

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

Parameters 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

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

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Inherited from driving block

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

Introduced before R2006a

Configurable Subsystem

Represent any block selected from user-specified library of blocks

Library

Ports & Subsystems

Description

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

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

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

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

Note that you cannot break library links in a configurable subsystem because Simulink uses those links to reconfigure the subsystem when you choose a new configuration.

Breaking links would be useful only if you do not intend to reconfigure the subsystem. In this case, you can replace the configurable subsystem with a nonconfigurable subsystem that implements the permanent configuration.

Creating a Master Configurable Subsystem

To create a master configurable subsystem:

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

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

   The context menu displays a submenu listing the blocks that the subsystem can represent.
8. Select the block that you want the subsystem to represent by default.
9. Save the library.

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

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

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

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

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

Creating an Instance of a Configurable Subsystem

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

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

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

Setting Instance Block Parameters

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

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

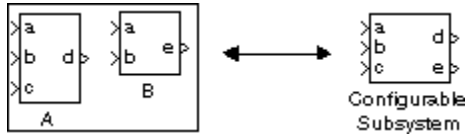
Mapping I/O Ports

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

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

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

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

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

During conversion, Simulink performs the following operations:

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

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

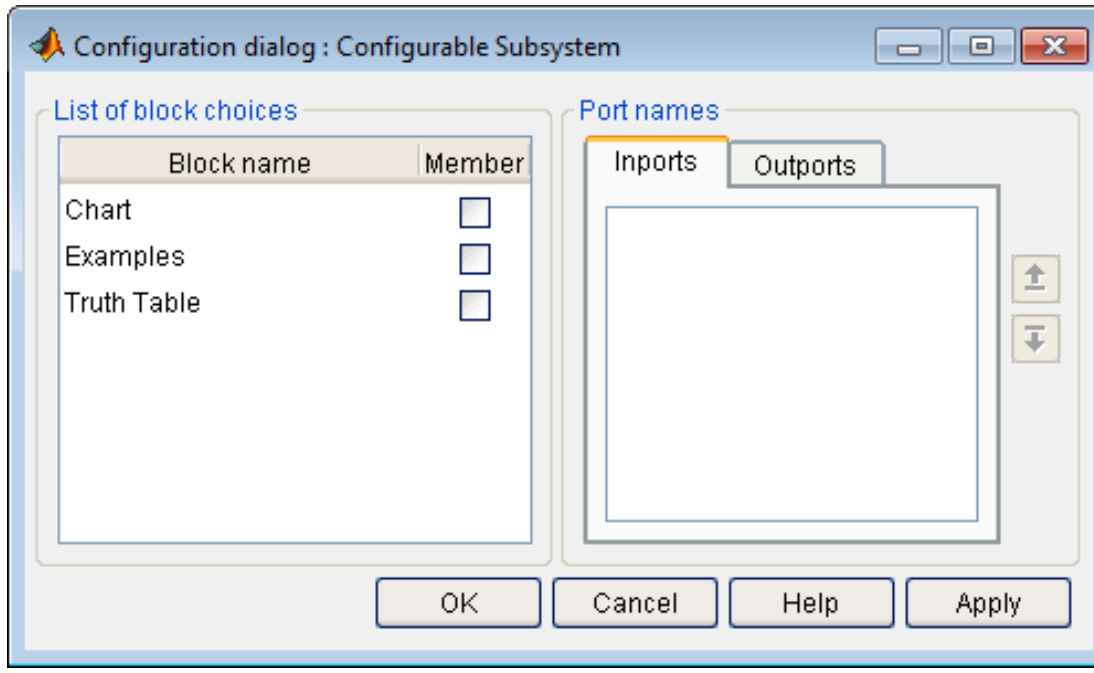
See `Variant Subsystem` for more information on variant choices.

Data Type Support

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

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

Parameters 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.

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

Introduced before R2006a

Constant

Generate constant value

Library

Sources

Description

The Constant block generates a real or complex constant value.

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

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

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

Data Type Support

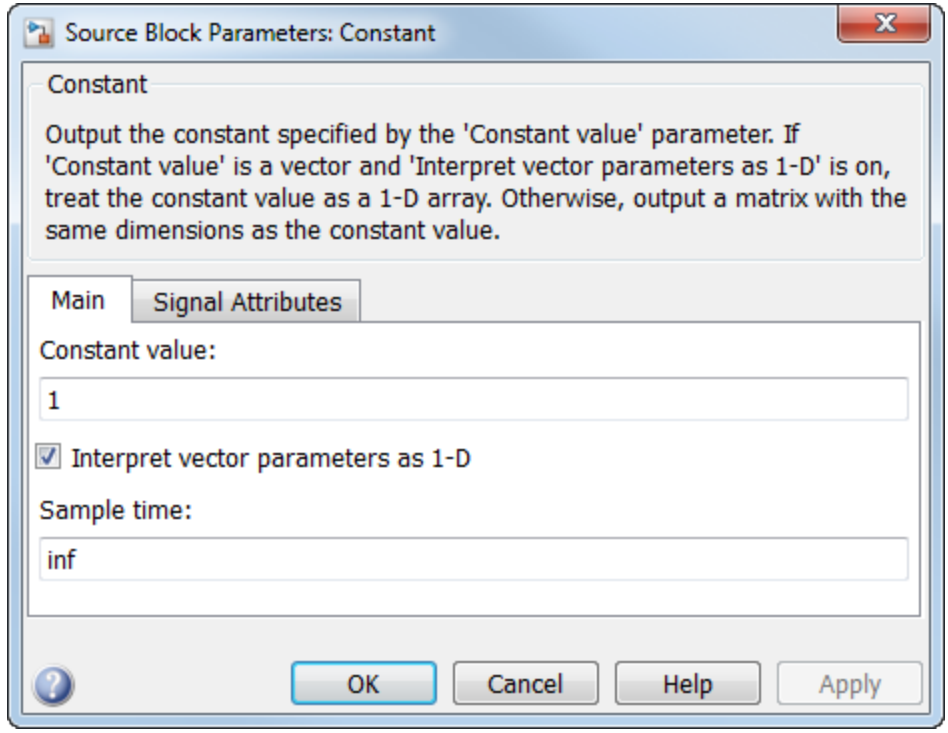
By default, the Constant block outputs a signal whose data type and complexity are the same as those of the **Constant value** parameter. However, you can specify the output to be any data type that Simulink supports, including fixed-point and enumerated data types. The `Enumerated Constant` block can be more convenient than the Constant block for outputting a constant enumerated value. You can also use a bus object as the output data type, which can help to simplify a model (see “Bus Support” on page 1-235 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



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 1-235.
- 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 6-101 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 "Control Block Parameter Value Ranges") 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 "Control Block Parameter Value Ranges") 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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**

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 6-101 for the command-line information.

See Also

See "Control Signal 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 “Control Signal 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 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.

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.

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.

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.

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.

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.

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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. The block output is a nonvirtual bus signal.

You can use the `Simulink.Bus.createMATLABStruct` to create a full structure that corresponds to a bus.

You can use `Simulink.Bus.createObject` to create a bus object from a MATLAB structure.

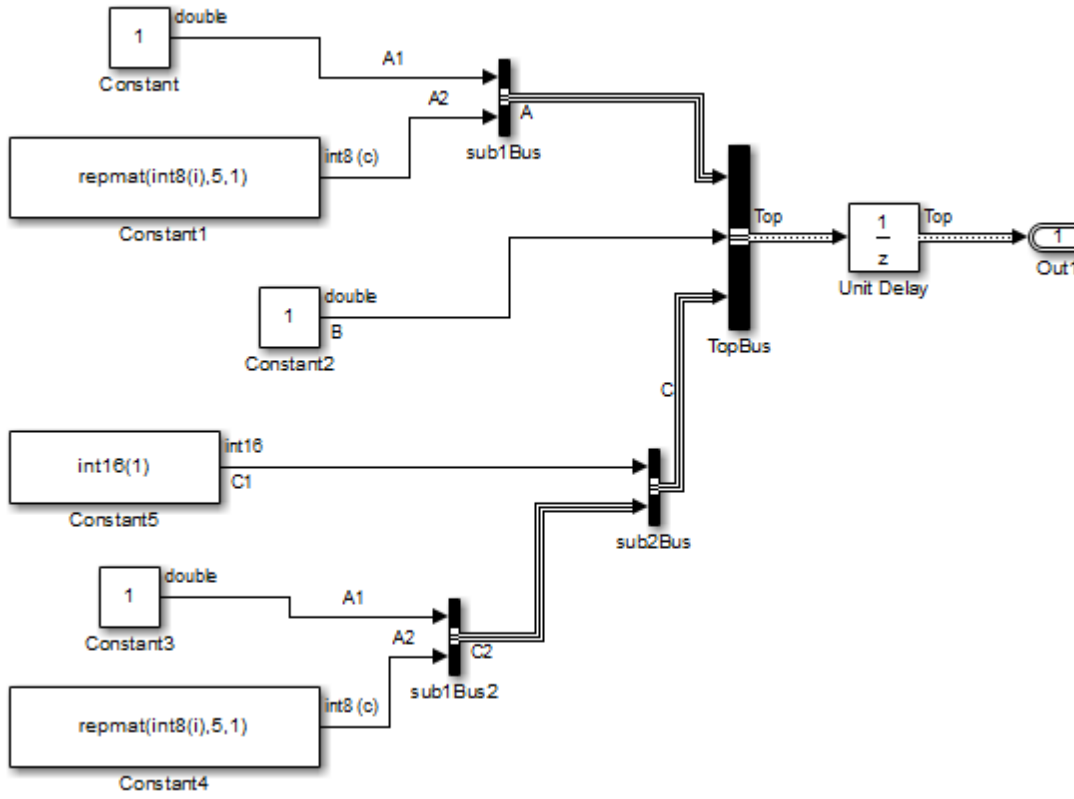
If the signal elements in the output bus use numeric data types other than `double`, you can specify the structure fields by using typed expressions such as `uint16(37)` or untyped expressions such as `37`. To control the field data types, you can use the bus object as the data type of a `Simulink.Parameter` object. To decide whether to use typed or untyped expressions, see "Control Data Types of Initial Condition Structure Fields".

Example of Using a Bus Object for a Constant Block

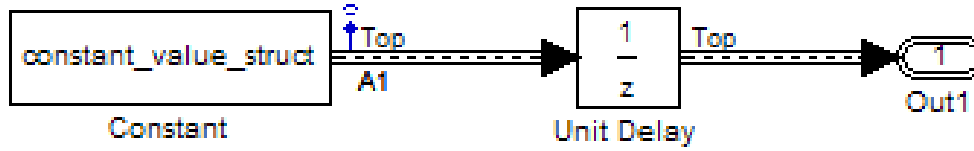
The following example illustrates how using a bus object as an output data type for a Constant block can help to simplify a model.

1 Open the ex_busic model and update it.

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



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



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

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

```
constant_value_struct
```

```
constant_value_struct =

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

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

```
logsout.get('A1').Values.B.Data
```

Group Constant Signals into an Array of Buses

You can use a `Constant` block to compactly represent multiple constant-valued signals as an array of buses. You can use this technique to reduce the number of signal lines in a model and the number of variables that the model uses, especially when the model repeats an algorithm with different parameter values.

To generate a constant-valued array of bus signals, use an array of MATLAB structures in a `Constant` block. The block output is an array of buses, and each field in the array of structures provides the simulation value for the corresponding signal element.

You can also use an array of structures to specify the `Value` property of a `Simulink.Parameter` object, and use the parameter object in a `Constant` block.

1. Open the example model ex_constantbus_array.

   The variables `ParamBus` and `const_param_struct` appear in the base workspace. The variable `const_param_struct` contains a structure whose field names match the elements of the bus type that `ParamBus` defines.

2. Update the diagram to view the signal line widths.

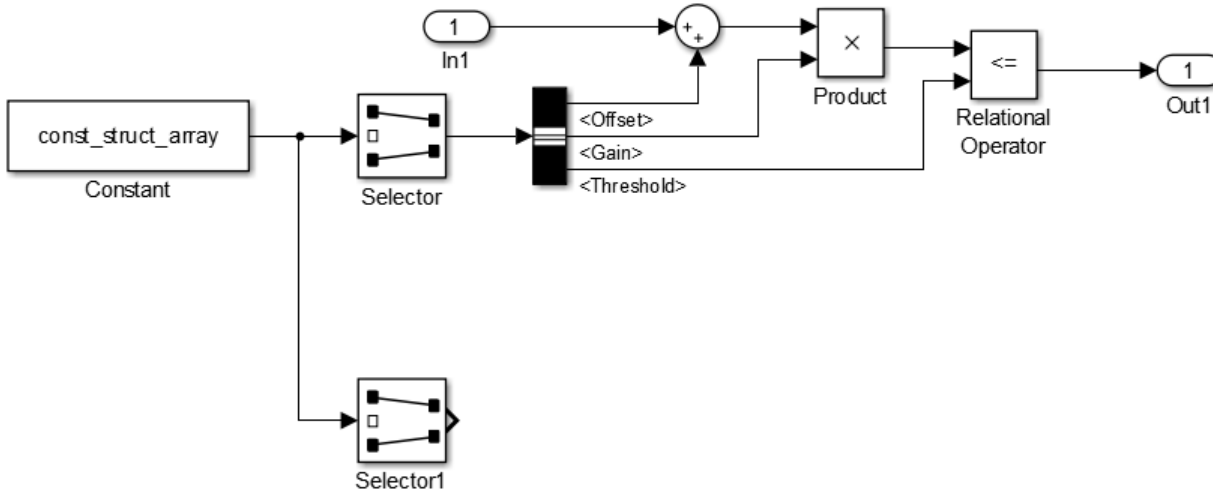
   The output of the `Constant` block is a single scalar bus of type `ParamBus`. The structure variable `const_param_struct` specifies the constant value in the block.

3. At the command prompt, create an array of two structures by copying the structure `const_param_struct`. Customize the field values by indexing into the individual structures in the array.

```
const_struct_array =...
 [const_param_struct const_param_struct];

const_struct_array(2).Offset = 158;
const_struct_array(2).Gain = 3.83;
const_struct_array(2).Threshold = 1039.77

const_struct_array =

1x2 struct array with fields:

    Offset
    Gain
    Threshold
```

4. In the `Constant` block dialog box, set **Constant value** to `const_struct_array`.

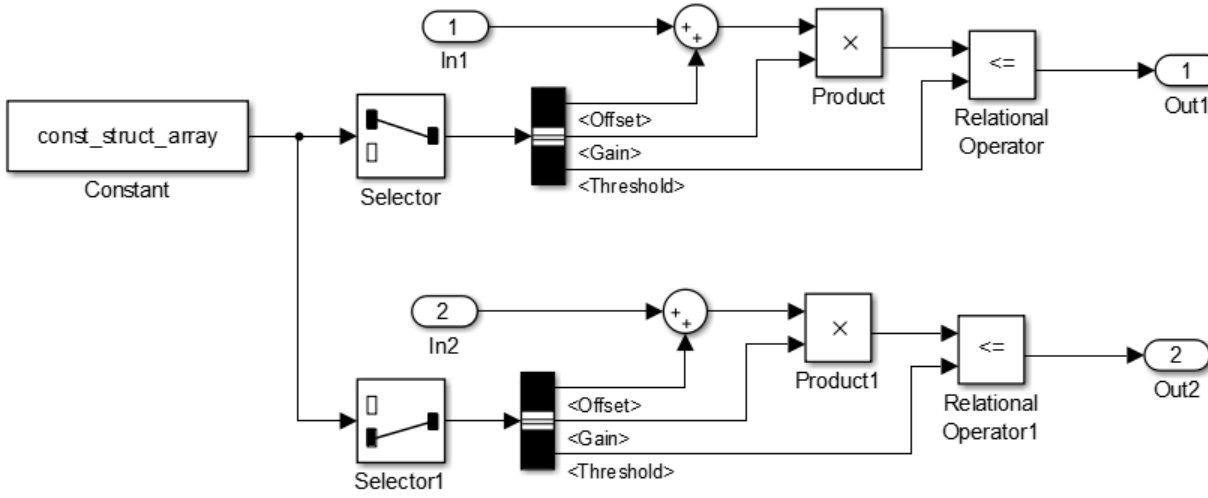
5. Add two `Selector` blocks to the model, and connect the `Constant` block as shown.



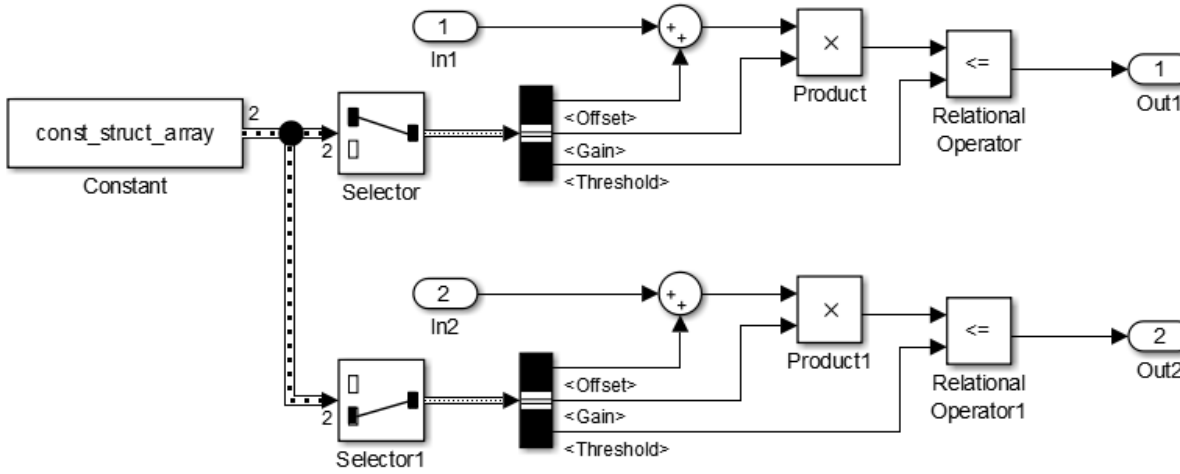
6 In the Selector block dialog box, set **Index** to 1 and **Input port size** to 2.

7 In the Selector1 block dialog box, set **Index** to 2 and **Input port size** to 2.

8 Copy the block algorithm in the model, and connect the blocks as shown.



9 Update the diagram. The signal line width and style indicate that the output of the `Constant` block is an array of buses. The `Selector` blocks each extract one of the buses in the array.



Each copy of the algorithm uses the constant values provided by the corresponding structure in the variable `const_struct_array`.

To create an array of structures for a bus that uses a large hierarchy of signal elements, consider using the function `Simulink.Bus.createMATLABStruct`. You can also use this technique to create an array of structures if you do not have a scalar structure that you can copy.

Setting Configuration Parameters to Support Using a Bus Object Data Type

To enable the use of a bus object as an output data type, before you start a simulation, set 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`.
- Set **Configuration Parameters > All Parameters > Underspecified initialization detection** to `simplified`. For more information, see "Underspecified initialization detection"

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

Examples

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

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

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	N/A
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Enumerated Constant` | `Simulink.Parameter`

More About

- "Bus Objects"

Introduced before R2006a

Coulomb and Viscous Friction

Model discontinuity at zero, with linear gain elsewhere

Library

Discontinuities

Description

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

The block output matches the MATLAB result for:

```
y = 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

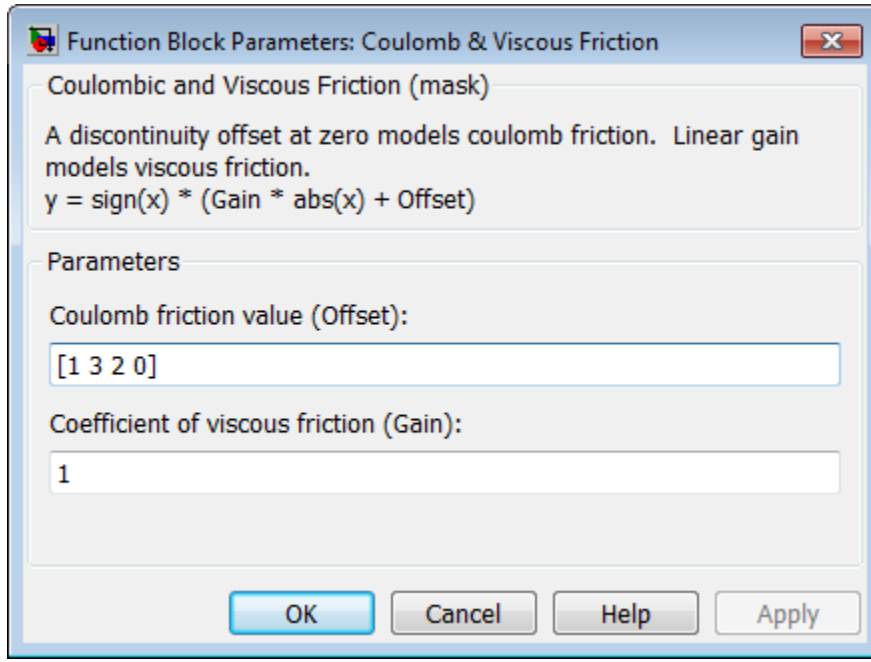
The Coulomb and Viscous Friction block supports real inputs of the following data types:

- 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.

Parameters and Dialog Box



Coulomb friction value

Specify the offset that applies to all input values.

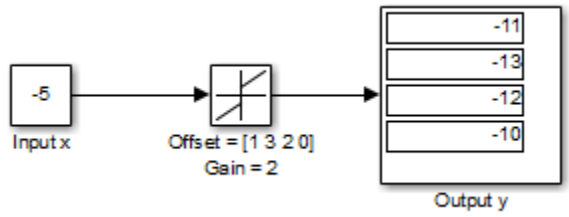
Coefficient of viscous friction

Specify the signal gain for nonzero input values.

Examples

Scalar Input

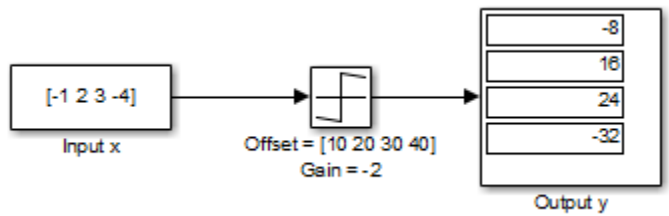
Suppose that you have the following model:



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

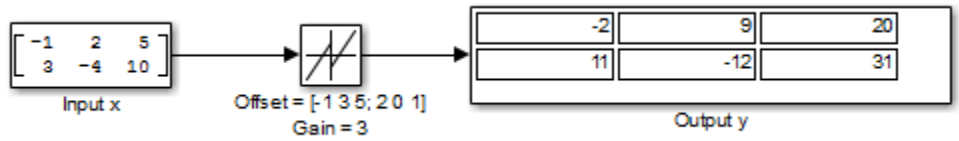
Suppose that you have the following model:



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

Matrix Input

Suppose that you have the following model:



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

Characteristics

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

Introduced before R2006a

Counter Free-Running

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

Library

Sources

Description

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

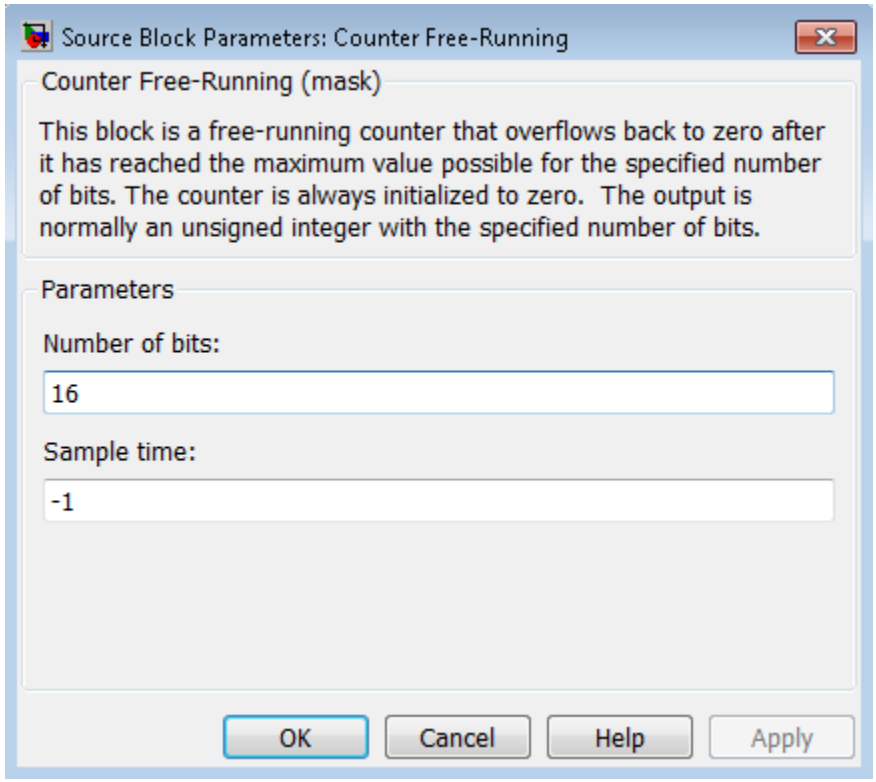
After overflow, the counter always initializes to zero. However, if you select the global doubles override, the Counter Free-Running block does not wrap back to zero.

Data Type Support

The Counter Free-Running block outputs an unsigned integer.

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

Parameters 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)`	$\text{uint8}(2^{\text{uint8(8)}} - 1)$	254

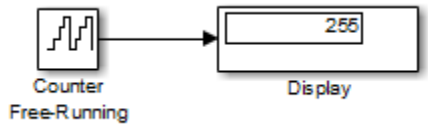
Sample time

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:

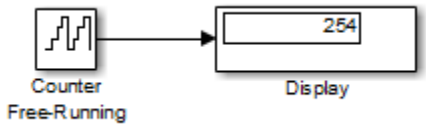
Parameter	Setting
Stop time	255
Type	Fixed-step
Solver	discrete (no continuous states)
Fixed-step size	1

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

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

Characteristics

Data Types	Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Counter Limited`

Introduced before R2006a

Counter Limited

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

Library

Sources

Description

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

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

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

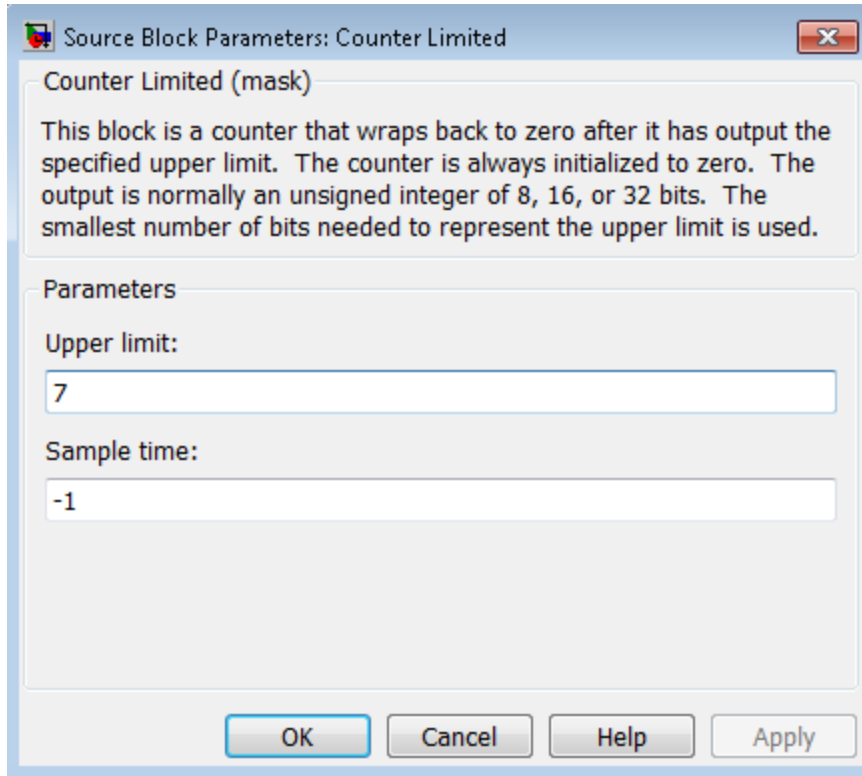
The output is an unsigned integer of 8, 16, or 32 bits, with the smallest number of bits needed to represent the upper limit.

Data Type Support

The Counter Limited block outputs an unsigned integer.

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

Parameters 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

Data Types	Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

`Counter Free-Running`

Introduced before R2006a

Dashboard Scope

Trace signals during simulation

Library

Dashboard

Description 

The `Dashboard Scope` block displays connected signals during simulation on a scope display.

To view data from signals, double-click the `Dashboard Scope` block to open the dialog box. Select signals in the model. The signals appear in the **Connection** table. Select the check box next to each signal you want to display in the scope. Click **Apply** to connect the signals.

You can also add data cursors to the `Dashboard Scope` to inspect signal data. To add a data cursor, select the `Dashboard Scope` block and right-click. Select **Data Cursors** > **One** from the menu.

To change zoom and pan modes, select the `Dashboard Scope` block, right-click, and select the zoom or pan mode you want.

Limitations

The `Dashboard Scope` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.

If you turn off streaming for a signal connected to the `Dashboard Scope` block, then signal data does not stream to the block. To view signal data again, double-click the `Dashboard Scope` block and reconnect the signal.

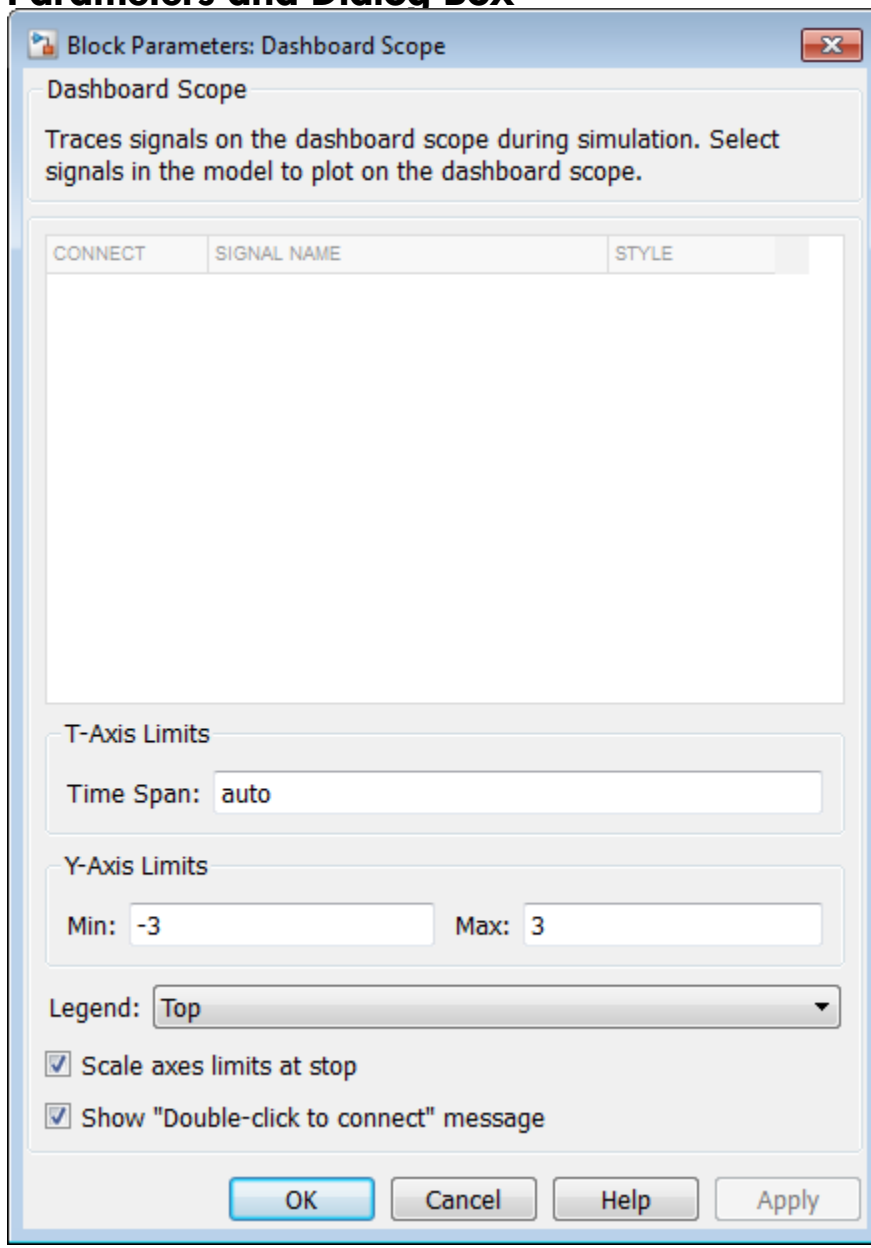
The External simulation mode is not supported for the `Dashboard Scope` block.

Data Type Support

The `Dashboard Scope` block accepts real (not complex) signals of any data type that Simulink supports, including enumerated data types.

For more information on data types in Simulink, see “Data Types Supported by Simulink”.

Parameters and Dialog Box



Connection

Select signals to connect and display.

To view data from signals, double-click the `Dashboard Scope` block to open the dialog box. Select signals in the model. The signals appear in the **Connection** table. Select the check box next to each signal you want to display in the scope. Click **Apply** to connect the signals.

Settings

The table has a row for the signals connected to the block. If there are no signals selected in the model or the block is not connected to any signals, then the table is empty.

T-Axis Limits

Horizontal axis time span.

Settings

Default: `auto`

Specify this number as a finite, real, double, scalar value. Specify `auto` for the `Dashboard Scope` to set the time span to the model simulation stop time.

Y-Axis Limits

Vertical axis range.

Settings

Default: –3 and 3

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

Dependencies

The **Min** value must be less than the **Max** value.

Legend

Position of the line legend.

Settings

Default: Top

Top
: Show the legend at the top of the plot.

Right
: Show the legend at the right of the plot.

Hide
: Do not show the legend.

Scale axes limits at stop

Perform a fit-to-view on the data displayed in the scope when the simulation has stopped.

Settings

Default: On

 On
: Perform a fit-to-view on the data displayed in the scope when the simulation has stopped.

 Off
: Do not perform a fit-to-view on the data displayed in the scope when the simulation has stopped.

Show "Double-click to connect" message

Show instructional text if the block is not connected. You can clear the check box to hide the text when the block is not connected.

Settings

Default: On

 On
: Show the instructional text if the block is not connected.

 Off

Do not show the instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

Introduced in R2015a

Data Store Memory

Define data store

Library

Signal Routing

Description

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

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

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

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

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

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 these data types:

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

The block does not support variable-size signals.

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

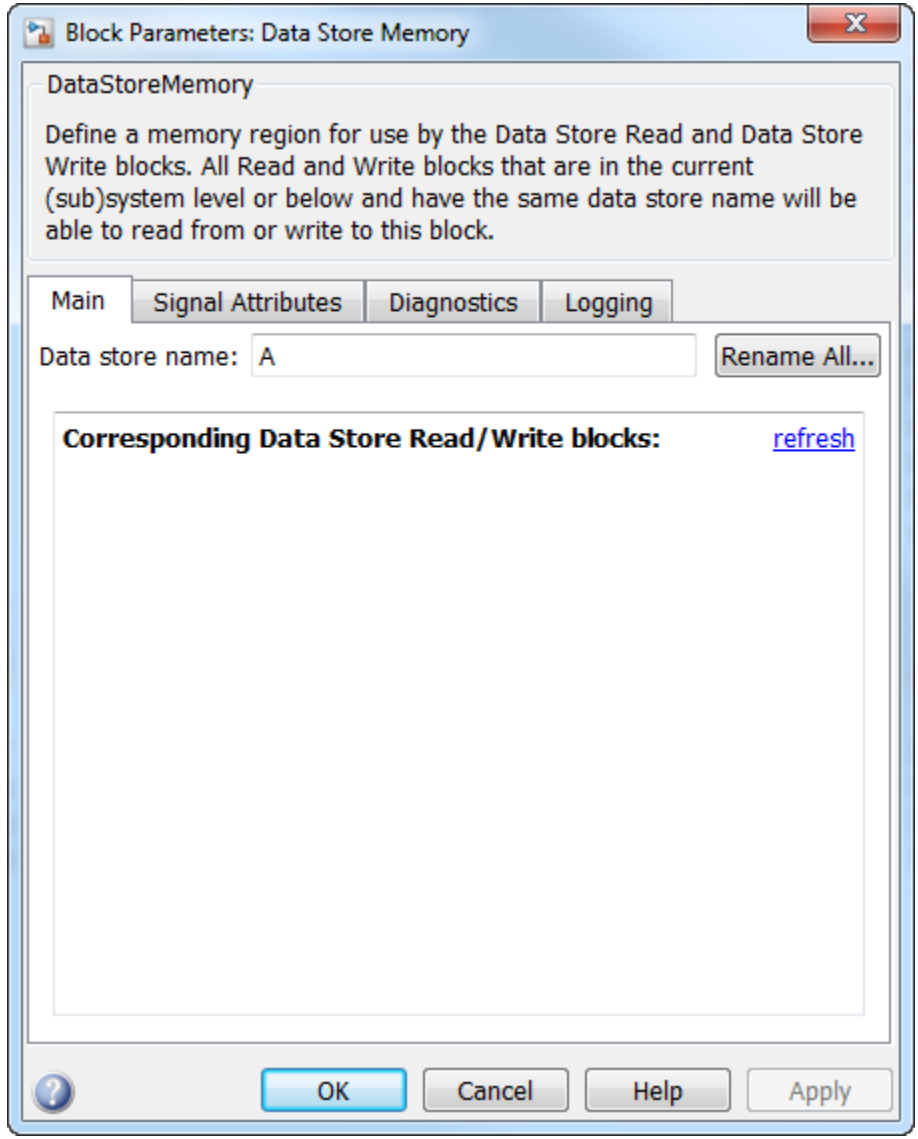
For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

For more information, see "Data Types Supported by Simulink" 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:



Data store name

Specify a name for the data store you are defining with this block. Data Store Read and Data Store Write blocks with the same name can read from, and write to, the

data store initialized by this block. The name can represent a Data Store Memory block or a sign object defined to be a data store.

Rename All

Rename the data store everywhere the `Data Store Read` and `Data Store Write` blocks use it in a model.

You cannot use **Rename All** to rename a data store if you:

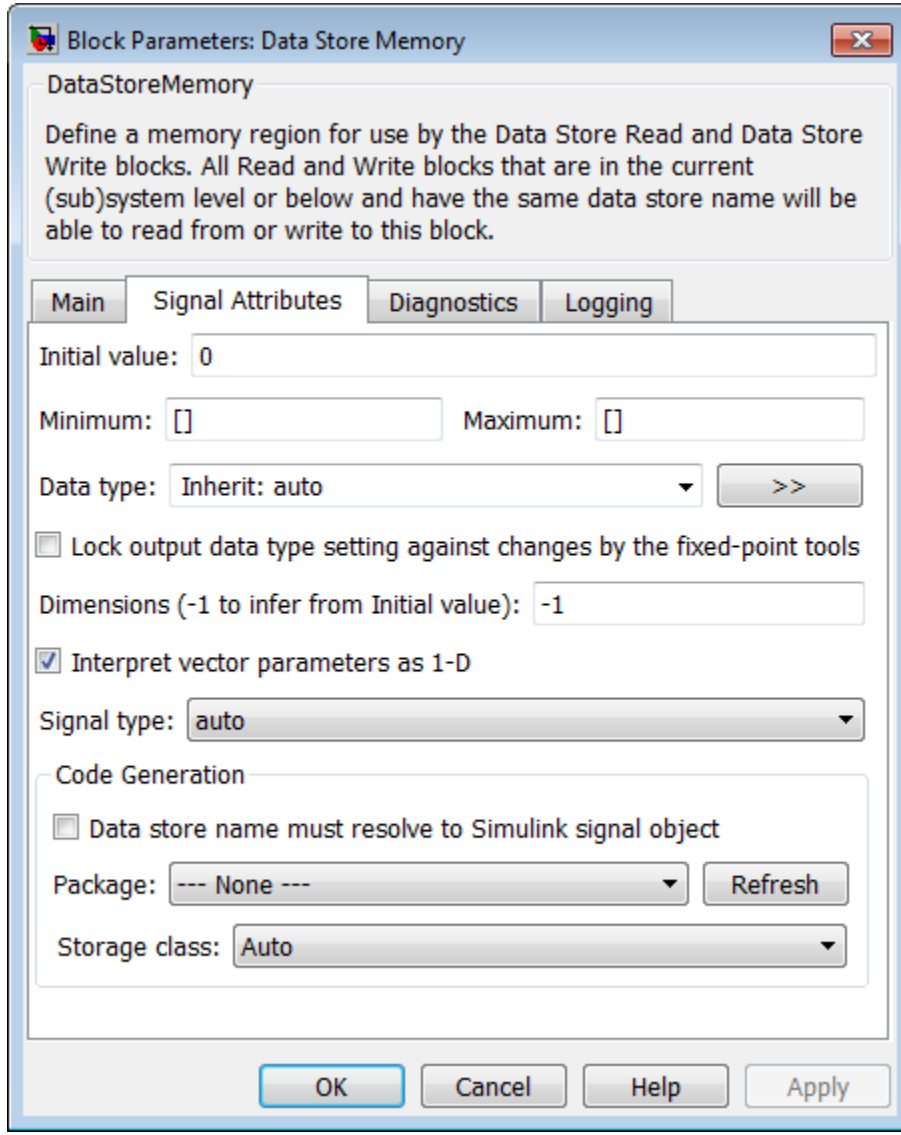
- Use a `Simulink.Signal` object in a workspace to control the code generated for the data store
- Use a `Simulink.Signal` object instead of a `Data Store Memory` block to define the data store

You must instead rename the corresponding `Simulink.Signal` object from Model Explorer. For an example, see "Rename Data Store Defined by Signal Object".

Corresponding Data Store Read/Write blocks

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

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



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.

To use this block to initialize a nonvirtual bus signal, specify the initial value as a MATLAB structure and set the model configuration parameter "Underspecified initialization detection" to `Simplified`. For more information about initializing nonvirtual bus signals using structures, see "Specify Initial Conditions for Bus Signals".

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). This number must be a finite real double scalar value.

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

Simulink uses the minimum value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges")
- 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`.

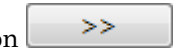
Simulink uses the maximum value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges")
- 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 "Control Signal Data Types".

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

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

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.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder® software, custom storage classes do not affect the generated code.

Storage class

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

TypeQualifier

Specify a storage type qualifier such as `const` or `volatile`.

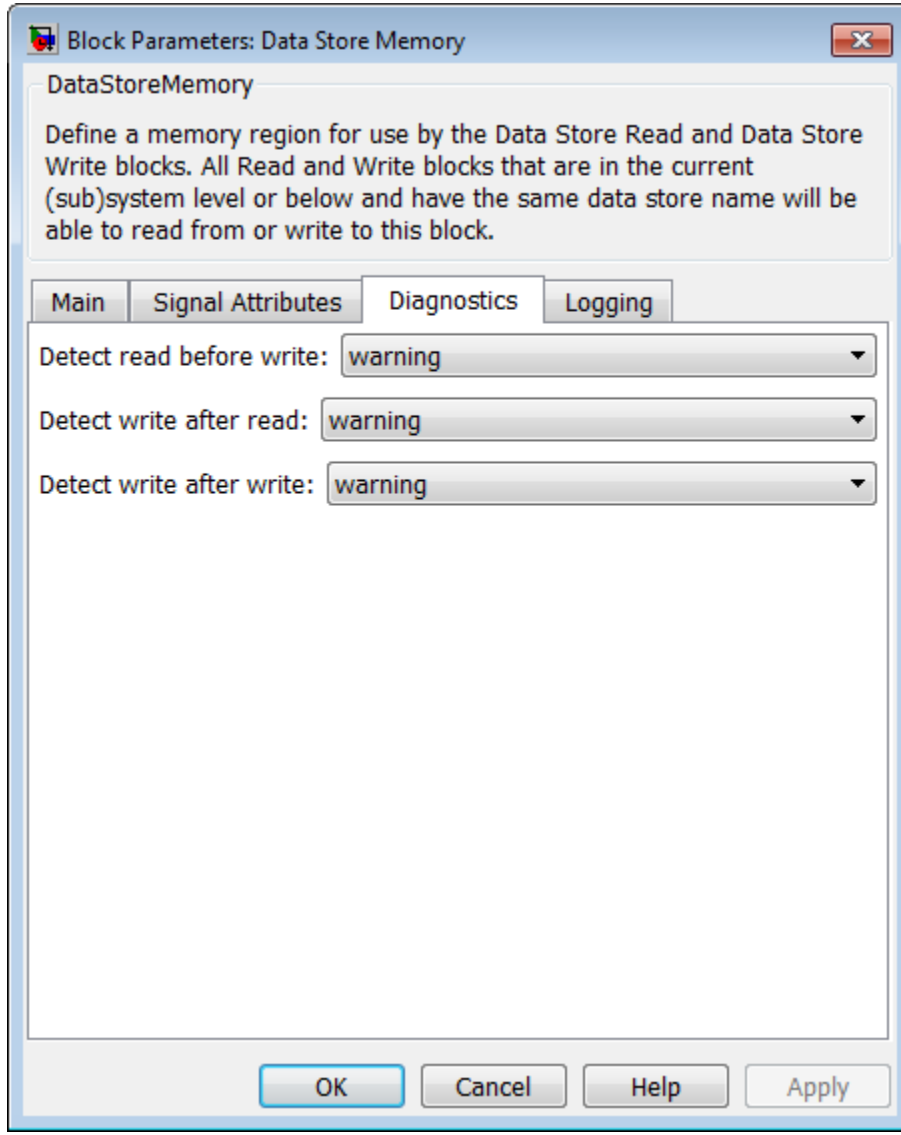
Setting **Storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use

an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

See “Discrete Block State Naming in Generated Code” in the Simulink Coder documentation for more information.

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



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.

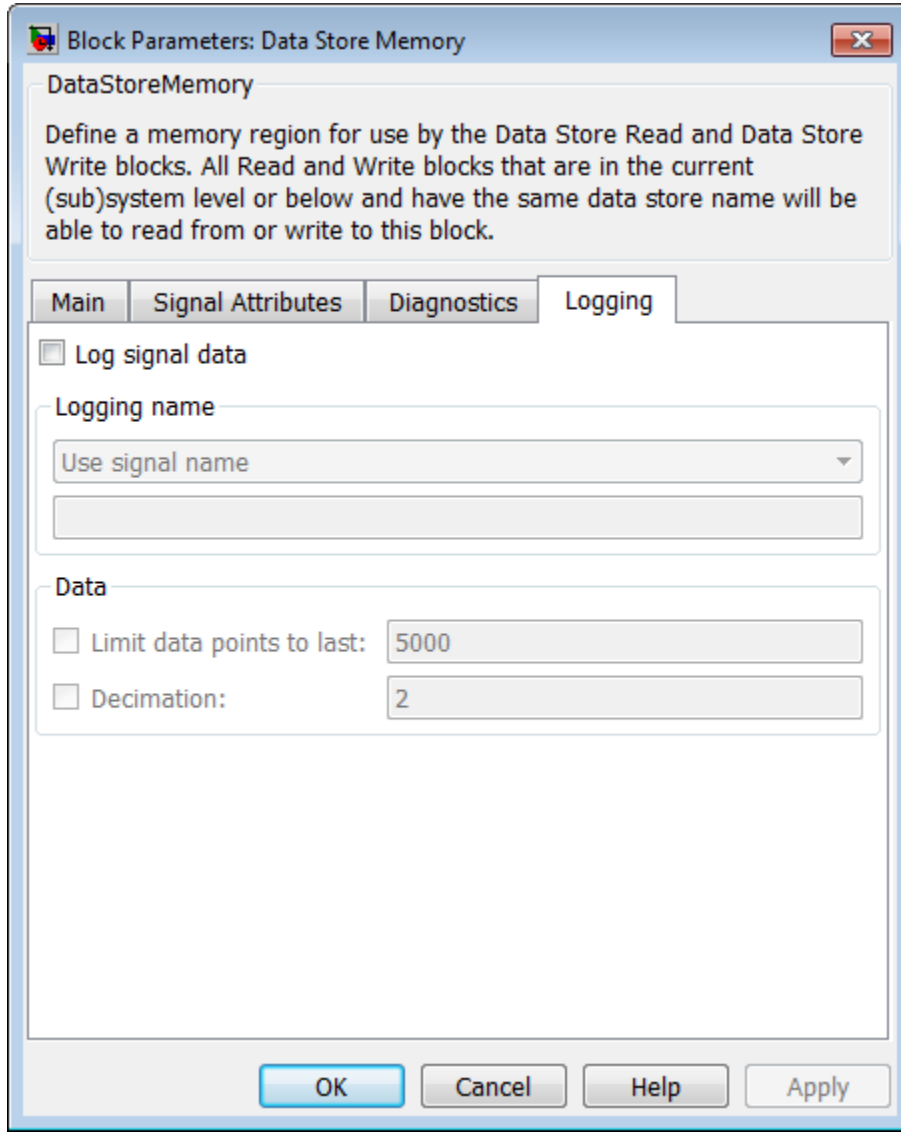
`warning`

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:



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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

See Also

- "Data Stores"
- "Access Data Stores with Simulink Blocks"
- `Data Store Read`

- `Data Store Write`
- “Log Data Stores”

Introduced before R2006a

Data Store Read

Read data from data store

Library

Signal Routing

Description

The Data Store Read block copies data from the named data store to its output. More than one Data Store Read block can read from the same data store.

The data store from which the data is read is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see “Data Stores” and `Data Store Memory`.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. See “Order Data Store Access” and “Data Store Diagnostics” for details.

Data Type Support

The Data Store Read block can output a real or complex signal of these data types:

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

- Bus

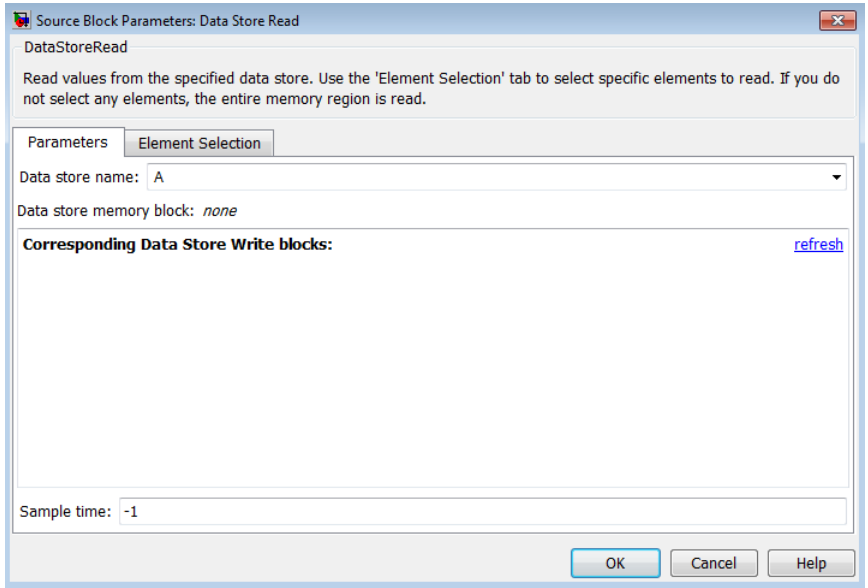
The block does not support variable-size signals.

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



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.

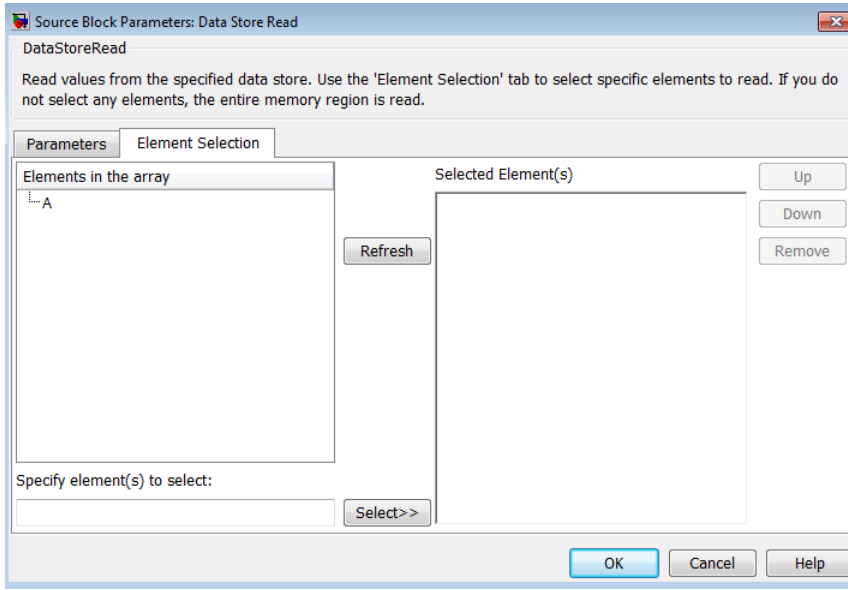
Data store write blocks

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:



Use the **Element Selection** pane to select a subset of the bus or matrix elements defined for the associated data store. The Data Store Read block icon reflects the elements that you specify. For details, see “Accessing Specific Bus and Matrix Elements”.

Elements in the array or **Signals in the bus** (Prompt is specific to the type of data.)

For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

For data stores with a bus data type, you can expand the tree to view the bus elements. For data stores with arrays, you can read the whole data store, or you can specify one or more elements of the whole data store.

You can select an element and then use one of the following approaches:

- Click **Select>>** to display that element (and all its subelements) in the **Selected element(s)** list.
- Use the **Specify element(s) to select** edit box to specify the bus or matrix elements that you want to select for reading. Then click **Select>>**.

To refresh the display to reflect modifications to the bus or matrix used in the data store, click **Refresh**.

Specify element(s) to select

Enter a MATLAB expression to define the specific element that you want to read. For example, for a data store named `DSM` that has maximum dimensions of `[3,5]`, you could enter expressions such as `DSM(2, 4)` or `DSM([1 3], 2)` in the edit box and then click **Select>>**.

To apply the element selection, click **OK**.

Selected Element(s)

Displays the elements that you select. The Data Store Read block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click **Up** or **Down**. Changing the order of the elements in the list changes the order of the ports. To remove an element, click **Remove**.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

See Also

- "Data Stores"
- "Rename Data Stores"
- "Access Data Stores with Simulink Blocks"
- `Data Store Memory`
- `Data Store Write`

Introduced before R2006a

Data Store Write

Write data to data store

Library

Signal Routing

Description

The Data Store Write block copies the value at its input to the named data store. Each write operation performed by a Data Store Write block writes over the data store, replacing the previous contents.

The data store to which this block writes is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see "Data Stores" and `Data Store Memory`. The size of the data store is set by the signal object or the Data Store Memory block that defines and initializes the data store. Each Data Store Write block that writes to that data store must write the same amount of data.

More than one Data Store Write block can write to the same data store. However, if two Data Store Write blocks attempt to write to the same data store during the same simulation step, results are unpredictable.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. For details, see "Order Data Store Access" and "Data Store Diagnostics".

You can log the values of a local or global data store data variable for all the steps in a simulation. For details, see "Log Data Stores".

Data Type Support

The Data Store Write block accepts a real or complex signal of these data types:

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

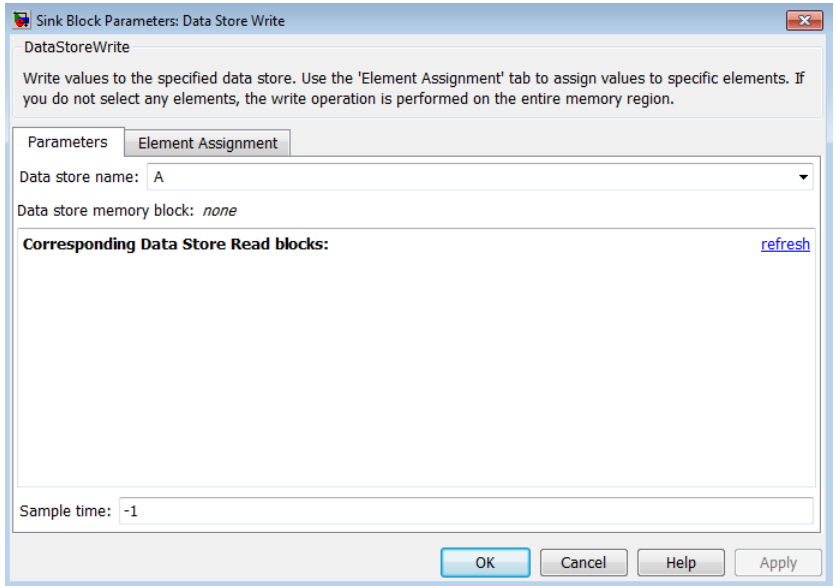
The block does not support variable-size signals.

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



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.

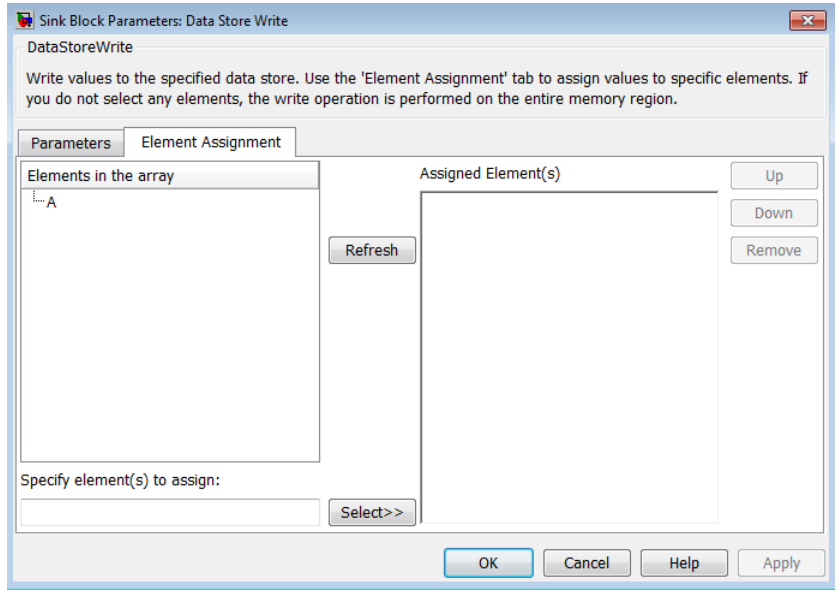
Data store read blocks

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:



Use the **Element Assignment** pane to assign a subset of the bus or matrix elements defined for writing to the associated data store. The Data Store Write block icon reflects the elements that you specify. For details, see “Accessing Specific Bus and Matrix Elements”.

Elements in the array or **Signals in the bus** (Prompt is specific to the type of data.)

For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

For data stores with a bus data type, you can expand the tree to view the bus elements. For data stores with arrays, you can write the whole data store, or you can assign one or more elements to the whole data store.

You can select an element and then use one of the following approaches:

- Click **Select>>** to display that element (and all its subelements) in the **Assigned element(s)** list.
- Use the **Specify element(s) to assign** edit box to specify the bus or matrix elements that you want to select for reading. Then click **Select>>**.

To refresh the display to reflect modifications to the bus or matrix used in the data store, click **Refresh**.

Specify element(s) to assign

Enter a MATLAB expression to define the specific element that you want to write. For example, for a data store named `DSM` that has maximum dimensions of `[3,5]`, you could enter expressions such as `DSM(2, 4)` or `DSM([1 3], 2)` in the edit box. Then click **Select>>**.

To apply the element selection, click **OK**.

Assigned Element(s)

Displays the elements that you selected for assignment. The Data Store Write block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click **Up** or **Down**. Changing the order of the elements in the list changes the order of the ports. To remove an element, click **Remove**.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter

Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

See Also

- “Data Stores”
- “Rename Data Stores”
- “Access Data Stores with Simulink Blocks”
- `Data Store Memory`
- `Data Store Read`
- “Log Data Stores”

Introduced before R2006a

Data Type Conversion

Convert input signal to specified data type

Library

Signal Attributes

Description

The Data Type Conversion block converts an input signal of any Simulink data type to the data type that you specify.

The input can be any real- or complex-valued signal. If the input is real, the output is real. If the input is complex, the output is complex.

Note To control the output data type by specifying block parameters, or to inherit a data type from a downstream block, use the `Data Type Conversion` block. To inherit a data type from a different signal in the model, use the `Data Type Conversion Inherited` block.

Convert Fixed-Point Signals

When you convert between fixed-point data types, the **Input and output to have equal** parameter controls block behavior. If neither input nor output use fixed-point scaling, because they are not of a fixed-point data type or have trivial fixed-point scaling, this parameter does not change the behavior of the block. For more information about fixed-point numbers, see “Fixed-Point Numbers in Simulink”.

To convert a signal from one data type to another by attempting to preserve the real-world value of the input signal, select `Real World Value (RWV)`, the default setting. The block accounts for the scaling of the input and output and, within the limits of the specified data types, attempts to generate an output of equal real-world value.

To change the real-world value of the input signal by performing a scaling reinterpretation of the stored integer value, select `Stored Integer (SI)`. Within the limits of the specified data types, the block attempts to preserve the stored integer value of the signal during conversion. A best practice is to specify input and output data types using the same word length and signedness so that the block changes only the scaling of the signal. Specifying a different signedness or word length for the input and output could produce unexpected results such as range loss or unexpected sign extensions. For an example, see "Reinterpret Signal Using a Fixed-Point Data Type" on page 1-314.

If you select `Stored Integer (SI)`, the block does not perform a lower-level bit reinterpretation of a floating-point input signal. For example, if the input is of the data type `single` and has value `5`, the bits that store the input in memory are given in hexadecimal by the following command.

```
num2hex(single(5))
```

```
40a00000
```

However, the `Data Type Conversion` block does not treat the stored integer value as `40a00000`, but instead as the real-world value, `5`. After conversion, the stored integer value of the output is `5`.

Cast Enumerated Signals

Use a Data Type Conversion block to cast enumerated signals as follows:

1. To cast a signal of enumerated type to a signal of any numeric type.

   The underlying integers of all enumerated values input to the Data Type Conversion block must be within the range of the numeric type. Otherwise, an error occurs during simulation.

2. To cast a signal of any integer type to a signal of enumerated type.

   The value input to the Data Type Conversion block must match the underlying value of an enumerated value. Otherwise, an error occurs during simulation.

You can enable the block's **Saturate on integer overflow** parameter so that Simulink uses the default value of the enumerated type when the value input to the block does not match the underlying value of an enumerated value. See "Type Casting for Enumerations".

You cannot use a Data Type Conversion block in the following cases:

- To cast a non-integer numeric signal to an enumerated signal.
- To cast a complex signal to an enumerated signal, regardless of the data types of the complex signal's real and imaginary parts.

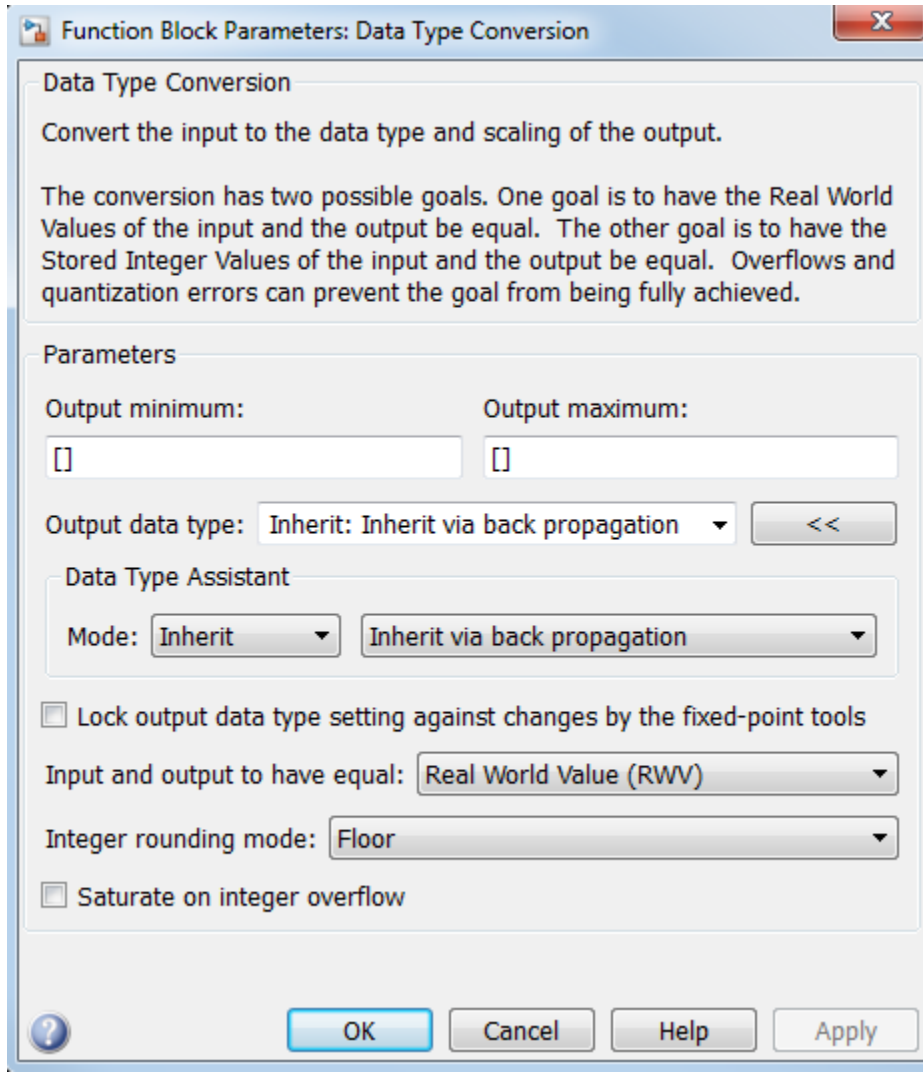
See "Simulink Enumerations" for information on working with enumerated types.

Data Type Support

The Data Type Conversion block handles any data type that Simulink supports, including fixed-point and enumerated data types.

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

Parameters and Dialog Box



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 “Control Signal Data Types”.

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

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

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 must be equal, in the context of fixed point data representation.

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

For the command-line information, see “Block-Specific Parameters” on page 6-101.

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Sample time

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

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the minimum to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 6-101 for the command-line information.

See Also

For more information, see "Control Signal 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 6-101 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.

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.

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**

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.

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.

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.

See Also

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

Examples

Real-World Values and Stored Integers

The example model ex_data_type_conversion_rwv_si uses `Data Type Conversion` blocks to show the meaning of the real-world value and the stored integer of a signal. For

basic information about fixed-point scaling, see “Scaling” in the Fixed-Point Designer documentation.

Conversion Between Fixed-Point Data Types

The Fixed-Point Constant block represents the real-world value `15` by using a fixed-point data type with binary-point scaling 2^{-5}. Due to the scaling, the output signal uses a stored integer value of `480`.

The model uses `Data Type Conversion` blocks to convert the signal to a fixed-point data type with binary-point scaling 2^{-2}.

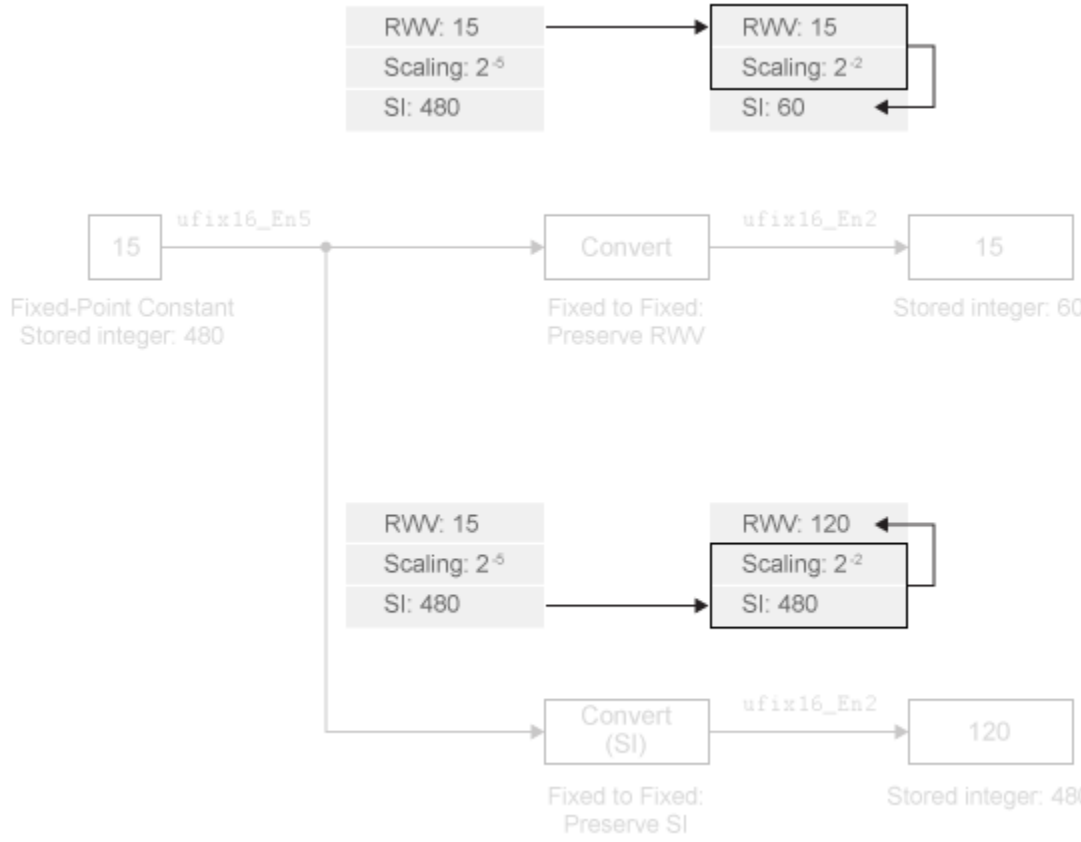
- The Fixed to Fixed: Preserve RWV block converts the input signal by preserving the real-world value, `15`. The parameter **Input and output to have equal** is set to `Real World Value (RWV)`.

  The output signal has the same real-world value as the input, that is, `15`. Due to the fixed-point scaling, the output uses a stored integer value of `60`.

- The Fixed to Fixed: Preserve SI block converts the input signal by preserving the stored integer value, `480`. The parameter **Input and output to have equal** is set to `Stored Integer (SI)`.

  The output signal uses the same stored integer value as the input, that is, `480`. Due to the fixed-point scaling, the output has a real-world value of `120`.

The figure shows the conversion mechanism for the two blocks.



Conversion Between Floating-Point and Fixed-Point Data Type

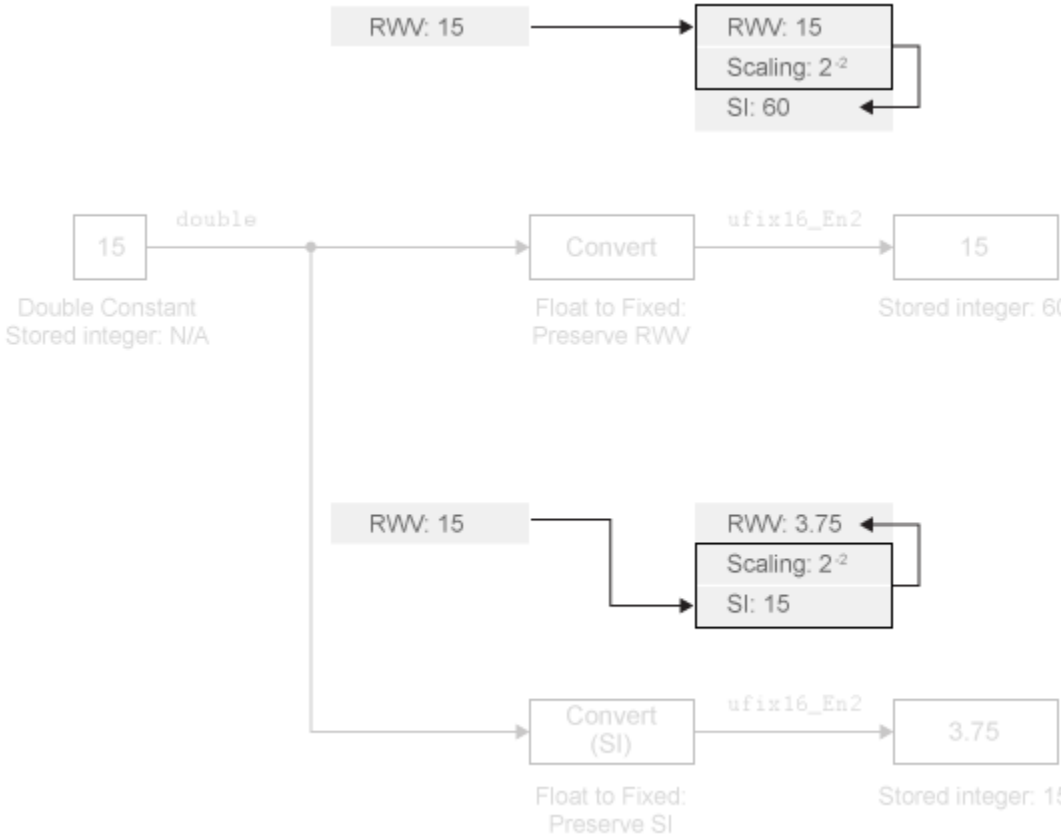
The Double Constant block represents the real-world value `15` by using the floating-point data type `double`. The output signal does not use fixed-point scaling.

The model uses `Data Type Conversion` blocks to convert the `double` signal to a fixed-point data type with binary-point scaling 2^{-2}.

- The Float to Fixed: Preserve RWV block converts the input signal by preserving the real-world value, `15`. The output signal has the same real-world value. Due to the fixed-point scaling, the output uses a stored integer value of `60`.

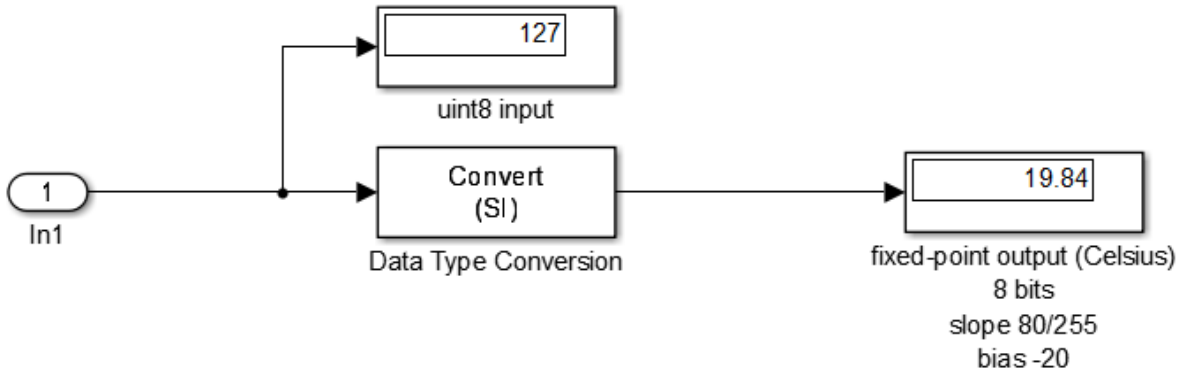
- The Float to Fixed: Preserve SI block converts the input signal by attempting to preserve the stored integer value. However, the block does not use the underlying bits that store the floating-point signal in memory. Instead, the block uses the real-world value of the input, `15`, as the stored integer of the output signal. Due to the fixed-point scaling, the real-world value of the output is `3.75`.

The figure shows the conversion mechanism for the two blocks. The blocks also use these mechanisms if the input uses the floating-point data type `single`.



Reinterpret Signal Using a Fixed-Point Data Type

Suppose your hardware uses the data type `uint8` to store data from a temperature sensor. Also suppose that the minimum stored integer value `0` represents `–20` degrees Celsius while the maximum `255` represents `60` degrees. The following model uses a `Data Type Conversion` block to convert the stored integer value of the sensor data to degrees Celsius.



The `Data Type Conversion` block parameter **Input and output to have equal** is set to `Stored Integer (SI)`. The block output signal is of a fixed-point data type with word length `8`, slope `80/255`, and bias `-20`.

The `Data Type Conversion` block reinterprets the integer input, `127`, as a Celsius output, `19.84` degrees. The block output uses the specified slope and bias to scale the stored integer of the input.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Data Type Conversion Inherited` | `Data Type Propagation`

Related Examples

- “Control Signal Data Types”

More About

- “About Data Types in Simulink”
- “Fixed Point”

Introduced before R2006a

Data Type Conversion Inherited

Convert from one data type to another using inherited data type and scaling

Library

Signal Attributes

Description

The Data Type Conversion Inherited block forces dissimilar data types to be the same. The first input is used as the reference signal and the second input is converted to the reference type by inheriting the data type and scaling information. (See “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

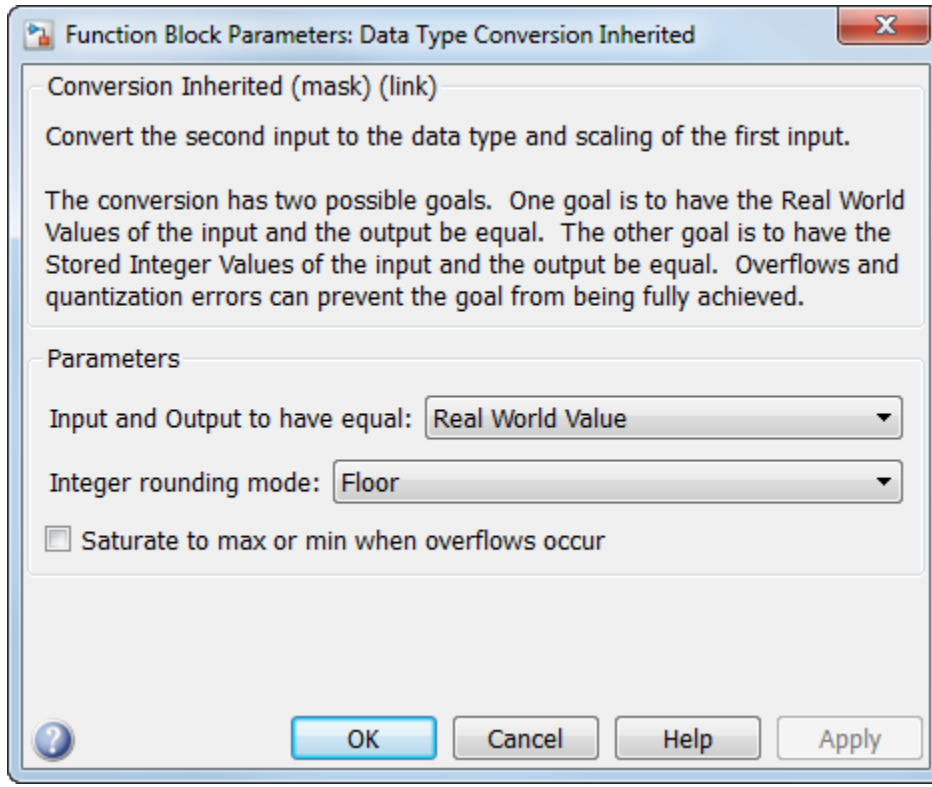
The Data Type Conversion Inherited block accepts signals of the following data types:

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

- Enumerated

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

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

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

See Also

`Data Type Conversion` | `Data Type Propagation`

Related Examples

- "Control Signal Data Types"

More About

- "About Data Types in Simulink"
- "Fixed Point"

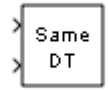
Introduced before R2006a

Data Type Duplicate

Force all inputs to same data type

Library

Signal Attributes



Description

The Data Type Duplicate block forces all inputs to have exactly the same data type. Other attributes of input signals, such as dimension, complexity, and sample time, are completely independent.

You can use the Data Type Duplicate block to check for consistency of data types among blocks. If all signals do not have the same data type, the block returns an error message.

The Data Type Duplicate block is typically used such that one signal to the block controls the data type for all other blocks. The other blocks are set to inherit their data types via back propagation.

The block is also used in a user created library. These library blocks can be placed in any model, and the data type for all library blocks are configured according to the usage in the model. To create a library block with more complex data type rules than duplication, use the Data Type Propagation block.

Data Type Support

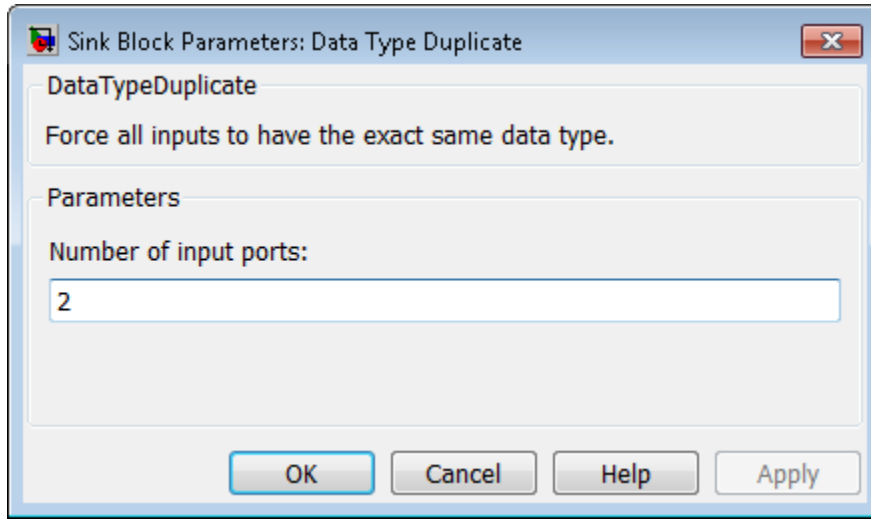
The Data Type Duplicate block accepts signals of the following data types:

- Floating point
- Built-in integer

- Fixed point
- Boolean
- Enumerated

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

Parameters and Dialog Box



Number of input ports

Specify the number of inputs to this block.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Inherited from driving block
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Data Type Conversion` | `Data Type Propagation`

Related Examples

- “Control Signal Data Types”

More About

- “About Data Types in Simulink”
- “Fixed Point”

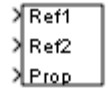
Introduced before R2006a

Data Type Propagation

Set data type and scaling of propagated signal based on information from reference signals

Library

Signal Attributes



Description

The Data Type Propagation block allows you to control the data type and scaling of signals in your model. You can use this block in conjunction with fixed-point blocks that have their **Output data type** parameter configured to `Inherit: Inherit via back propagation`.

The block has three inputs: Ref1 and Ref2 are the reference inputs, while the Prop input back propagates the data type and scaling information gathered from the reference inputs. This information is then passed on to other fixed-point blocks.

The block provides you with many choices for propagating data type and scaling information. For example, you can:

- Use the number of bits from the Ref1 reference signal, or use the number of bits from widest reference signal.
- Use the range from the Ref2 reference signal, or use the range of the reference signal with the greatest range.
- Use a bias of zero, regardless of the biases used by the reference signals.
- Use the precision of the reference signal with the least precision.

You specify how data type information is propagated with the **Propagated data type** parameter list. If the parameter list is configured as `Specify via dialog`, then

you manually specify the data type via the **Propagated data type** edit field. If the parameter list is configured as `Inherit via propagation rule`, then you must use the parameters described in "Parameters and Dialog Box" on page 1-324.

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 1-324.

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

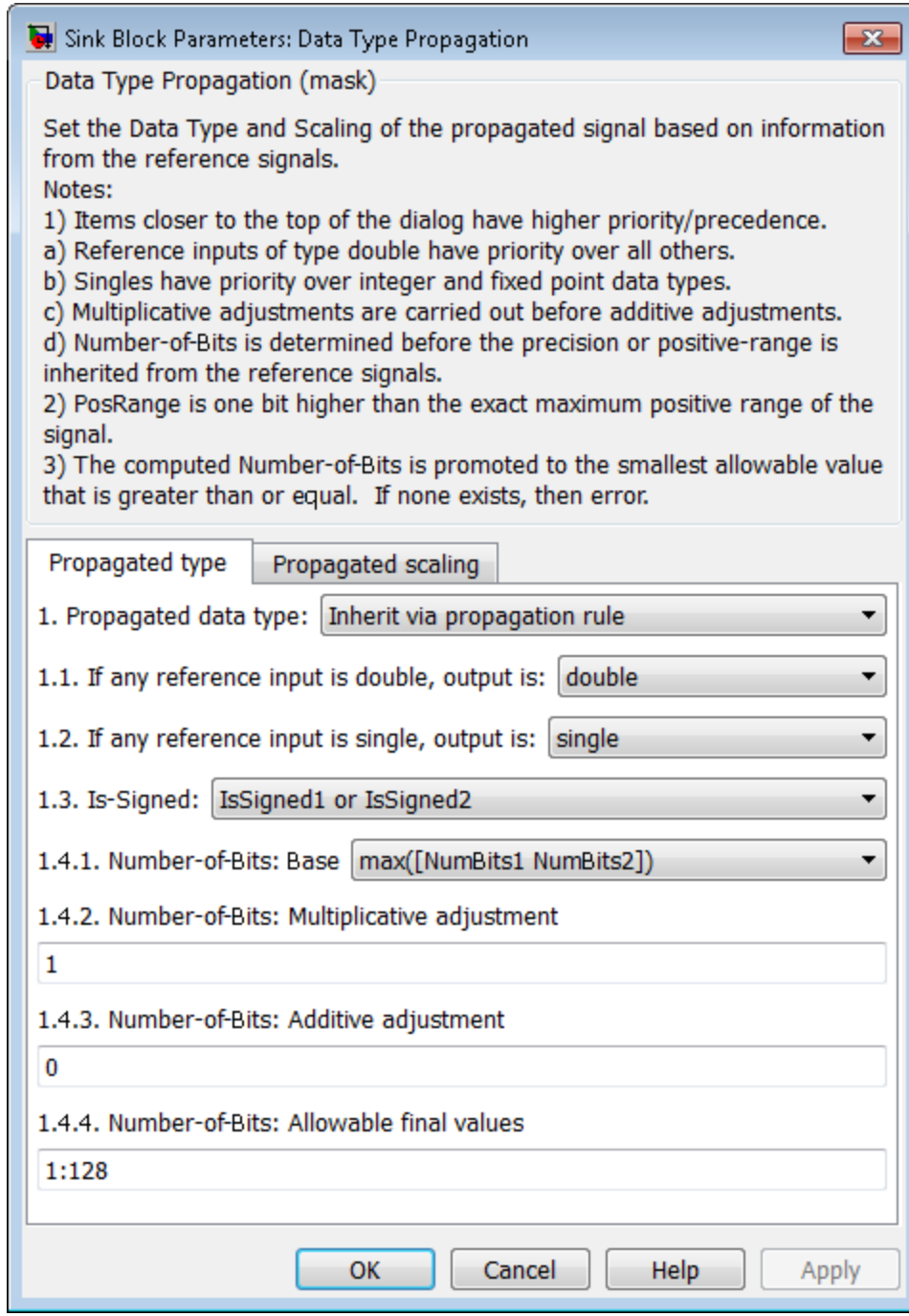
The Data Type Propagation block accepts signals of the following data types:

- 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:


Sink Block Parameters: Data Type Propagation
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.
Propagated type
Propagated scaling
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
OK
Cancel
Help
Apply

Propagated data type

Use the parameter list to propagate the data type via the dialog box, or inherit the data type from the reference signals. Use the edit field to specify the data type via the dialog box.

If any reference input is double, output is

Specify `single` or `double`. This parameter makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or vice versa.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

If any reference input is single, output is

Specify `single` or `double`. This parameter makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or visa versa.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Is-Signed

Specify the sign of Prop as one of the following values:

Parameter Value	Description
`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 that uses a data type of `double`. For example, suppose you want to guarantee that the number of bits associated with a multiply and accumulate (MAC) operation is twice as wide as the input signal. To do this, you configure this parameter to the value `2`.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Number-of-bits: Additive adjustment

Specify the number of bits used by Prop by including an additive adjustment that uses a data type of `double`. For example, if you are performing multiple additions during a MAC operation, the result might overflow. To prevent overflow, you can associate guard bits with the propagated data type. To associate four guard bits, you specify the value `4`.

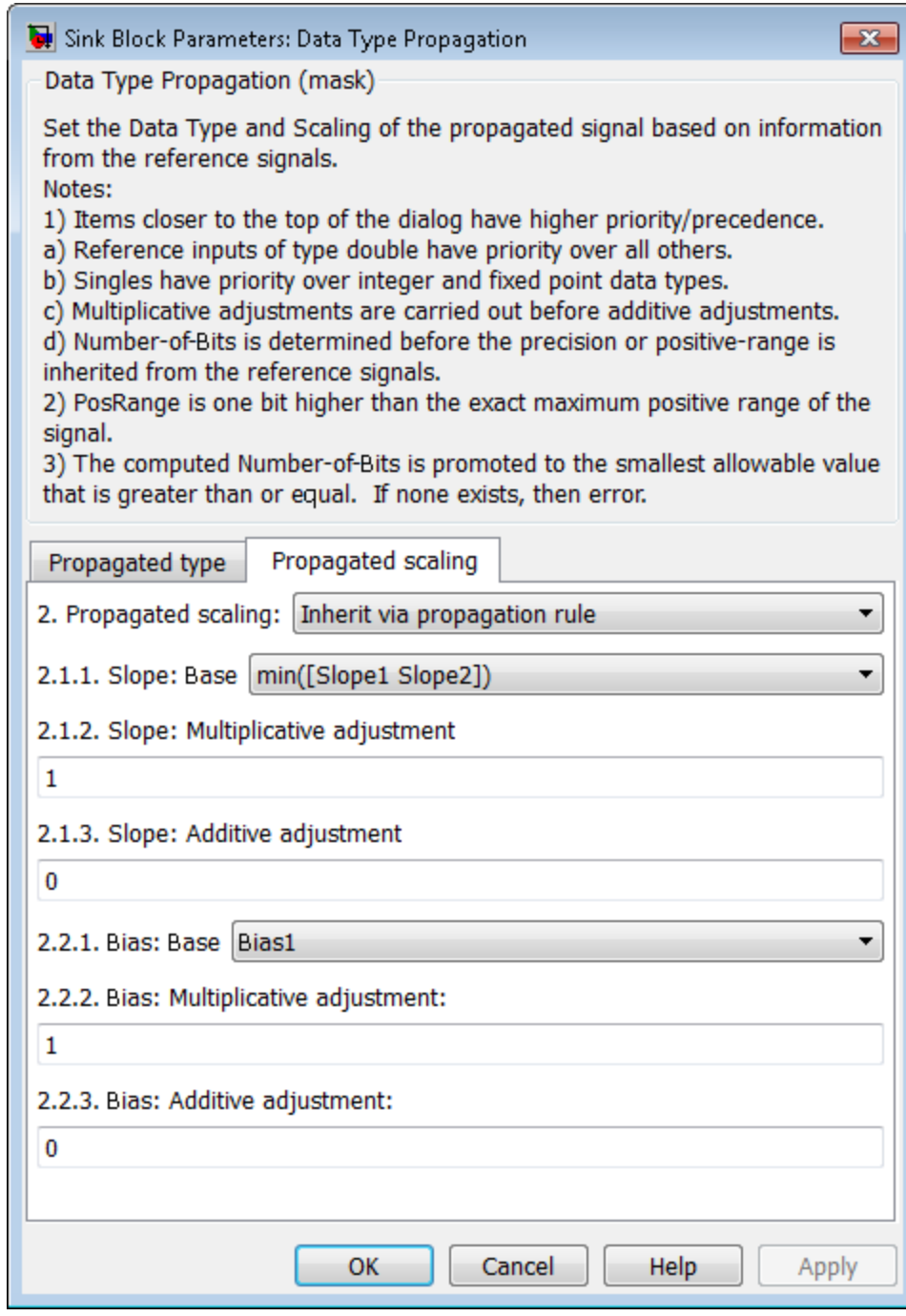
This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

Number-of-bits: Allowable final values

Force the computed number of bits used by Prop to a useful value. For example, if you are targeting a processor that supports only 8, 16, and 32 bits, then you configure this parameter to `[8,16,32]`. The block always propagates the smallest specified value that fits. If you want to allow all fixed-point data types, you would specify the value `1:128`.

This parameter is visible only when you set **Propagated data type** to `Inherit via propagation rule`.

The **Propagated scaling** pane of the Data Type Propagation block dialog box appears as follows:


Sink Block Parameters: Data Type Propagation
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.
Propagated type
Propagated scaling
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
OK
Cancel
Help
Apply

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.
`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.

Parameter Value	Description
`min([PosRange1 PosRange2])`	The range of Prop is given by the minimum range of the reference signals.
`PosRange1*PosRange2`	The range of Prop is given by the product of the reference signal ranges.
`PosRange1/PosRange2`	The range of Prop is given by the ratio of the Ref1 range to the Ref2 range.

You control the precision of Prop with `Slope1` and `Slope2`, and you control the range of Prop with `PosRange1` and `PosRange2`. Additionally, `PosRange1` and `PosRange2` are one bit higher than the maximum positive range of the associated reference signal.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Slope: Multiplicative adjustment

Specify the slope used by Prop by including a multiplicative adjustment that uses a data type of `double`. For example, if you want 3 bits of additional precision (with a corresponding decrease in range), the multiplicative adjustment is `2^-3`.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Slope: Additive adjustment

Specify the slope used by Prop by including an additive adjustment that uses a data type of `double`. An additive slope adjustment is often not needed. The most likely use is to set the multiplicative adjustment to `0`, and set the additive adjustment to force the final slope to a specified value.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Bias: Base

Specify the bias used by Prop for the base data type. The parameter values are described as follows:

Parameter Value	Description
`Bias1`	The bias of Prop is given by the bias of Ref1.
`Bias2`	The bias of Prop is given by the bias of Ref2.

Parameter Value	Description
`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 that uses a data type of `double`.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Bias: Additive adjustment

Specify the bias used by Prop by including an additive adjustment that uses a data type of `double`.

If you want to guarantee that the bias associated with Prop is zero, you should configure both the multiplicative adjustment and the additive adjustment to `0`.

This parameter is visible only when you set **Propagated scaling** to `Inherit via propagation rule`.

Characteristics

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

Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Ye
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Data Type Conversion` | `Data Type Conversion Inherited` | `Data Type Duplicate`

Related Examples

- “Control Signal Data Types”

More About

- “About Data Types in Simulink”
- “Fixed Point”

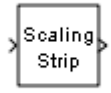
Introduced before R2006a

Data Type Scaling Strip

Remove scaling and map to built in integer

Library

Signal Attributes



Description

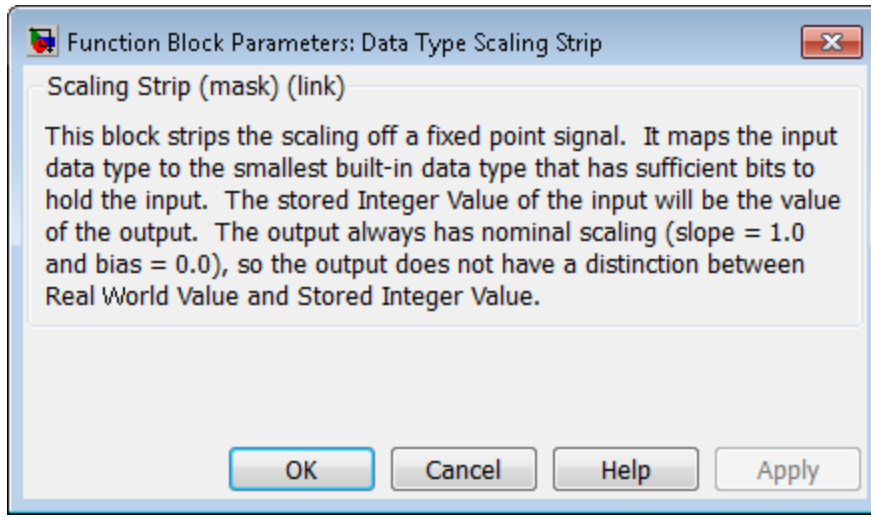
The Scaling Strip block strips the scaling off a fixed point signal. It maps the input data type to the smallest built in data type that has enough data bits to hold the input. The stored integer value of the input is the value of the output. The output always has nominal scaling (slope = 1.0 and bias = 0.0), so the output does not make a distinction between real world value and stored integer value.

Data Type Support

The Data Type Scaling Strip block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

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

Parameters and Dialog Box



Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

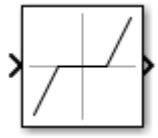
Dead Zone

Provide region of zero output

Library

Discontinuities

Description



The Dead Zone block generates zero output within a specified region, called its dead zone. You specify the lower limit (`LL`) and upper limit (`UL`) of the dead zone as the **Start of dead zone** and **End of dead zone** parameters, respectively. The block output depends on the input (`U`) and the values for the lower and upper limits:

Input	Output
`U >= LL` and `U <= UL`	Zero
`U > UL`	`U – UL`
`U < LL`	`U – LL`

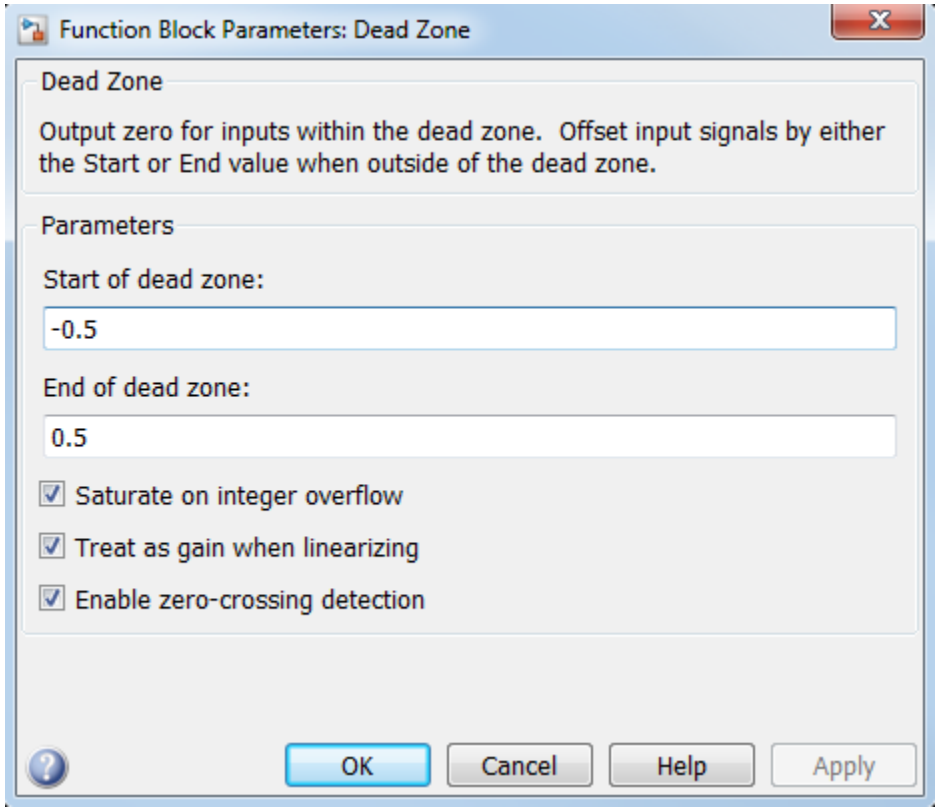
Data Type Support

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.

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	Overflows saturate to either the minimum or	The maximum value that the `int8` (signed, 8-bit integer) data type

Action	Reasons for Taking This Action	What Happens for Overflows	Example
	protection in the generated code.	maximum value that the data type can represent.	can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

Treat as gain when linearizing

The linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the commands to treat the gain as 1; otherwise, the commands treat the gain as 0.

Enable zero-crossing detection

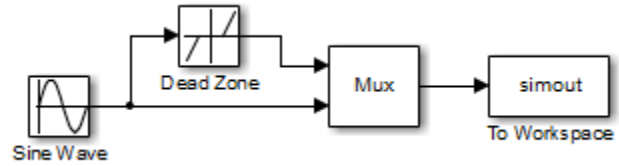
Select to enable zero-crossing detection. For more information, see "Zero-Crossing Detection" in the Simulink documentation.

Sample time

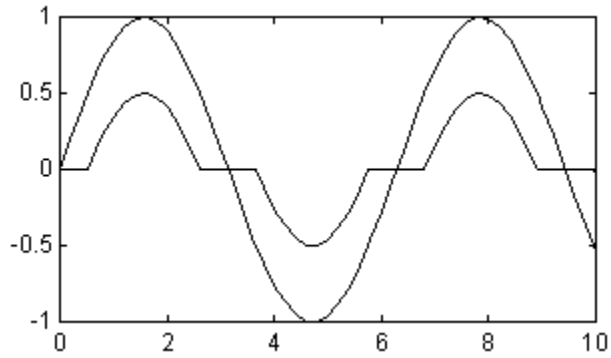
Note: This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see "Blocks for Which Sample Time Is Not Recommended".

Examples

The following model uses lower and upper limits of -0.5 and 0.5, with a sine wave as input.



This plot shows the effect of the Dead Zone block on the sine wave. When the input sine wave is between –0.5 and 0.5, the output is zero.



Characteristics

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

See Also

`Dead Zone Dynamic`

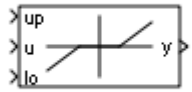
Introduced before R2006a

Dead Zone Dynamic

Set inputs within bounds to zero

Library

Discontinuities



Description

The Dead Zone Dynamic block dynamically bounds the range of the input signal, providing a region of zero output. The bounds change according to the upper and lower limit input signals where

- The input within the bounds is set to zero.
- The input below the lower limit is shifted down by the lower limit.
- The input above the upper limit is shifted down by the upper limit.

The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

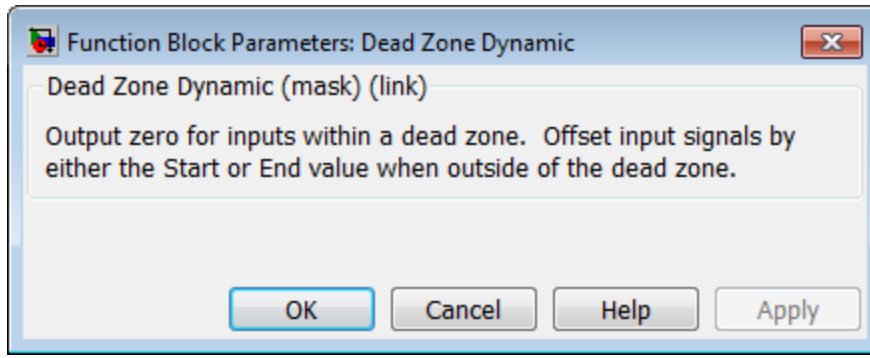
Data Type Support

The Dead Zone Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Characteristics

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

See Also

`Dead Zone`

Introduced before R2006a

Decrement Real World

Decrease real world value of signal by one

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Real World block decreases the real world value of the signal by one. Overflows always wrap.

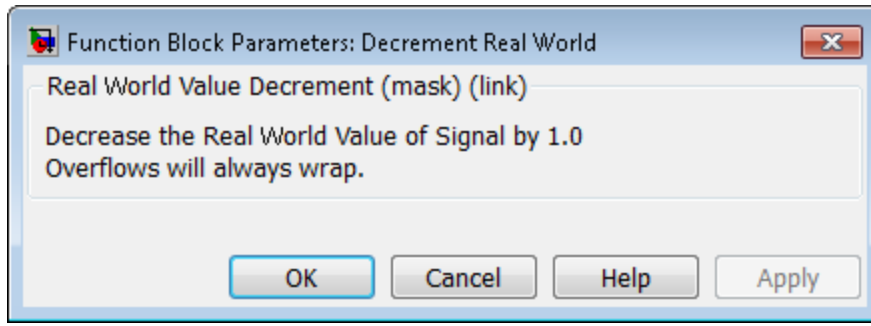
Data Type Support

The Decrement Real World block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Decrement Stored Integer`, `Decrement Time To Zero`, `Decrement To Zero`, `Increment Real World`

Introduced before R2006a

Decrement Stored Integer

Decrease stored integer value of signal by one

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Stored Integer block decreases the stored integer value of a signal by one.

Floating-point signals also decrease by one, and overflows always wrap.

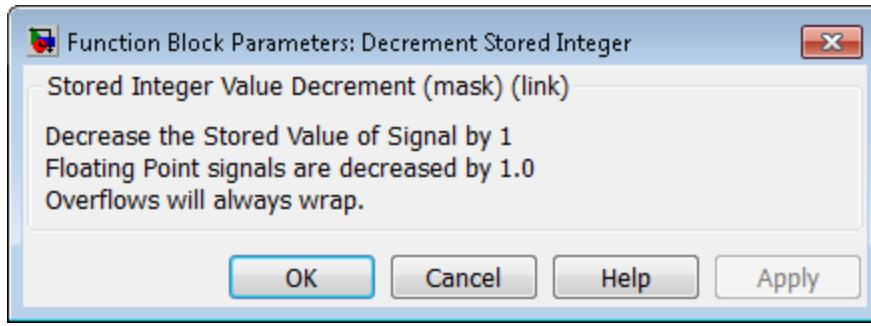
Data Type Support

The Decrement Stored Integer block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Decrement Real World`, `Decrement Time To Zero`, `Decrement To Zero`, `Increment Stored Integer`

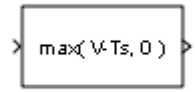
Introduced before R2006a

Decrement Time To Zero

Decrease real-world value of signal by sample time, but only to zero

Library

Additional Math & Discrete / Additional Math: Increment - Decrement



Description

The Decrement Time To Zero block decreases the real-world value of the signal by the sample time, `Ts`. The output never goes below zero. This block works only with fixed sample rates and does not work inside a triggered subsystem.

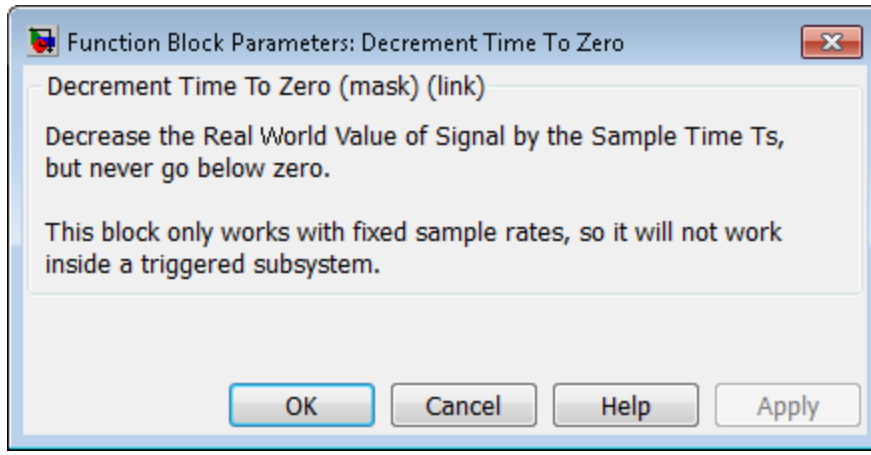
Data Type Support

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.

Parameters and Dialog Box



Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Decrement Real World`, `Decrement Stored Integer`, `Decrement To Zero`

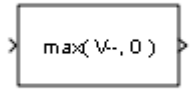
Introduced before R2006a

Decrement To Zero

Decrease real-world value of signal by one, but only to zero

Library

Additional Math & Discrete / Additional Math: Increment - Decrement



Description

The Decrement To Zero block decreases the real-world value of the signal by one. The output never goes below zero.

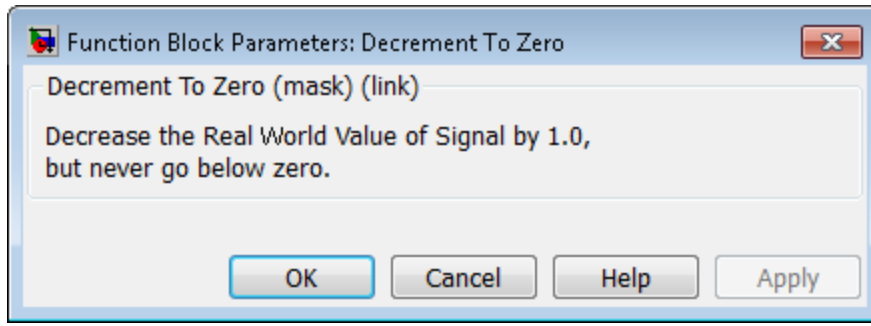
Data Type Support

The Decrement To Zero block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Decrement Real World`, `Decrement Stored Integer`, `Decrement Time To Zero`

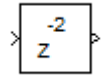
Introduced before R2006a

Delay

Delay input signal by fixed or variable sample periods

Library

Discrete

Description 

The Delay block outputs the input of the block after a delay. The block determines the delay time based on the value of the **Delay length** parameter. The block supports:

- Variable delay length
- Specification of the initial condition from an input port
- State storage
- Using a circular buffer instead of an array buffer for state storage
- Resetting the state to the initial condition with an external reset signal
- Controlling execution of the block at every time step with an external enable signal

The initial block output depends on a number of factors such as the **Initial condition** parameter and the simulation start time. For more information, see “Initial Block Output” on page 1-351. The **External reset** parameter determines if the block output resets to the initial condition on triggering. The **Show enable port** parameter determines if the block execution is controlled in every time step by an external enable signal.

Initial Block Output

The output of the Delay block in the first few time steps of the simulation depends on the block sample time, the delay length, and the simulation start time. The block supports specifying or inheriting discrete sample times to determine the time interval between samples. For more information, see “Specify Sample Time”.

Suppose that the block inherits a discrete sample time as `[Tsampling,Toffset]`, where `Tsampling` is the sampling period and `Toffset` is the initial time offset. `n` is the value of the block's **Delay length** parameter and `Tstart` is the simulation start time for the model.

The table shows the Delay block output for the first few time steps.

Simulation Time Range	Block Output
(`Tstart`) to (`Tstart + Toffset`)	Zero
(`Tstart + Toffset`) to (`Tstart + Toffset + n * Tsampling`)	**Initial condition** parameter
After (`Tstart + Toffset + n * Tsampling`)	Input signal

Data Type Support

The block's parameters have these dimensional requirements:

- **Delay length** and **External reset** must be scalar.
- **Initial condition** can be scalar or nonscalar.
- For frame-based processing, signal dimensions of the data input port `u` cannot be larger than two.

The block supports input signals with these data types.

Input Signal	Supported Data Types
Data input port `u`	• Floating point • Built-in integer • Fixed point • Boolean • Enumerated
Delay length `d`	• Floating point • Fixed-point integer • Built-in integer
Enable port 	• Floating point • Built-in integer

Input Signal	Supported Data Types
	• Fixed point Integer (only `ufix1`) • Boolean
External reset port 	• Floating point • Built-in integer • Fixed point Integer (only `ufix1`) • Boolean
Initial condition `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`.

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

Variable-Size Support

The Delay block provides the following support for variable-size signals:

- The data input port `u` accepts variable-size signals. The other input ports do not accept variable-size signals.
- The output port has the same signal dimensions as the data input port `u` for variable-size inputs.

The rules that apply to variable-size signals depend on the input processing mode of the Delay block.

Input Processing Mode	Rules for Variable-Size Signal Support
`Elements as channels (sample based)`	• The signal dimensions change only during state reset when the block is enabled. • The initial condition must be scalar.
`Columns as channels (frame based)`	• No support
`Inherited` (where input is a sample-based signal)	• The signal dimensions change only during state reset when the block is enabled. • The initial condition must be scalar.
`Inherited` (where input is a frame-based signal)	• The channel size changes only during state reset when the block is enabled. • The initial condition must be scalar. • The frame size must be constant.

Bus Support

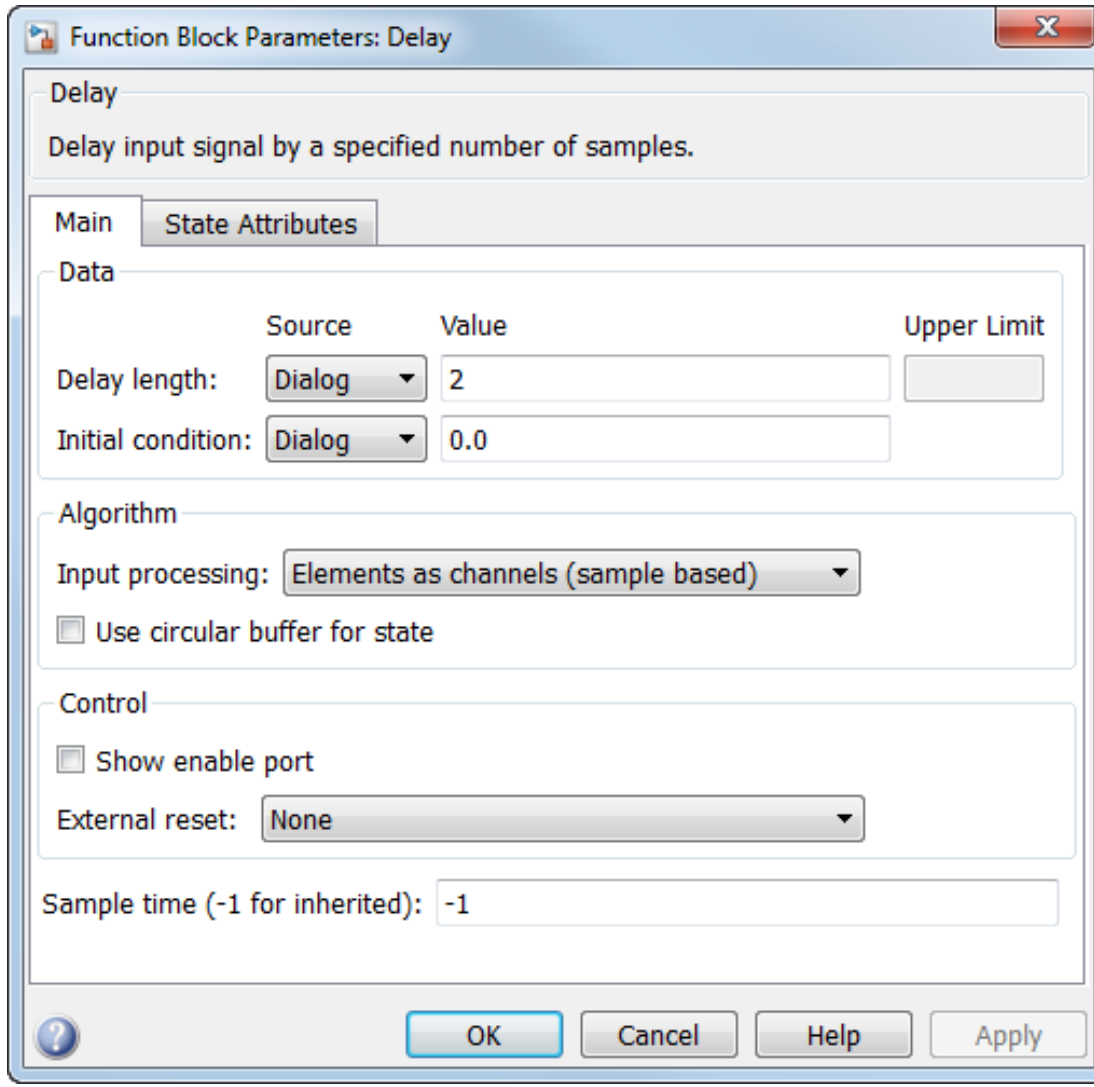
The Delay block provides the following support for bus signals:

- The data input port `u` accepts virtual and nonvirtual bus signals. The other input ports do not accept bus signals.
- The output port has the same bus type as the data input port `u` for bus inputs.
- Buses work with:
  - Sample-based and frame-based processing
  - Fixed and variable delay length
  - Array and circular buffers

To use a bus signal as the input to a Delay block, you should 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

Parameters and Dialog Box



Delay length

Specify whether to enter the delay length directly on the dialog box (fixed delay) or to inherit the delay from an input port (variable delay).

- If you set **Source** to `Dialog`, enter the delay length in the edit field under **Value**.
- If you set **Source** to `Input port`, verify that an upstream signal supplies a delay length for the `d` input port. You can also specify its maximum value by specifying the parameter **Upper limit**.

Specify the scalar delay length as a real, non-negative integer. An out-of-range or non-integer value in the dialog box (fixed delay) returns an error. An out-of-range value from an input port (variable delay) casts it into the range. A non-integer value from an input port (variable delay) truncates it to the integer.

This parameter is not tunable for simulation or code generation.

Initial condition

Specify whether to enter the initial condition directly on the dialog box or to inherit the initial condition from an input port.

- If you set **Source** to `Dialog`, enter the initial condition in the edit field under **Value**.
- If you set **Source** to `Input port`, verify that an upstream signal supplies an initial condition for the `x0` input port.

Simulink converts offline the data type of **Initial condition** to the data type of the input signal `u` using a round-to-nearest operation and saturation.

Note: When **State name must resolve to Simulink signal object** is selected on the **State Attributes** pane, the block copies the initial value of the signal object to the **Initial condition** parameter. However, when the source for **Initial condition** is `Input port`, the block ignores the initial value of the signal object.

Input processing

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

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

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

For more information, see "Sample- and Frame-Based Concepts" 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 Diagnostic Viewer.

This parameter is available when you set **Delay length: Source** to `Input port`.

Show enable port

Select to show an enable port for this block. This port can control execution of the block. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
`None`	No reset.
`Rising`	Reset on a rising edge.
`Falling`	Reset on a falling edge.
`Either`	Reset on either a rising or falling edge.
`Level`	Reset in either of these cases: • when the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
`Level hold`	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. This block supports discrete sample time, but not continuous sample time.

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see "Discrete Block State Naming in Generated Code" 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**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

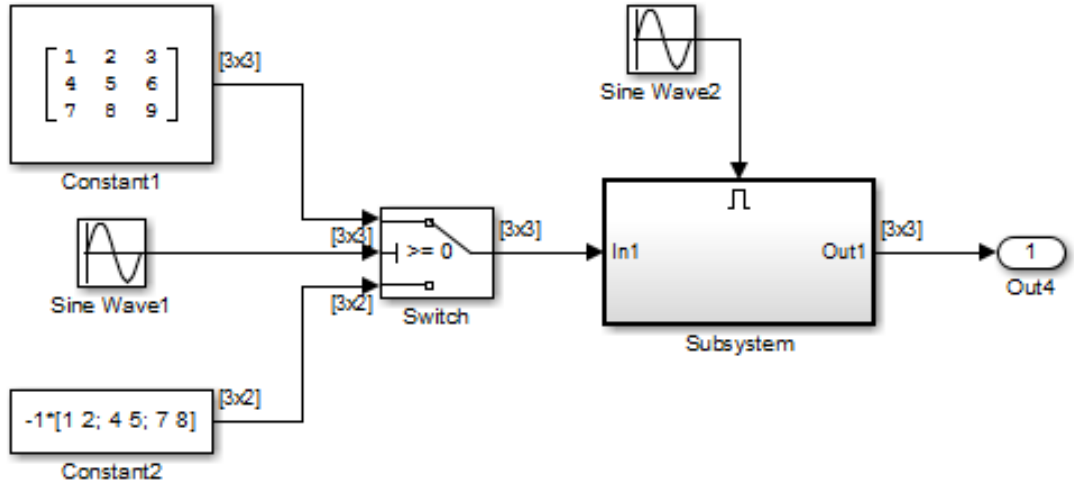
Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

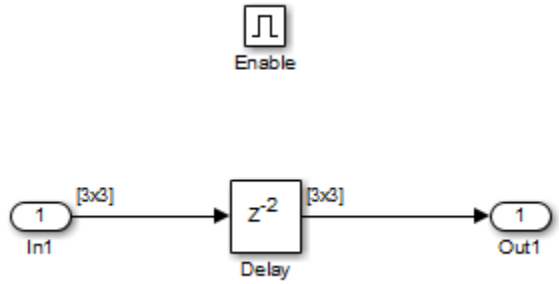
Examples

Variable-Size Signals for Sample-Based Processing

This model shows how the Delay block supports variable-size signals for sample-based processing.



The Switch block controls whether the input signal to the enabled subsystem is a 3-by-3 or 3-by-2 matrix. The Delay block appears inside the enabled subsystem.



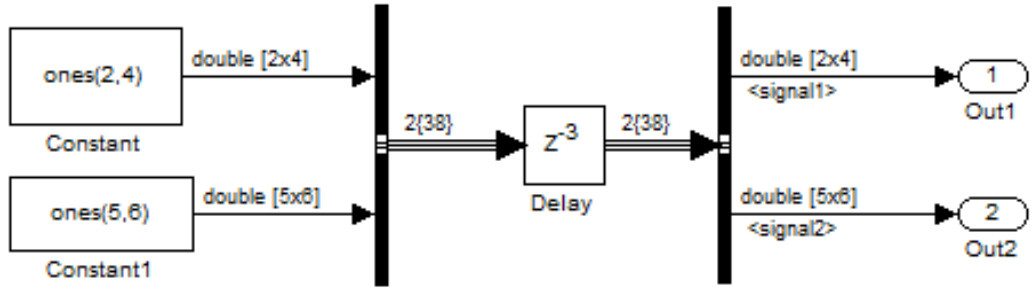
The model follows the rules for variable-size signals when the Delay block uses sample-based processing.

Rule	How the Model Follows the Rule
The signal dimensions change only during state reset when the block is enabled.	The Enable block sets **Propagate sizes of variable-size signals** to `Only when enabling`.

Rule	How the Model Follows the Rule
The initial condition must be scalar.	The Delay block sets **Initial condition** to `0.0`, a scalar value.

Bus Signals for Frame-Based Processing

This model shows how the Delay block supports bus signals for frame-based processing.



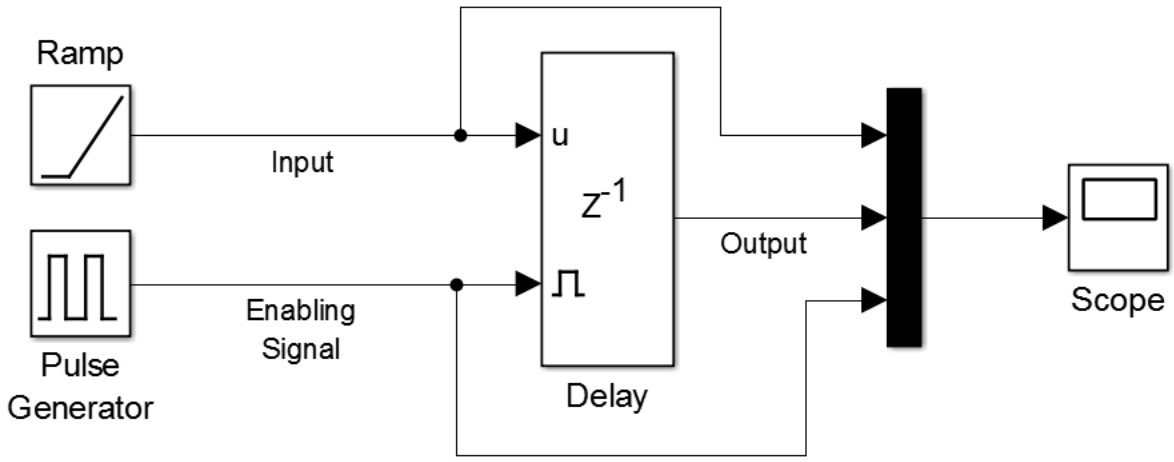
Each Constant block supplies an input signal to the Bus Creator block, which outputs a two-dimensional bus signal. After the Delay block delays the bus signal by three sample periods, the Bus Selector block separates the bus back into the two original signals.

The model follows the rules for bus signals when the Delay block uses frame-based processing.

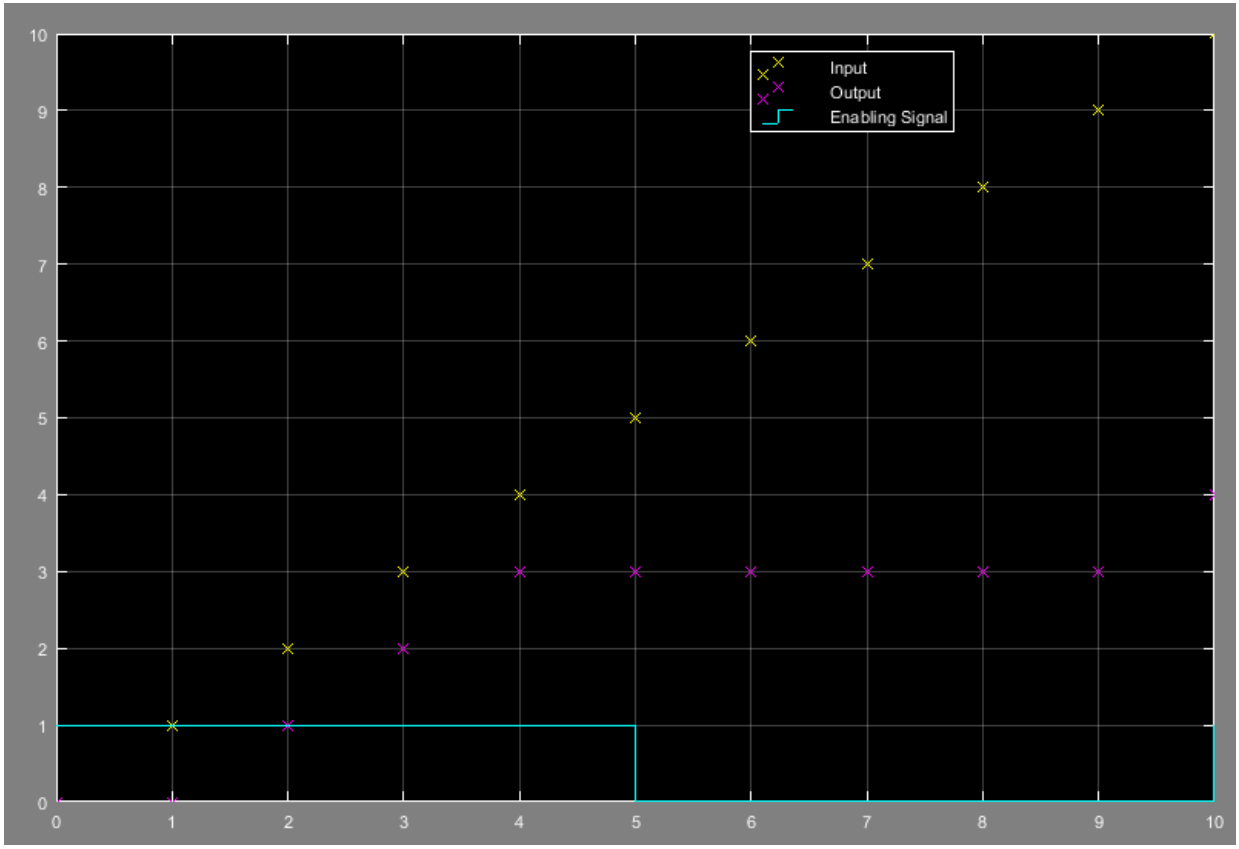
Rule	How the Model Follows the Rule
For the initial condition, set the value on the dialog box.	The Delay block sets **Initial condition** to `0`, a scalar value.
For frame-based processing, signal dimensions of the data input port `u` cannot be larger than two. (This rule applies to all inputs for the port `u`, not just bus signals.)	The bus input to the Delay block has two dimensions.

Enable or Disable Execution of the Delay Block

This example shows how you can enable or disable the execution of the Delay block using the enable port of the block. Consider this model. A ramp input signal feeds into a Delay block whose execution is controlled by an enabling signal. A Pulse Generator block generates this enabling signal.



The Scope block displays the output of the Delay block along with the enabling signal and the ramp input. Simulating the model and viewing the scope output shows the following graph.



The magenta marks show that the Delay block outputs the input signal delayed by one time step only while the enabling signal is 1. At t=5 sec, the enabling signal becomes 0 and the Delay block does not execute. Hence, the output is held constant until the next time the enabling signal becomes 1.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter

Direct Feedthrough	Yes, when you clear **Prevent direct feedthrough by increasing delay length to lower limit**
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Resettable Delay` | `Tapped Delay` | `Unit Delay` | `Variable Integer Delay`

Demux

Extract and output elements of 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 "Prevent Bus and Mux 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.

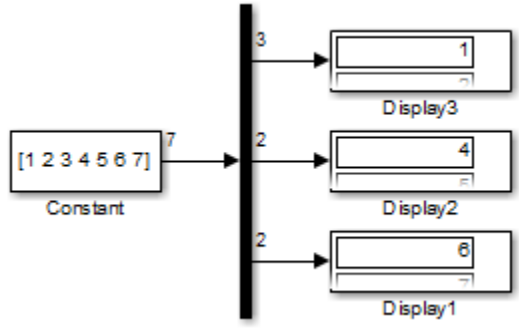
`p < n`
 	
`n mod p = m`
	`m` vector signals each having `(n/p)+1` elements and `p-m` signals having `n/p` elements	If the input is a five-element vector and you specify three outputs, the block outputs two two-element vector signals and one scalar signal.

$[p_1\ p_2\ \ldots\ p_m]$ $p_1+p_2+\ldots+p_m=n$ $p_i > 0$	`m` vector signals having widths p_1, p_2, ... p_m	If the input is a five-element vector and you specify `[3, 2]` as the output, the block outputs three of the input elements on one port and the other two elements on the other port.
An array that has one or more of `m` elements with a value of `-1`, which specifies that	`m` vector signals	If p_i is greater than zero, the corresponding output has width p_i. If p_i is -1, the width

Parameter Value	Block outputs...	Comments
Simulink infers the size for the element. For example, suppose you have a four-element array with a total width of 14 and you specify the parameter to be `[p1 p2 -1 p4]` The value for the third element (the `-1` element) is `14 - (p1 + p2 + p)4`		of the corresponding output is computed dynamically.

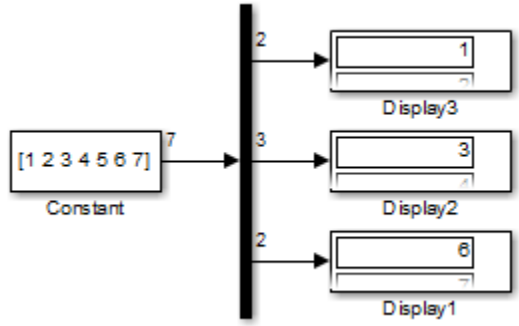
`[p1 p2 ... pm]`
 	
`p1+p2+...+pm!=n`
 	
`pi = > 0`
	Error	

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.

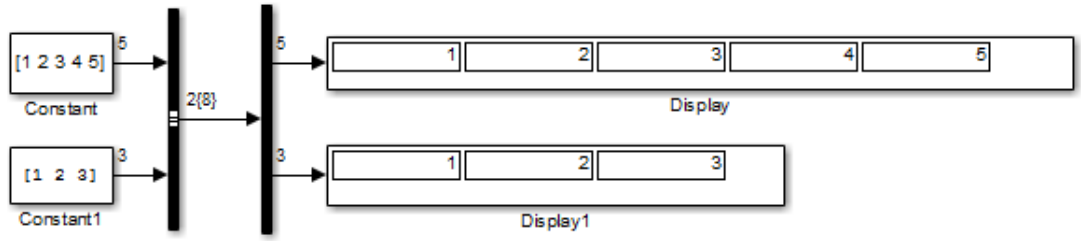


Specifying the Number of Outputs in Bus Selection Mode

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`.

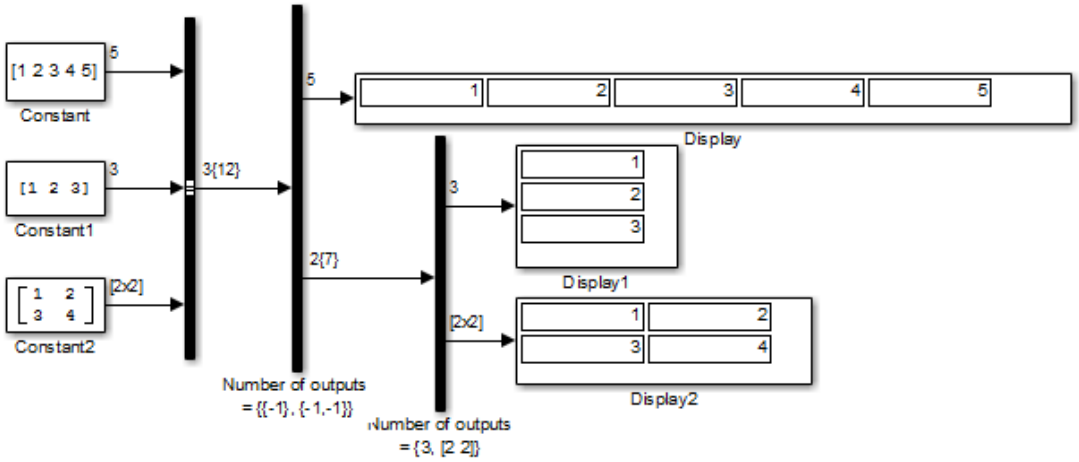


- Vector each of whose elements specifies the number of signals to output on the corresponding port

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.



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.

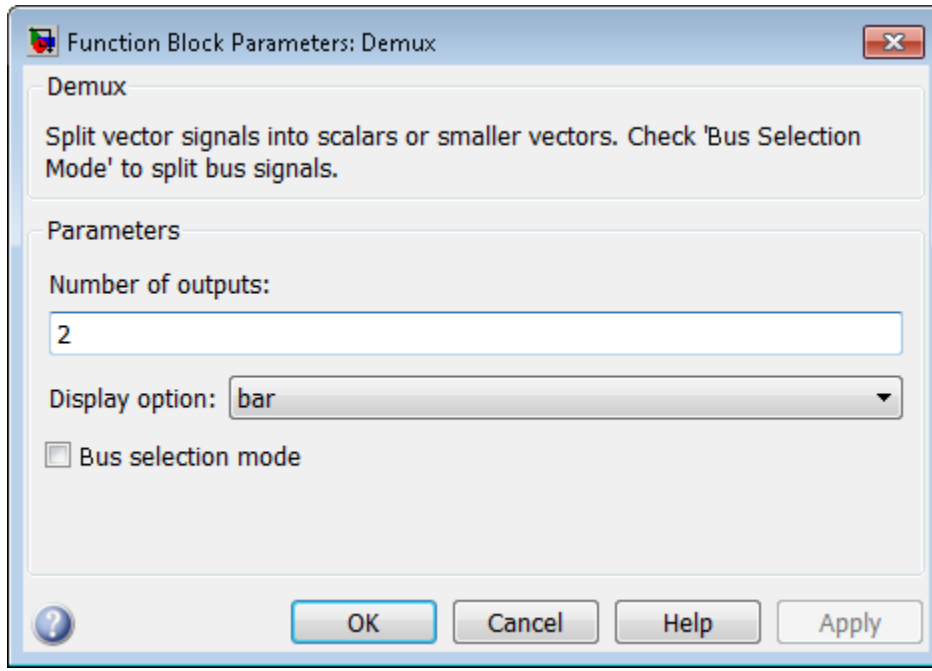
Note: MathWorks discourages enabling **Bus selection mode** and using a Demux block to extract elements of a bus signal. Muxes and buses should not be intermixed in new models. See “Prevent Bus and Mux Mixtures” for more information.

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.

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 6-101 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 foreground color.

`none`

Display the icon as a box containing the block type name.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 “Prevent Bus and Mux Mixtures” for more information.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

`Mux`

Introduced before R2006a

Derivative

Output time derivative of input

Library

Continuous

Description

The `Derivative` block approximates the derivative of the input signal u with respect to the simulation time t. You obtain the approximation of

$$\frac{du}{dt},$$

by computing a numerical difference $\Delta u/\Delta t$, where Δu is the change in input value and Δt is the change in time since the previous simulation (major) time step.

This block accepts one input and generates one output. The initial output for the block is zero.

The precise relationship between the input and output of this block is:

$$y(t) = \frac{\Delta u}{\Delta t} = \left.\frac{u(t) - u(T_{previous})}{t - T_{previous}}\right| t > T_{previous},$$

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` block to approximate the discrete-time derivative of a discrete signal.

Improved Linearization with Transfer Fcn Blocks

The Laplace domain transfer function for the operation of differentiation is:

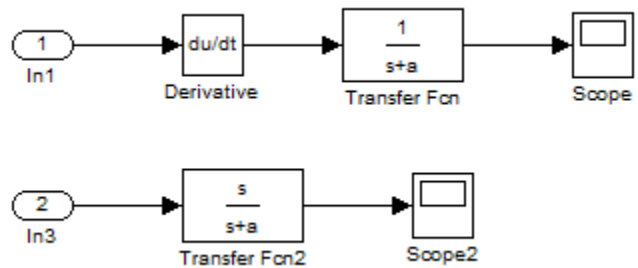
$$Y(s)/X(s) = s$$

This equation is not a proper transfer function, nor does it have a state-space representation. As such, the Simulink software linearizes this block as an effective gain of 0 unless you explicitly specify that a proper first-order transfer function should be used to approximate the linear behavior of this block (see “Coefficient c in the transfer function approximation s/(c*s + 1) used for linearization” on page 1-381).

To improve linearization, you can also try to incorporate the derivative term in other blocks. For example, if you have a `Derivative` block in series with a `Transfer Fcn` block, try using a single Transfer Fcn block of the form

$$\frac{s}{s+a}.$$

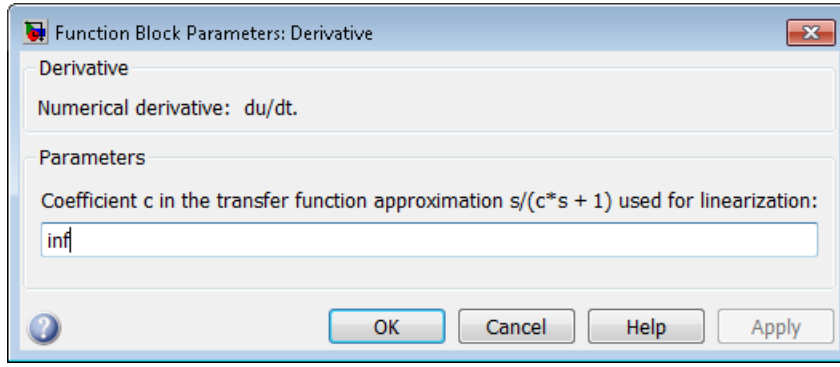
For example, you can replace the first set of blocks in this figure with the blocks below them.



Data Type Support

The Derivative block accepts and outputs a real signal of type `double`. For more information, see “Data Types Supported by Simulink”.

Parameters 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

Parameter: `CoefficientInTFapproximation`
Type: string
Value: `'inf'`
Default: `'inf'`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Yes
Multidimensional Signals	No

Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Discrete Derivative`

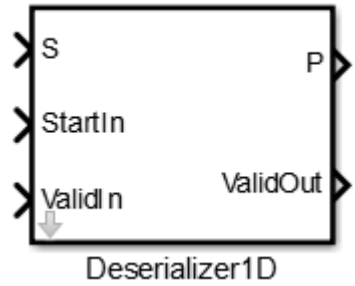
Introduced before R2006a

Deserializer1D

Convert scalar stream or smaller vectors to vector signal

Library

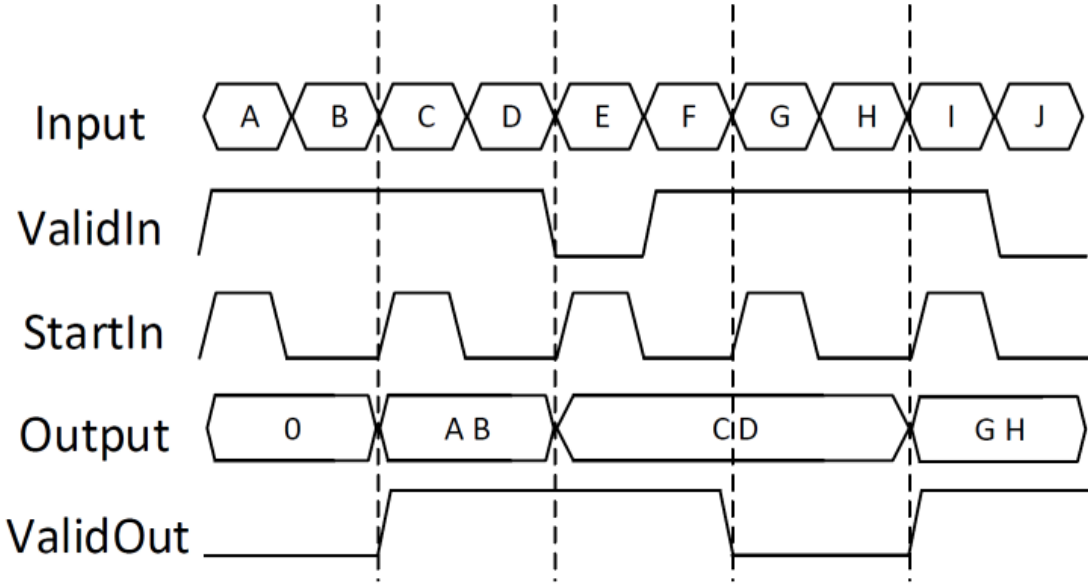
HDL Coder / HDL Operations



Description

The `Deserializer1D` block buffers a faster, scalar stream or vector signals into a larger, slower vector signal. The faster input signal is converted to a slower signal based on the **Ratio** and **Idle Cycle** values, the conversion changes sample time. Also, the output signal is delayed one slow signal cycle because the serialized data needs to be collected before it can be output as a vector. See the examples below for more details.

You can configure the deserialization to depend on a valid input signal ValidIn and a start signal StartIn. If the **ValidIn** and **StartIn** block parameters are both selected, data collection starts only if both ValidIn and StartIn signals are true. Consider this example:

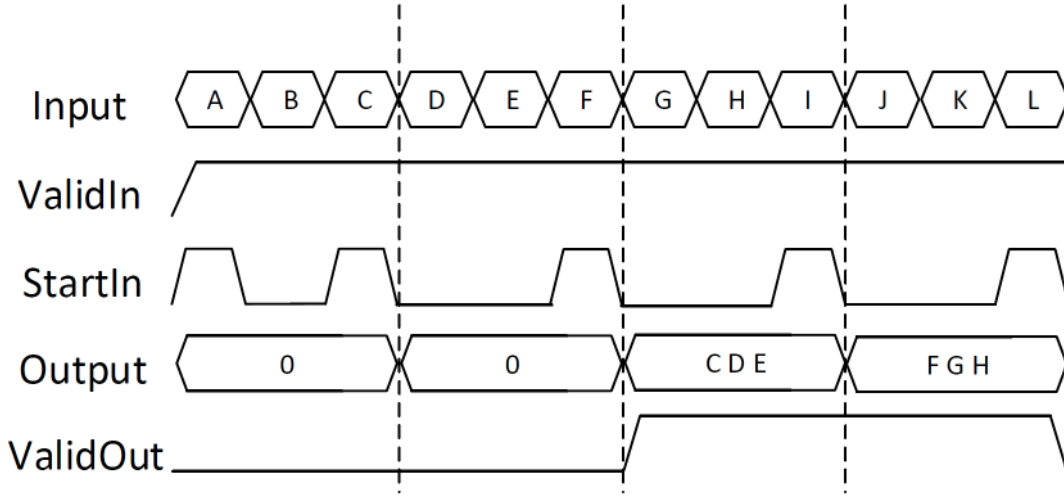


- **Ratio** is `2` and **Idle Cycles** is `0`, so each output cycle is two input signals long with all data points considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection begins for A and B. The block outputs the deserialized vector in the next valid cycle, so the AB vector is output in the next cycle. This is also true in the second cycle for C and D.

In the third cycle, starting at E, StartIn is true, but ValidIn is not. E is dropped. At F, ValidIn is true, but StartIn is not, so F is also dropped. Since it cannot collect data for E or F, `Deserializer1D` outputs the previous cycle vector, CD, but ValidOut changes to false.

Another scenario to consider is when the StartIn signal arrives too early. If the length between two StartIn signals is not long enough to collect a full ratio cycle, the insufficient signal data is dropped. Consider this example:

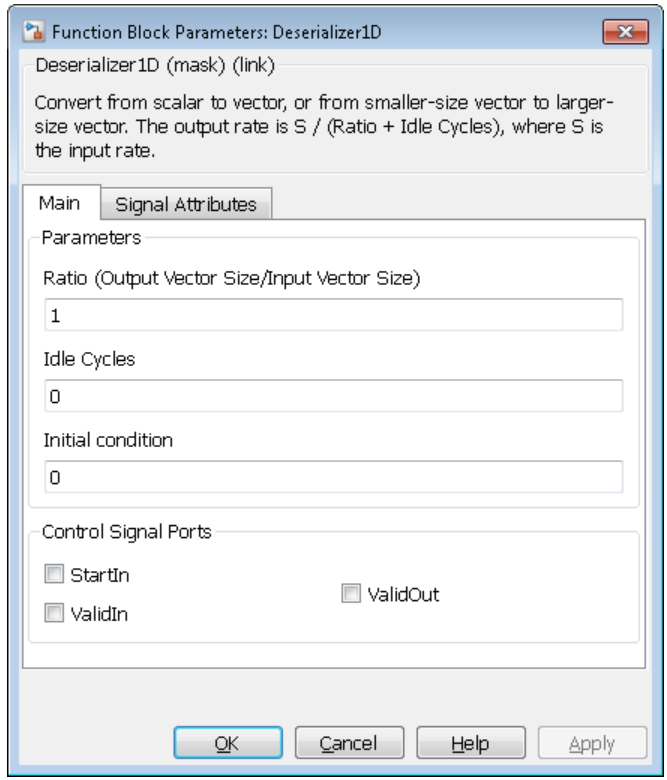


- **Ratio** is 3, so each cycle is two sections long.
- **Idle Cycles** is 0, so all data inputs are considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection can begin for A and B. However, at C another StartIn signal arrives before three signals can be collected. Because the StartIn arrived early, A and B are dropped and no valid vector is collected during the first cycle. Therefore, the output of the second cycle is still zero. Deserialization begins at the StartIn at C, for C, D, and E. This vector is output at the next valid cycle, which is cycle 3. Similarly, deserialization starts again at the StartIn at F, and outputs the FGH vector in the fourth cycle.

You specify the block output for the first sampling period with the value of the **Initial condition** parameter.

Dialog Box and Parameters



Ratio

Enter the deserialization ratio. Default is `1`.

The ratio is the output vector size, divided by the input vector size. The ratio must be divisible by the input vector size.

Idle Cycles

Enter the number of idle cycles added to the end of each serialized input. Default is `0`.

The value of **Idle Cycles** affects the deserialized output rate. For example, if **Ratio** is `2` and the input signal is `A, B, B, C, D, D, ...`, without idle cycles the output

would be `AB, BC, DD...`. However for the same input and ratio with **Idle Cycles** set to `1`, the output is `AB, CD...`. The idle cycles, `B` and `D`, are dropped.

The `Deserializer1D` behavior changes if **Idle Cycles** is not zero, and **ValidIn** or **StartIn** are on. The idle cycles value affects only the output rate, while **ValidIn** and **StartIn** control what input data is deserialized.

Initial condition

Specify the initial output of the simulation. Default is 0.

StartIn

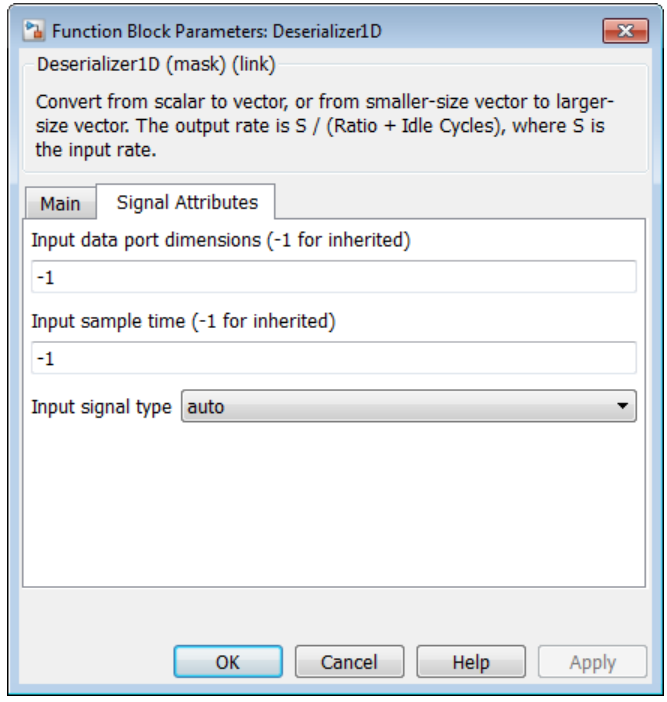
Select to activate the StartIn port. Default is off.

ValidIn

Select to activate the ValidIn port. Default is off.

ValidOut

Select to activate ValidOut port. Default is off.



Input data port dimensions (-1 for inherited)

Enter the size of the input data signal. The input size must be divisible by the ratio plus the number of idle cycles. By default, the block inherits size based on context within the model.

Input sample time (-1 for inherited)

Enter the time interval between sample time hits or specify another appropriate sample time such as continuous. By default, the block inherits its sample time based on context within the model. For more information, see "Sample Time".

Input signal type

Specify the input signal type of the block as `auto`, `real`, or `complex`.

Ports

`S`

Input signal to deserialize. Bus data types are not supported.

`ValidIn`

Indicates valid input signal. Use with the `Serializer1D` block. This port is available when you select the **ValidIn** check box.

Data type: Boolean

`StartOut`

Indicates where to start deserialization. Use with the `Serializer1D` block. This port is available when you select the **StartOut** check box.

Data type: Boolean

`P`

Deserialized output signal. Bus data types are not supported.

`ValidOut`

Indicates valid output signal. This port is available when you select the **ValidOut** check box.

Data type: Boolean

See Also

`Serializer1D`

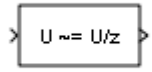
Introduced in R2014b

Detect Change

Detect change in signal value

Library

Logic and Bit Operations



Description

The Detect Change block determines if an input does not equal its previous value.

- The output is true (equal to 1) when the input signal does not equal its previous value.
- The output is false (equal to 0) when the input signal equals its previous value.

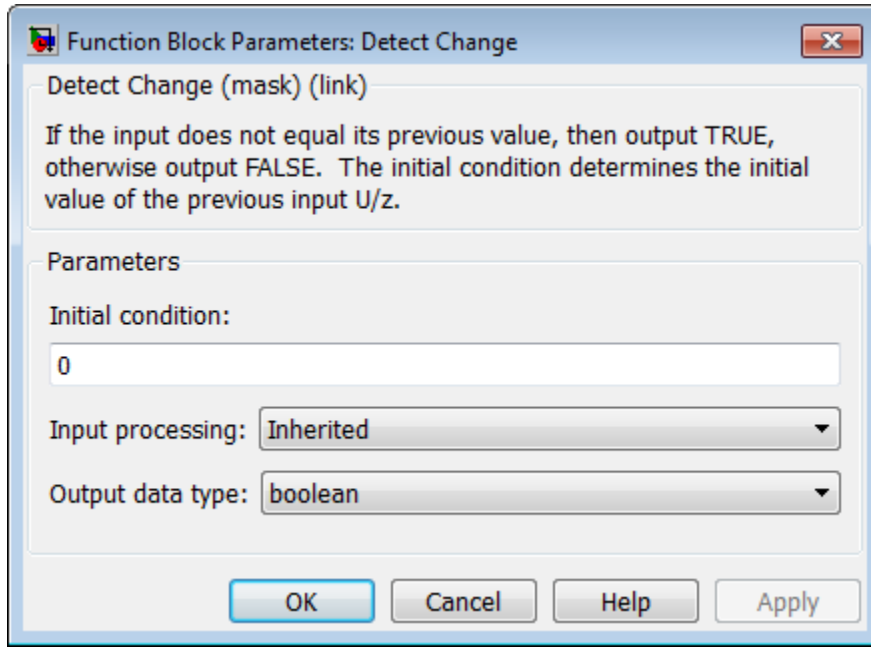
Data Type Support

The Detect Change block accepts signals of the following data types:

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

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

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes

Code Generation	Yes

See Also

`Detect Decrease`, `Detect Fall Negative`, `Detect Fall Nonpositive`, `Detect Increase`, `Detect Rise Nonnegative`, `Detect Rise Positive`

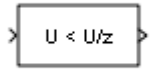
Introduced before R2006a

Detect Decrease

Detect decrease in signal value

Library

Logic and Bit Operations



Description

The Detect Decrease block determines if an input is strictly less than its previous value.

- The output is true (equal to 1) when the input signal is less than its previous value.
- The output is false (equal to 0) when the input signal is greater than or equal to its previous value.

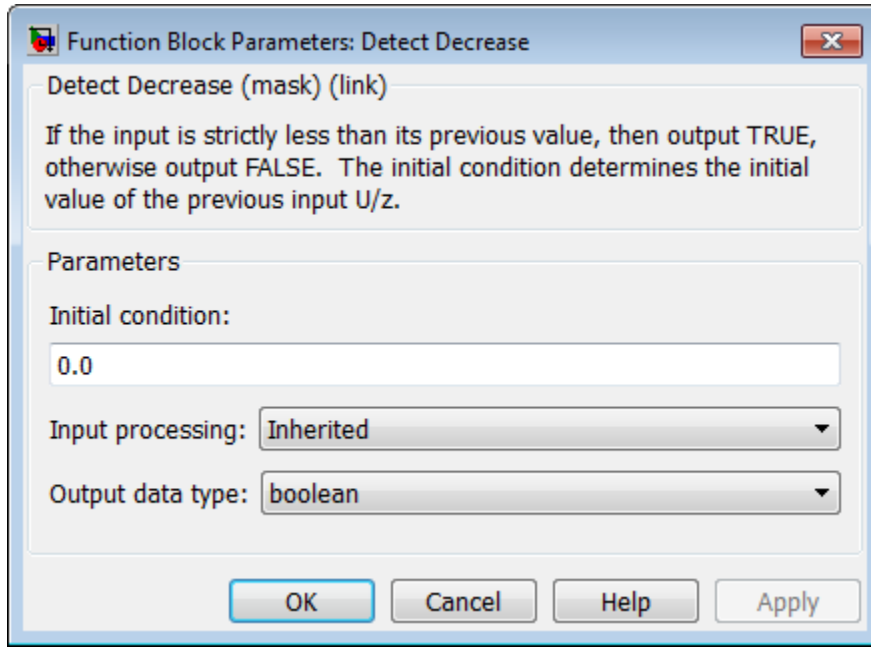
Data Type Support

The Detect Decrease block accepts signals of the following data types:

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

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

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes

Code Generation	Yes

See Also

`Detect Change`, `Detect Fall Negative`, `Detect Fall Nonpositive`, `Detect Increase`, `Detect Rise Nonnegative`, `Detect Rise Positive`

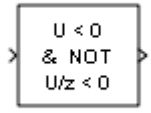
Introduced before R2006a

Detect Fall Negative

Detect falling edge when signal value decreases to strictly negative value, and its previous value was nonnegative

Library

Logic and Bit Operations



Description

The Detect Fall Negative block determines if the input is less than zero, and its previous value was greater than or equal to zero.

- The output is true (equal to 1) when the input signal is less than zero, and its previous value was greater than or equal to zero.
- The output is false (equal to 0) when the input signal is greater than or equal to zero, or if the input signal is negative, its previous value was also negative.

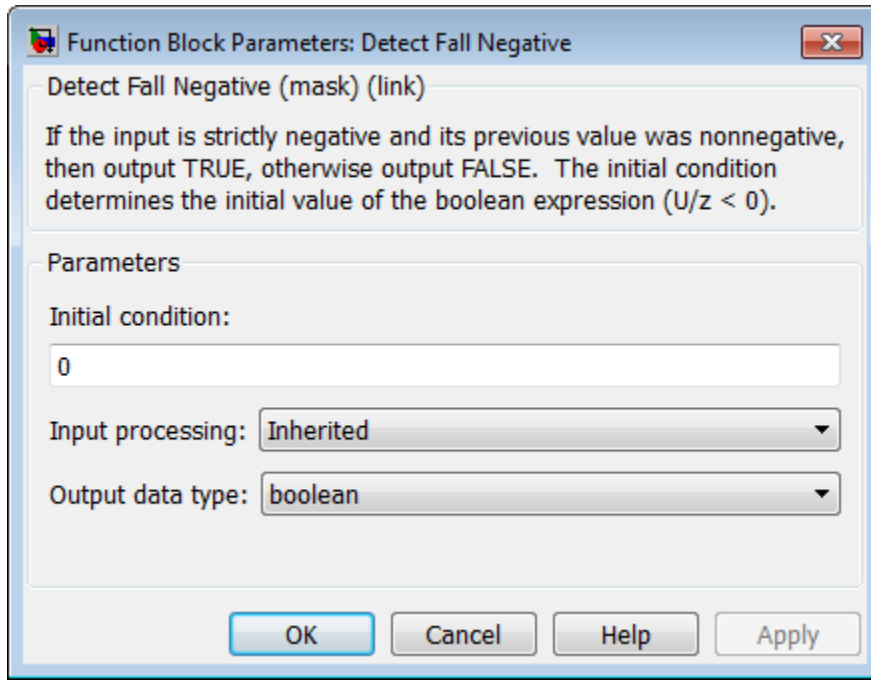
Data Type Support

The Detect Fall Negative block accepts signals of the following data types:

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

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

Parameters 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

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

Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

See Also

`Detect Change`, `Detect Decrease`, `Detect Fall Nonpositive`, `Detect Increase`, `Detect Rise Nonnegative`, `Detect Rise Positive`

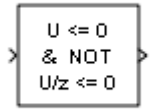
Introduced before R2006a

Detect Fall Nonpositive

Detect falling edge when signal value decreases to nonpositive value, and its previous value was strictly positive

Library

Logic and Bit Operations



Description

The Detect Fall Nonpositive block determines if the input is less than or equal to zero, and its previous value was greater than zero.

- The output is true (equal to 1) when the input signal is less than or equal to zero, and its previous value was greater than zero.
- The output is false (equal to 0) when the input signal is greater than zero, or if it is nonpositive, its previous value was also nonpositive.

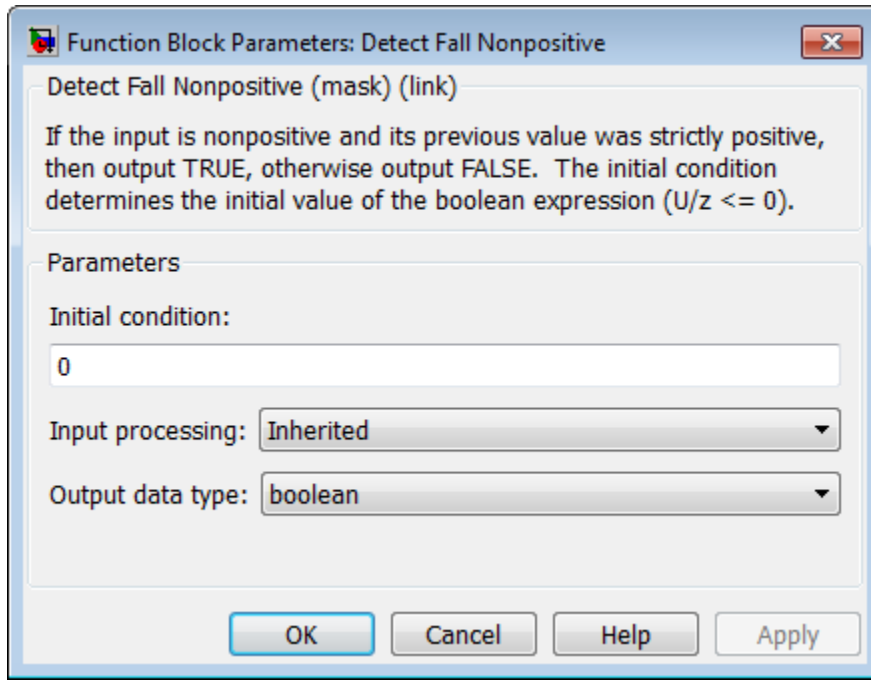
Data Type Support

The Detect Fall Nonpositive block accepts signals of the following data types:

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

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

Parameters 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

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

Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

See Also

`Detect Change`, `Detect Decrease`, `Detect Fall Negative`, `Detect Increase`, `Detect Rise Nonnegative`, `Detect Rise Positive`

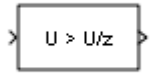
Introduced before R2006a

Detect Increase

Detect increase in signal value

Library

Logic and Bit Operations



Description

The Detect Increase block determines if an input is strictly greater than its previous value.

- The output is true (equal to 1) when the input signal is greater than its previous value.
- The output is false (equal to 0) when the input signal is less than or equal to its previous value.

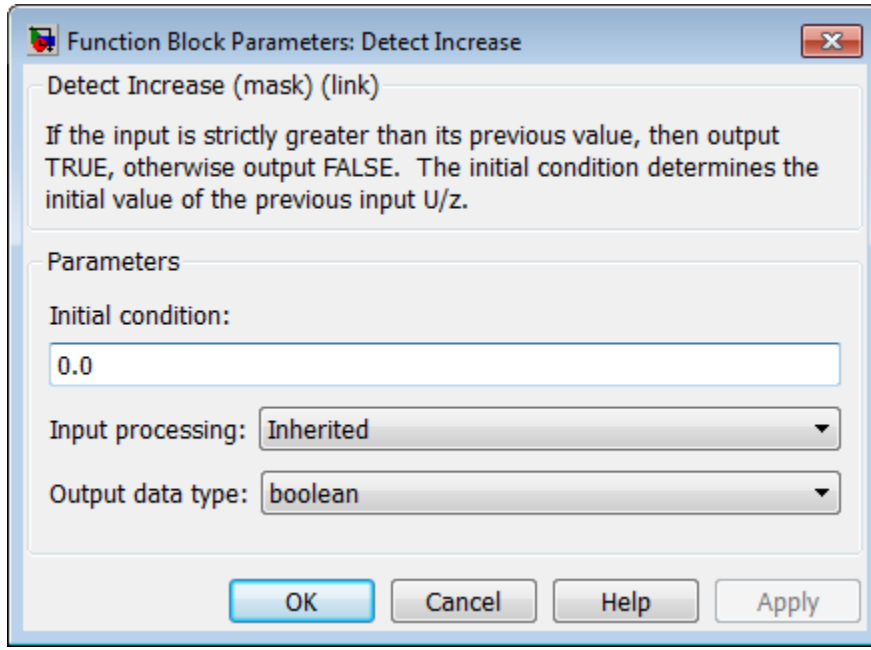
Data Type Support

The Detect Increase block accepts signals of the following data types:

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

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

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes

Code Generation	Yes

See Also

`Detect Change`, `Detect Decrease`, `Detect Fall Negative`, `Detect Fall Nonpositive`, `Detect Rise Nonnegative`, `Detect Rise Positive`

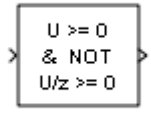
Introduced before R2006a

Detect Rise Nonnegative

Detect rising edge when signal value increases to nonnegative value, and its previous value was strictly negative

Library

Logic and Bit Operations



Description

The Detect Rise Nonnegative block determines if the input is greater than or equal to zero, and its previous value was less than zero.

- The output is true (equal to 1) when the input signal is greater than or equal to zero, and its previous value was less than zero.
- The output is false (equal to 0) when the input signal is less than zero, or if the input signal is nonnegative, its previous value was also nonnegative.

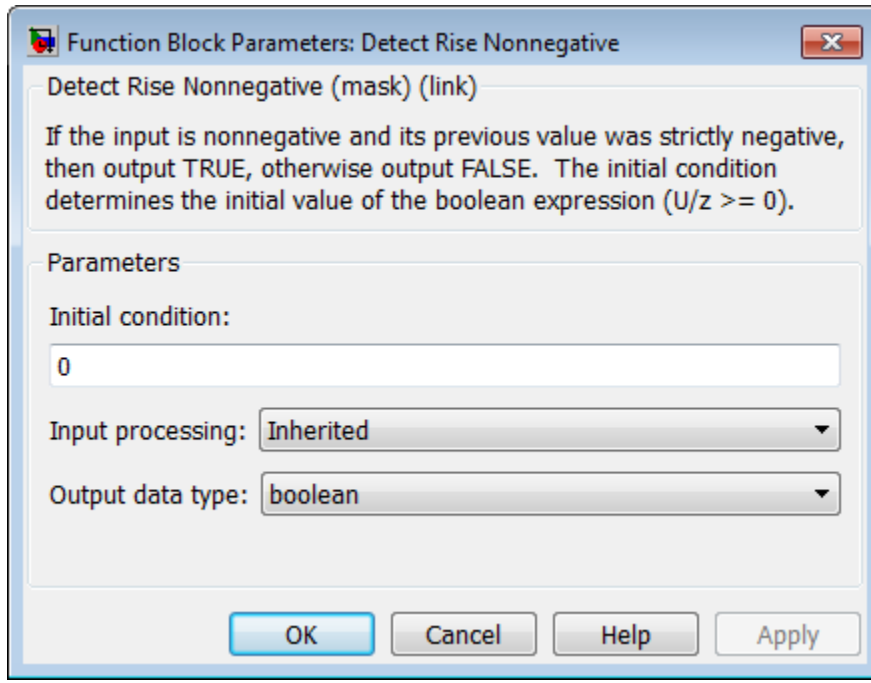
Data Type Support

The Detect Rise Nonnegative block accepts signals of the following data types:

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

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

Parameters 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

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

Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

See Also

`Detect Change`, `Detect Decrease`, `Detect Fall Negative`, `Detect Fall Nonpositive`, `Detect Increase`, `Detect Rise Positive`

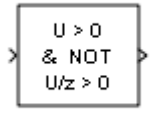
Introduced before R2006a

Detect Rise Positive

Detect rising edge when signal value increases to strictly positive value, and its previous value was nonpositive

Library

Logic and Bit Operations



Description

The Detect Rise Positive block determines if the input is strictly positive, and its previous value was nonpositive.

- The output is true (equal to 1) when the input signal is greater than zero, and the previous value was less than or equal to zero.
- The output is false (equal to 0) when the input is negative or zero, or if the input is positive, the previous value was also positive.

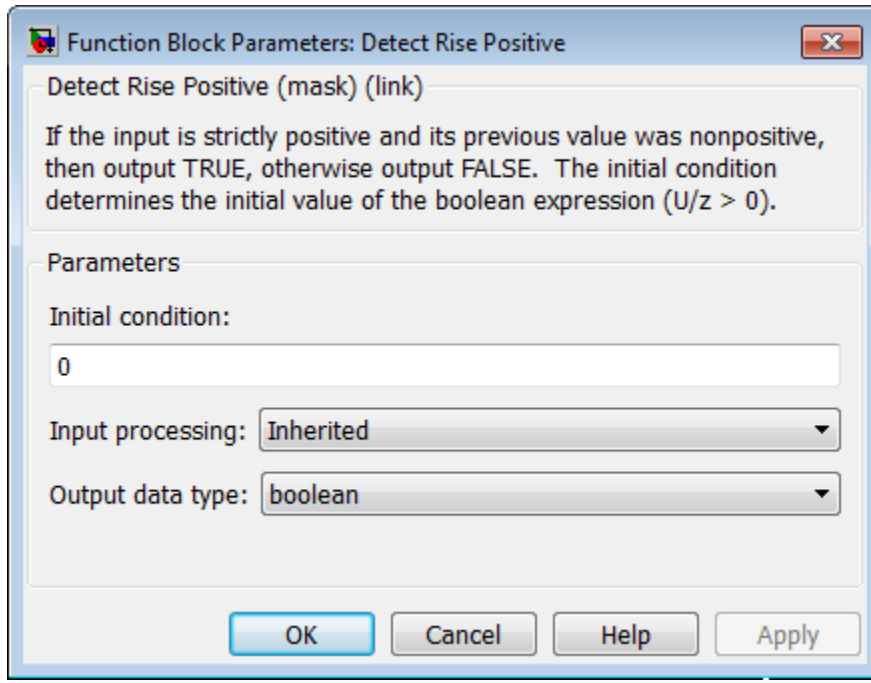
Data Type Support

The Detect Rise Positive block accepts signals of the following data types:

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

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

Parameters 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

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

Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

See Also

`Detect Change`, `Detect Decrease`, `Detect Fall Negative`, `Detect Fall Nonpositive`, `Detect Increase`, `Detect Rise Nonnegative`

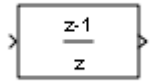
Introduced before R2006a

Difference

Calculate change in signal over one time step

Library

Discrete



Description

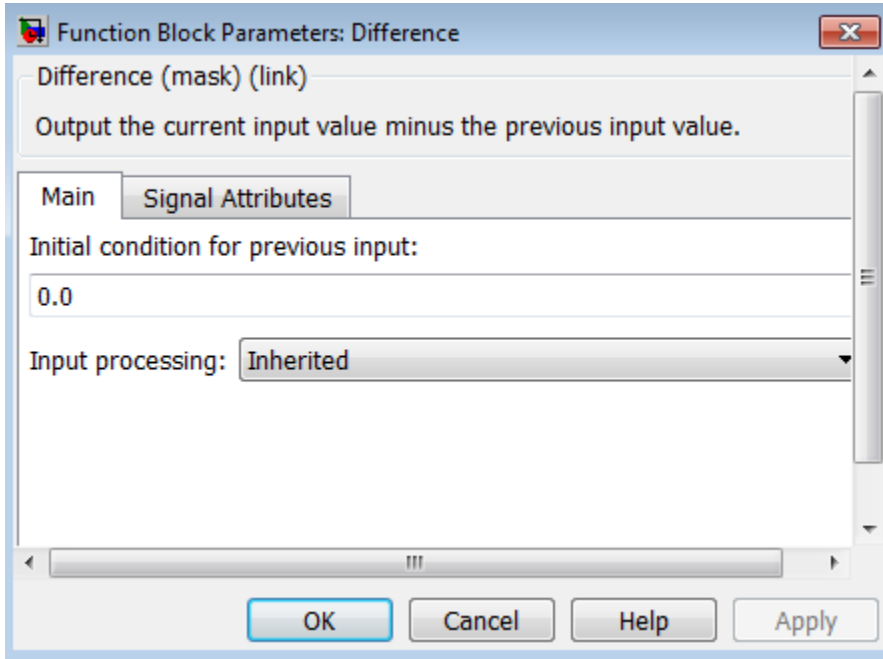
The Difference block outputs the current input value minus the previous input value.

Data Type Support

The Difference block accepts signals of any numeric data type that Simulink supports, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

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



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.

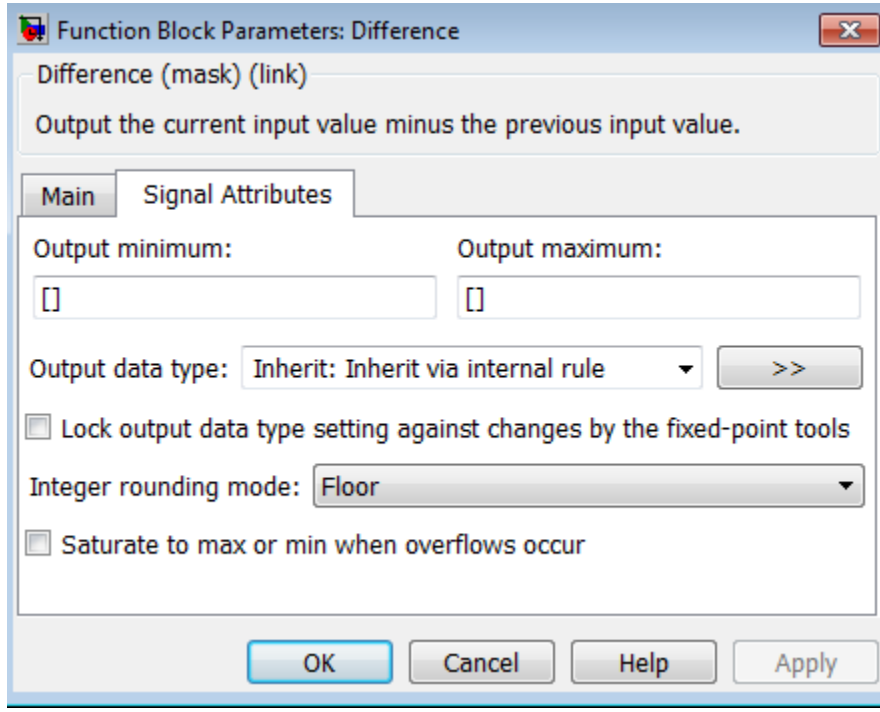
Note: When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

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:



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

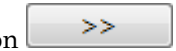
Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- 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 "Control Signal Data Types" in the Simulink User's Guide for more information.

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

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

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

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

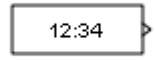
Introduced before R2006a

Digital Clock

Output simulation time at specified sampling interval

Library

Sources



Description

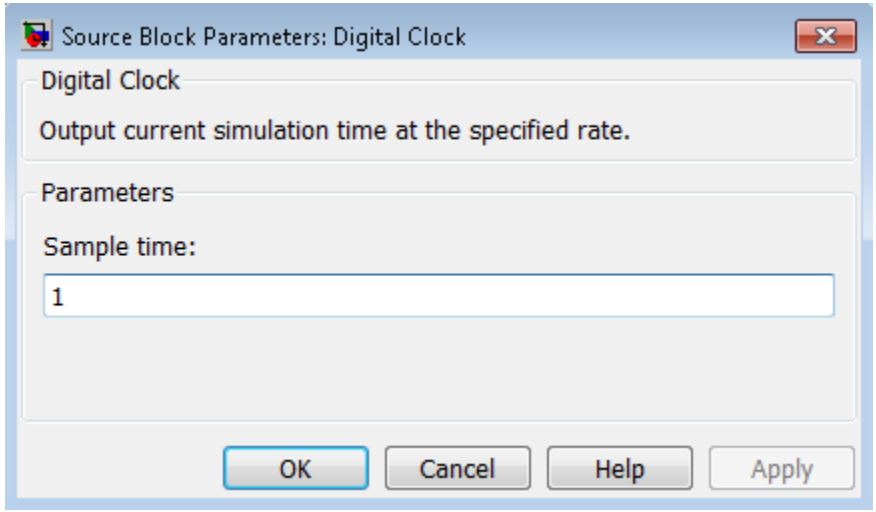
The Digital Clock block outputs the simulation time only at the specified sampling interval. At other times, the block holds the output at the previous value. To control the precision of this block, set the **Sample time** parameter in the block dialog box.

Use this block rather than the `Clock` block (which outputs continuous time) when you need the current simulation time within a discrete system.

Data Type Support

The Digital Clock block outputs a real signal of type `double`. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



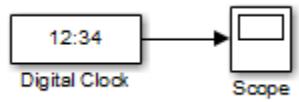
Sample time

Specify the sampling interval. The default value is `1` second. For more information, see Specifying Sample Time in the Simulink documentation.

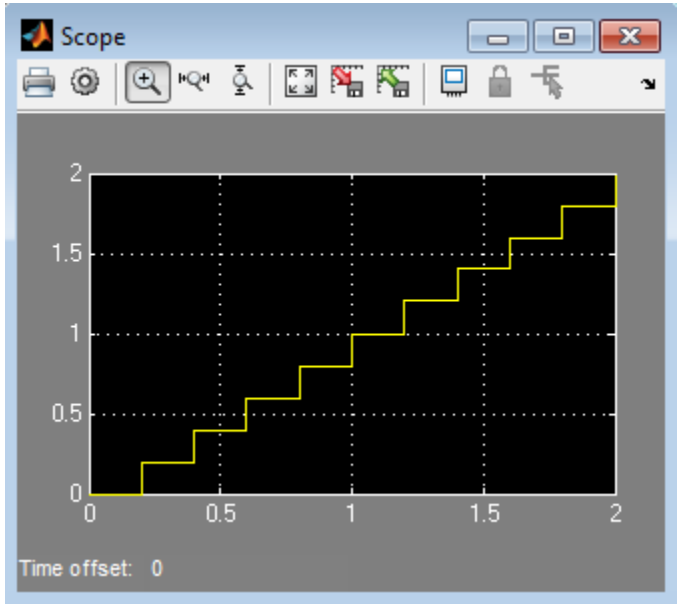
Do not specify a continuous sample time, either `0` or `[0,0]`. Also, avoid specifying `-1` (inheriting the sample time) because this block is a source.

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.



Characteristics

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

See Also

`Clock`

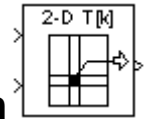
Introduced before R2006a

Direct Lookup Table (n-D)

Index into N-dimensional table to retrieve element, column, or 2-D matrix

Library

Lookup Tables

Description 

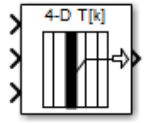
Block Inputs and Outputs

The Direct Lookup Table (n-D) block uses inputs as zero-based indices into an *n*-dimensional table. The number of inputs varies with the shape of the output: an element, column, or 2-D matrix.

You define a set of output values as the **Table data** parameter. The first input specifies the zero-based index to the table dimension that is *one higher* than the output dimensionality. The next input specifies the zero-based index to the next table dimension, and so on.

Output Shape	Output Dimensionality	Table Dimension That Maps to the First Input
Element	0	1
Column	1	2
Matrix	2	3

Suppose that you want to select a column of values from a 4-D table:



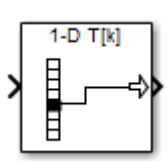
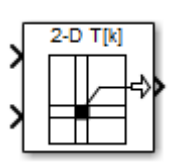
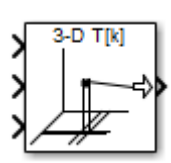
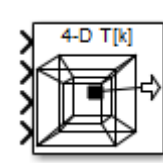
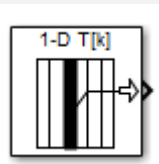
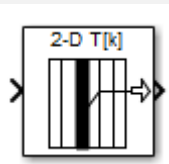
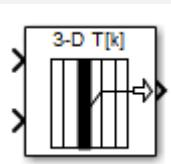
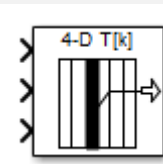
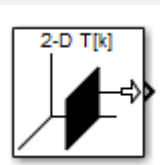
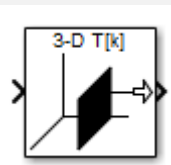
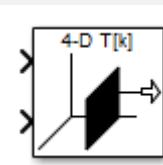
The following mapping of block input port to table dimension applies:

This input port...	Is the index for this table dimension...
1	2
2	3
3	4

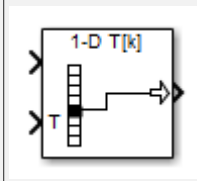
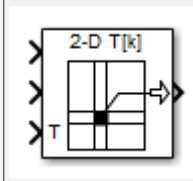
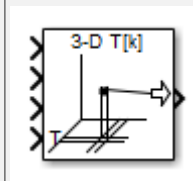
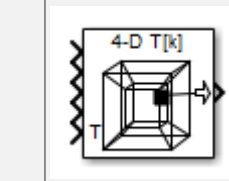
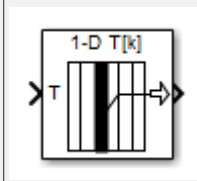
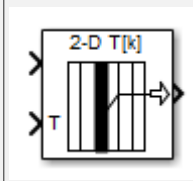
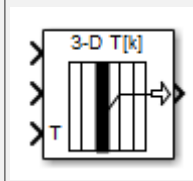
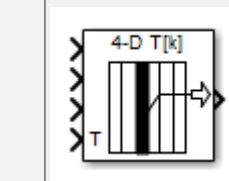
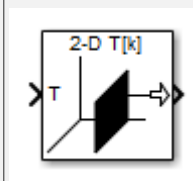
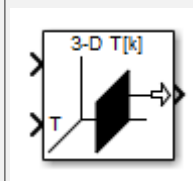
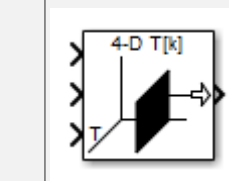
Changes in Block Icon Appearance

Depending on parameters you set, the block icon changes appearance. For table dimensions higher than 4, the icon matches the 4-D version but shows the exact number of dimensions in the top text.

When you use the **Table data** parameter, you see the following icons:

Object That Inputs Select from the Table	Number of Table Dimensions			
	1	2	3	4
Element				
Column				
2-D Matrix	Not applicable			

When you use the table input port, you see the following icons:

Object That Inputs Select from the Table	Number of Table Dimensions			
	1	2	3	4
Element				
Column				
2-D Matrix	Not applicable			

Data Type Support

The Direct Lookup Table (n-D) block accepts input signals of different data types.

Type of Input Port	Data Types Supported
Index port	• Floating point • Built-in integer • Boolean • Enumerated data types
Table port (with the label T)	• Floating point • Built-in integer • Fixed point • Boolean

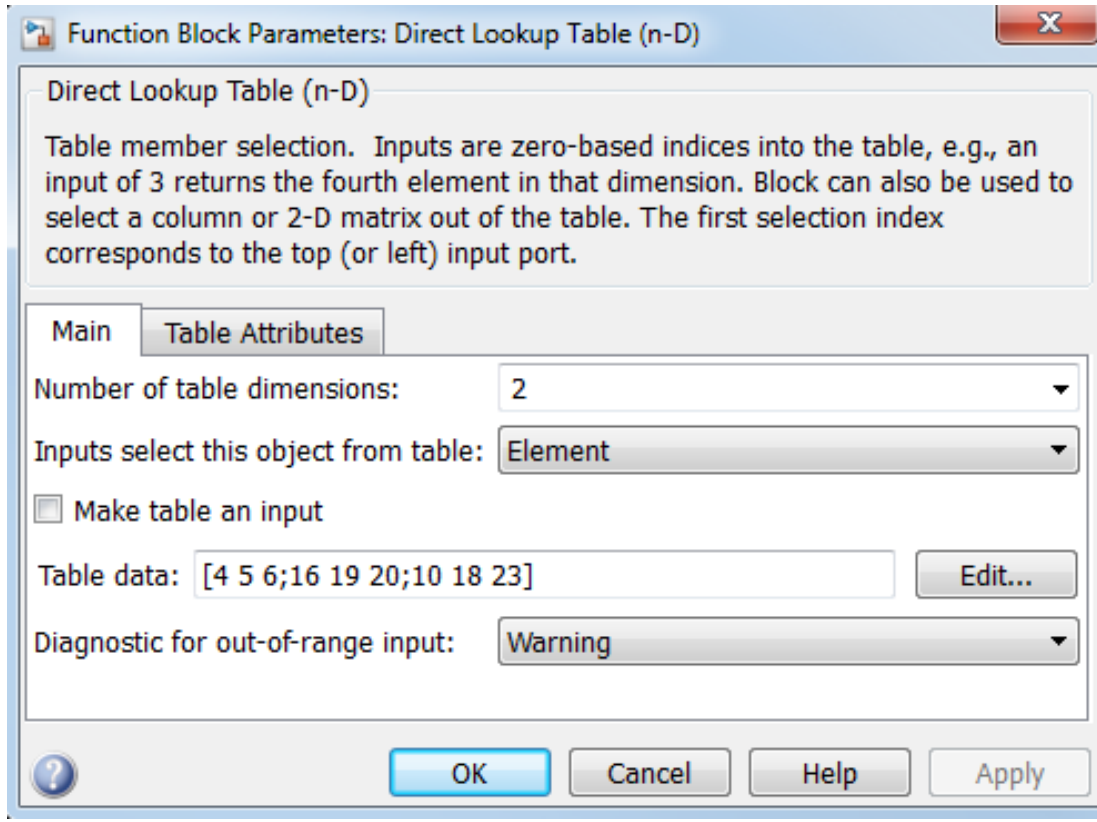
Type of Input Port	Data Types Supported
	• Enumerated data types

The output data type is the same as the table data type. Inputs for indexing must be real, but table data can be complex.

When the table data is...	The block inherits the output type from...
Not an input	The **Table data type** parameter
An input	The table input port

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

Parameters and Dialog Box



Main tab

Number of table dimensions

Specify the number of dimensions that the **Table data** parameter must have. This value determines the number of independent variables for the table and the number of inputs to the block.

To specify...	Do this...
1, 2, 3, or 4	Select the value from the drop-down list.
A higher number of table dimensions	Enter a positive integer directly in the field. The maximum number of table dimensions that this block supports is 30.

Inputs select this object from table

Specify whether the output data is a single element, a column, or a 2-D matrix. The number of input ports for indexing depends on your selection.

Selection	Number of Input Ports for Indexing
`Element`	Number of table dimensions
`Column`	Number of table dimensions – 1
`2-D Matrix`	Number of table dimensions – 2

This numbering matches MATLAB indexing. For example, if you have a 4-D table of data, follow these guidelines:

To access...	Specify...	As in...
An element	Four indices	`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 Lookup Tables" in the Simulink documentation.

Diagnostic for out-of-range input

Specify whether to show a warning or error when an index is out of range with respect to the table dimension. Options include:

- `None` — 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 in the Diagnostic 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

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

Table Attributes tab

Note: The parameters in the **Table Attributes** pane are not available if you select **Make table an input**. In this case, the block inherits all table attributes from the input port with the label `T`.

Table minimum

Specify the minimum value for table data. The default value is `[]` (unspecified).

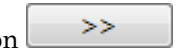
Table maximum

Specify the maximum value for table data. The default value is `[]` (unspecified).

Table data type

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

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

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Table data type** parameter.

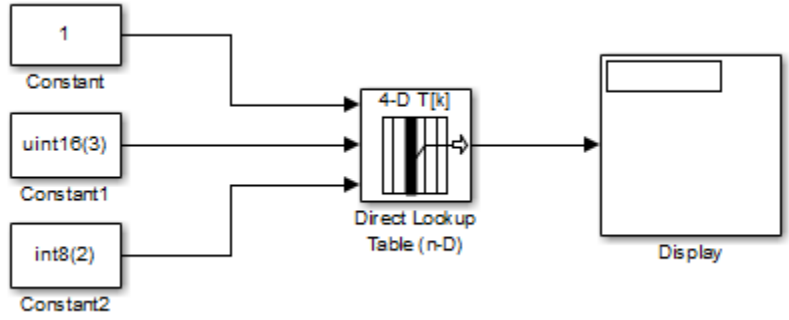
Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Lock the Output Data Type Setting" in the Fixed-Point Designer documentation.

Examples

When Table Data Is Not an Input

Suppose that you have the following model:



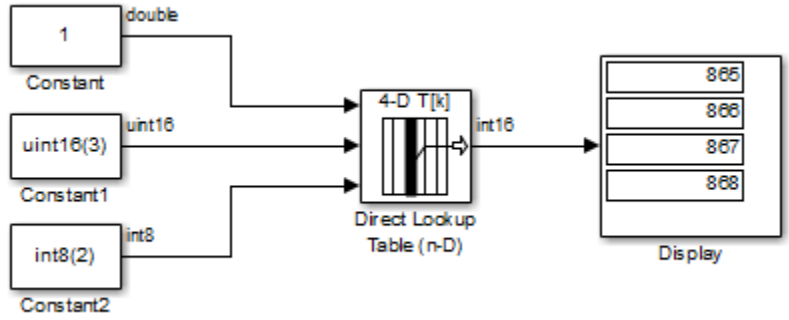
The Direct Lookup Table (n-D) block parameters are:

Block Parameter	Value
Number of table dimensions	`4`
Inputs select this object from table	`Column`
Make table an input	`off`
Table data	`a`
Diagnostic for out-of-range input	`Warning`
Sample time	`-1`
Table minimum	`[]`
Table maximum	`[]`
Table data type	`int16`
Lock data type settings against changes by the fixed-point tools	`off`

In this example, `a` is a 4-D array of linearly increasing values that you define with the following model preload function:

```
a = reshape(1:2800, [4 5 20 7]);
```

When you run the model, you get the following results:



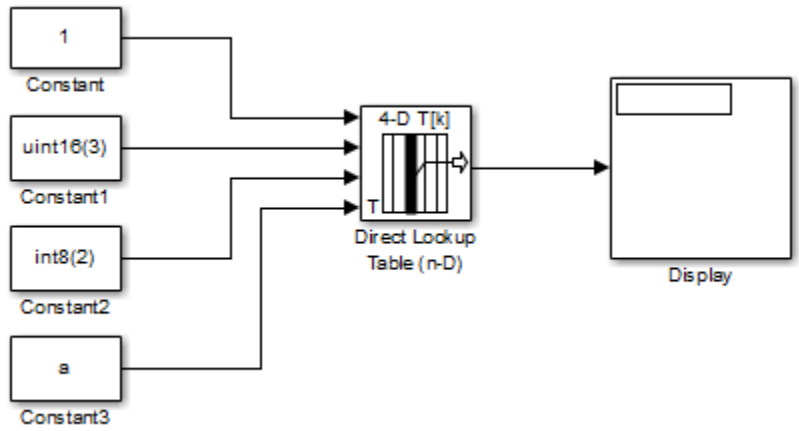
Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```
a(:,2,4,3)
```

The output has the same data type as the table: `int16`.

When Table Data Is an Input

Suppose that you have the following model:



The Direct Lookup Table (n-D) block parameters are:

Block Parameter	Value
Number of table dimensions	`4`
Inputs select this object from table	`Column`
Make table an input	`on`
Diagnostic for out-of-range input	`Warning`
Sample time	`–1`

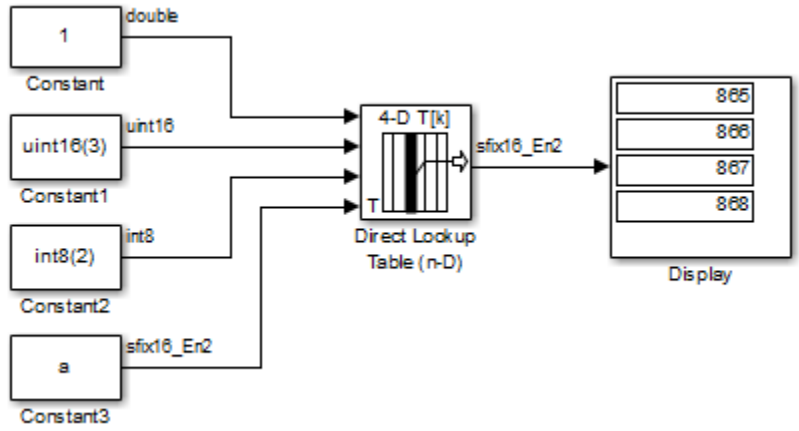
The key parameters of the Constant3 block are:

Block Parameter	Value
Constant value	`a`
Output data type	`fixdt(1,16,2)`

In this example, `a` is a 4-D array of linearly increasing values that you define with the following model preload function:

```
a = reshape(1:2800, [4 5 20 7]);
```

When you run the model, you get the following results:



The Constant3 block feeds the 4-D array to the Direct Lookup Table (n-D) block, using the fixed-point data type `fixdt(1,16,2)`. Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```
a(:,2,4,3)
```

The output has the same data type as the table: `fixdt(1,16,2)`.

Characteristics

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

Code Generation	Yes

See Also

`n-D Lookup Table`

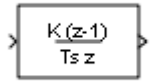
Introduced before R2006a

Discrete Derivative

Compute discrete-time derivative

Library

Discrete



Description

The Discrete Derivative block computes an optionally scaled discrete time derivative as follows

$$y(t_n) = \frac{Ku(t_n)}{T_s} - \frac{Ku(t_{n-1})}{T_s}$$

where

- $u(t_n)$ and $y(t_n)$ are the block's input and output at the current time step, respectively.
- $u(t_{n-1})$ is the block's input at the previous time step.
- K is a scaling factor.
- T_s is the simulation's discrete step size, which must be fixed.

Note: Do not use this block in subsystems with a non-periodic trigger (for example, non-periodic function-call subsystems). This configuration will produce inaccurate results.

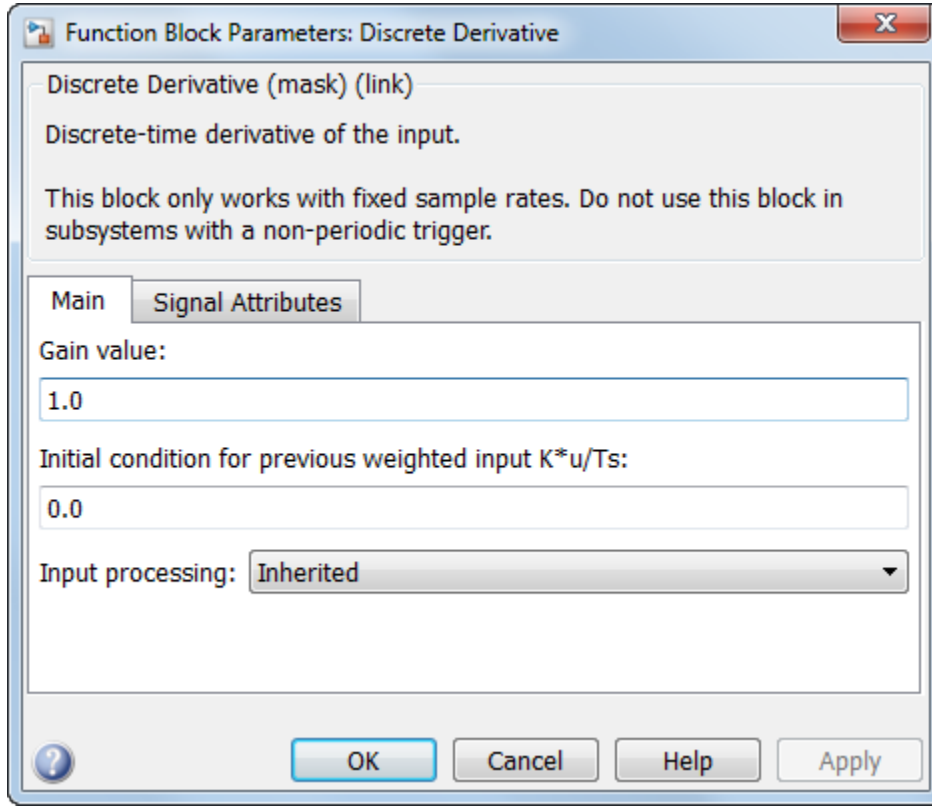
Data Type Support

The Discrete Derivative block supports all numeric Simulink data types, including fixed-point data types.

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

Parameters and Dialog Box

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



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.

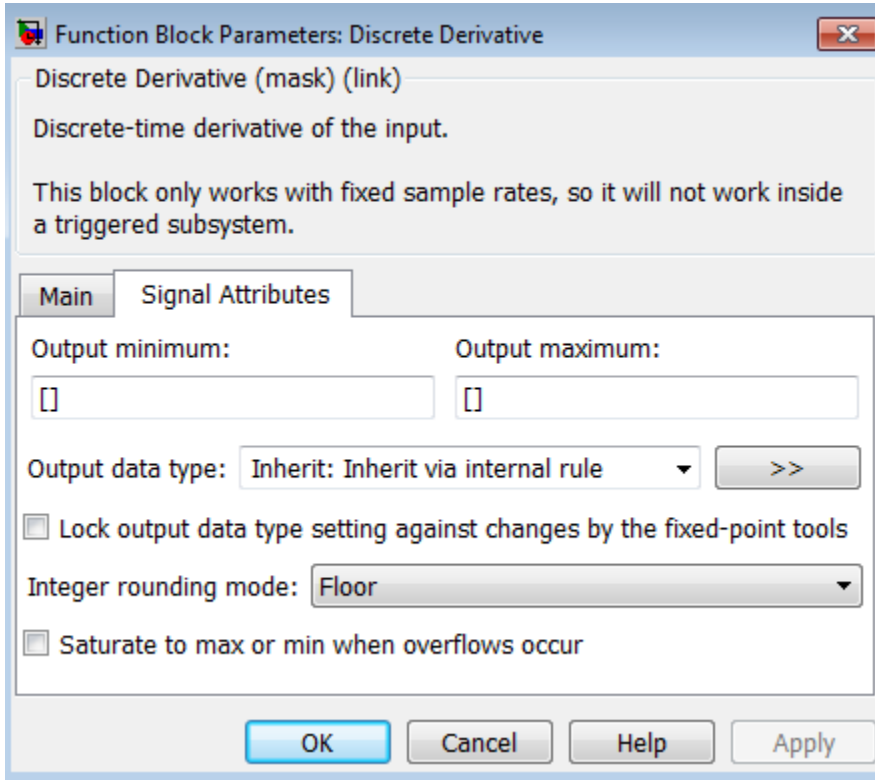
Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

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

Input Signal u	Input Processing Mode	Block Works?
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:



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

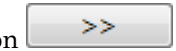
Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- 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 "Control Signal Data Types" in Simulink User's Guide for more information.

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

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

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

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

See Also

`Derivative`

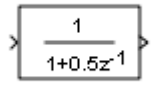
Introduced before R2006a

Discrete Filter

Model Infinite Impulse Response (IIR) filters

Library

Discrete



Description

The Discrete Filter block independently filters each channel of the input signal with the specified digital IIR filter. You can specify the filter structure as one of `| Direct form I | Direct form I transposed | Direct form II | Direct form II transposed`. The block implements static filters with fixed coefficients. You can tune the coefficients of these static filters.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify how the block treats each element of the input. You can specify treating input elements as an independent channel (sample-based processing), or treating each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal those of the input, except when you specify a matrix of filter taps for the **Numerator coefficients** parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

Use the **Numerator coefficients** parameter to specify the coefficients of the discrete filter numerator polynomial. Use the **Denominator coefficients** parameter to specify the coefficients of the denominator polynomial of the function. The **Denominator coefficients** parameter must be a vector of coefficients.

Specify the coefficients of the numerator and denominator polynomials in ascending powers of z^{-1}. The Discrete Filter block lets you use polynomials in z^{-1} (the delay operator)

to represent a discrete system. This method is the one that signal processing engineers typically use. Conversely, the Discrete Transfer Fcn block lets you use polynomials in z to represent a discrete system. This method is the one that control engineers typically use. When the numerator and denominator polynomials have the same length, the two methods are identical.

Specifying Initial States

In **Dialog parameters** and **Input port(s)** modes, the block initializes the internal filter states to zero by default, which is equivalent to assuming past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial states for the filter delays.

To determine the number of initial state values you must specify, and how to specify them, see the following table on Valid Initial States and Number of Delay Elements (Filter States). The **Initial states** parameter can take one of four forms as described in the following table.

Valid Initial States

Initial state	Examples	Description
Scalar	`5` Each delay element for each channel is set to `5`.	The block initializes all delay elements in the filter to the scalar value.
Vector (for applying the same delay elements to each channel)	For a filter with two delay elements: [d_1 d_2] The delay elements for all channels are d1 and d2.	Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)).
Vector or matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: [d_1 d_2 D_1 D_2 d_1 d_2] or	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

Initial state	Examples	Description
	$\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$ • 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.	filter (specified in the table Number of Delay Elements (Filter States)). • The matrix must have the same number of rows as the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)), and must have one column for each channel of the input signal.
Empty matrix	[ ] Each delay element for each channel is set to 0.	The empty matrix, [ ], is equivalent to setting the **Initial conditions** parameter to the scalar value 0.

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.

Number of Delay Elements (Filter States)

Filter Structure	Number of Delay Elements per Channel
`Direct form I` `Direct form I transposed`	• `number of zeros - 1` • `number of poles - 1`
`Direct form II` `Direct form II transposed`	`max(number of zeros, number of poles)-1`

The following tables describe the valid initial states for different sizes of input and different number of channels. These tables provide this information according to whether you set the **Input processing** parameter to frame based or sample based.

Frame-Based Processing

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
• Column vector (K-by-1) • Unoriented vector (K)	1	• Scalar • Column vector (M-by-1)	• Scalar • Column vector (M-by-1)

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
		• Row vector (1-by-*M*)	
• Row vector (1-by-*N*) • Matrix (*K*-by-*N*)	*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-*N*)	• Scalar • Matrix (*M*-by-*N*)

Sample-Based Processing

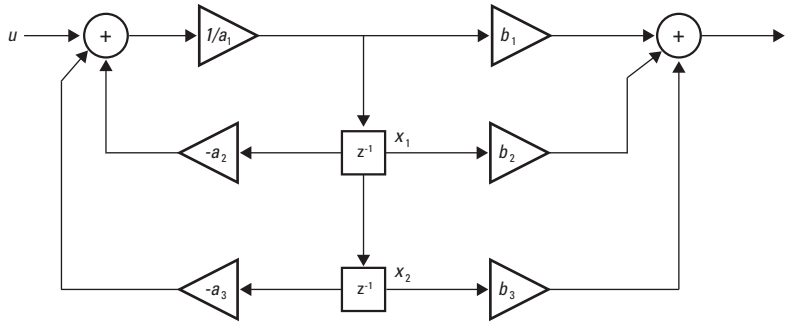
Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
• Scalar	1	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)
• Row vector (1-by-*N*) • Column vector (*N*-by–1) • Unoriented vector (*N*)	*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-*N*)	• Scalar
• Matrix (*K*-by-*N*)	*K*×*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-(*K*×*N*))	• Scalar

When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. Enter `0` to initialize all states to zero. When the **Initial states** is a vector or a matrix, each vector or matrix element specifies a unique initial state. This unique state corresponds to a delay element in a corresponding channel:

- The vector length must equal the number of delay elements in the filter, `M = max(number of zeros, number of poles)`.
- The matrix must have the same number of rows as the number of delay elements in the filter, `M = max(number of zeros, number of poles)`. The matrix must also have one column for each channel of the input signal.

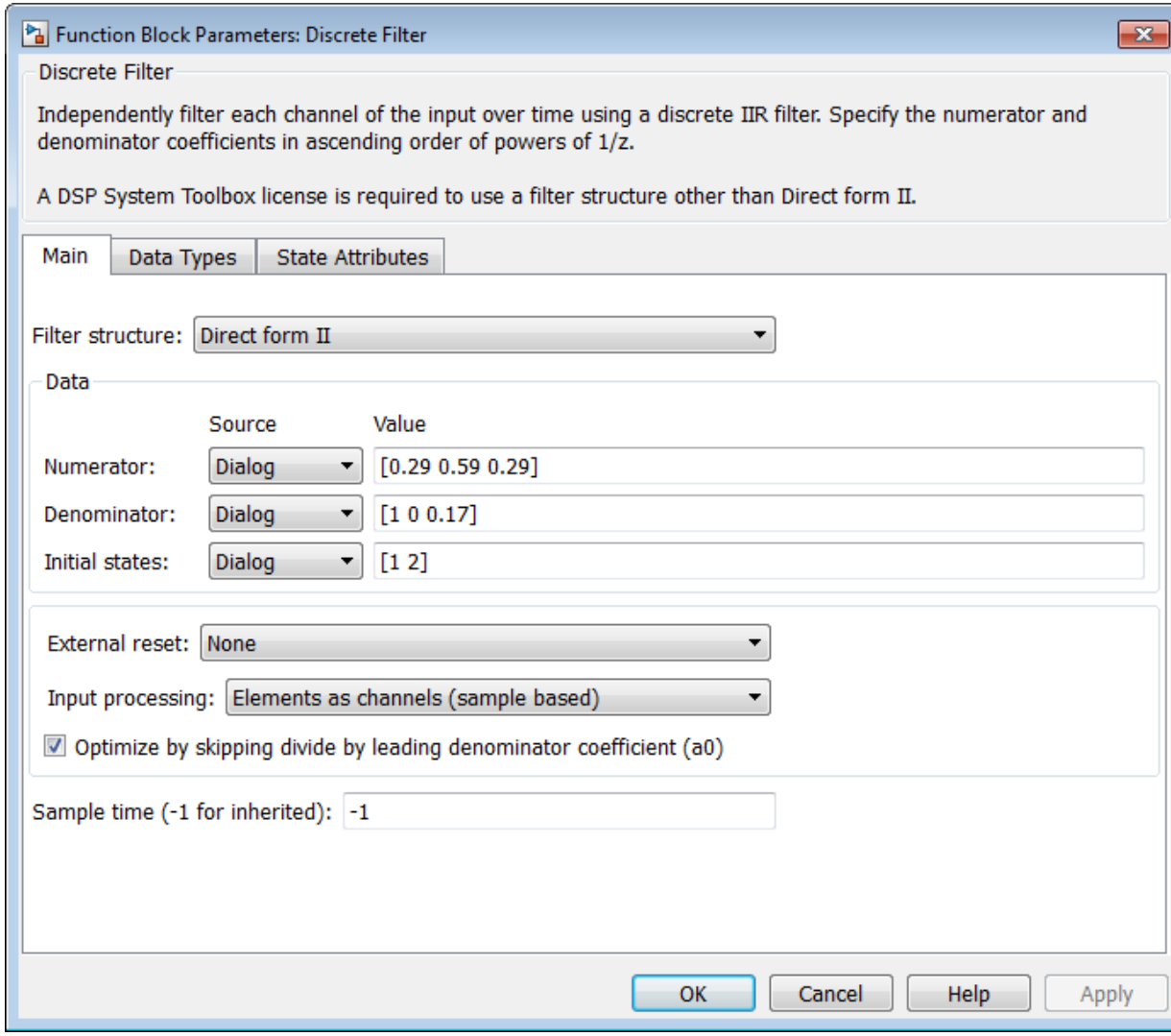
The following example shows the relationship between the initial filter output and the initial input and state. Given an initial input u_1, the first output y_1 is related to the initial state [x_1, x_2] and initial input by:

$$y_1 = b_1\left[\frac{\left(u_1 - a_2x_1 - a_3x_2\right)}{a_1}\right] + b_2x_1 + b_3x_2$$

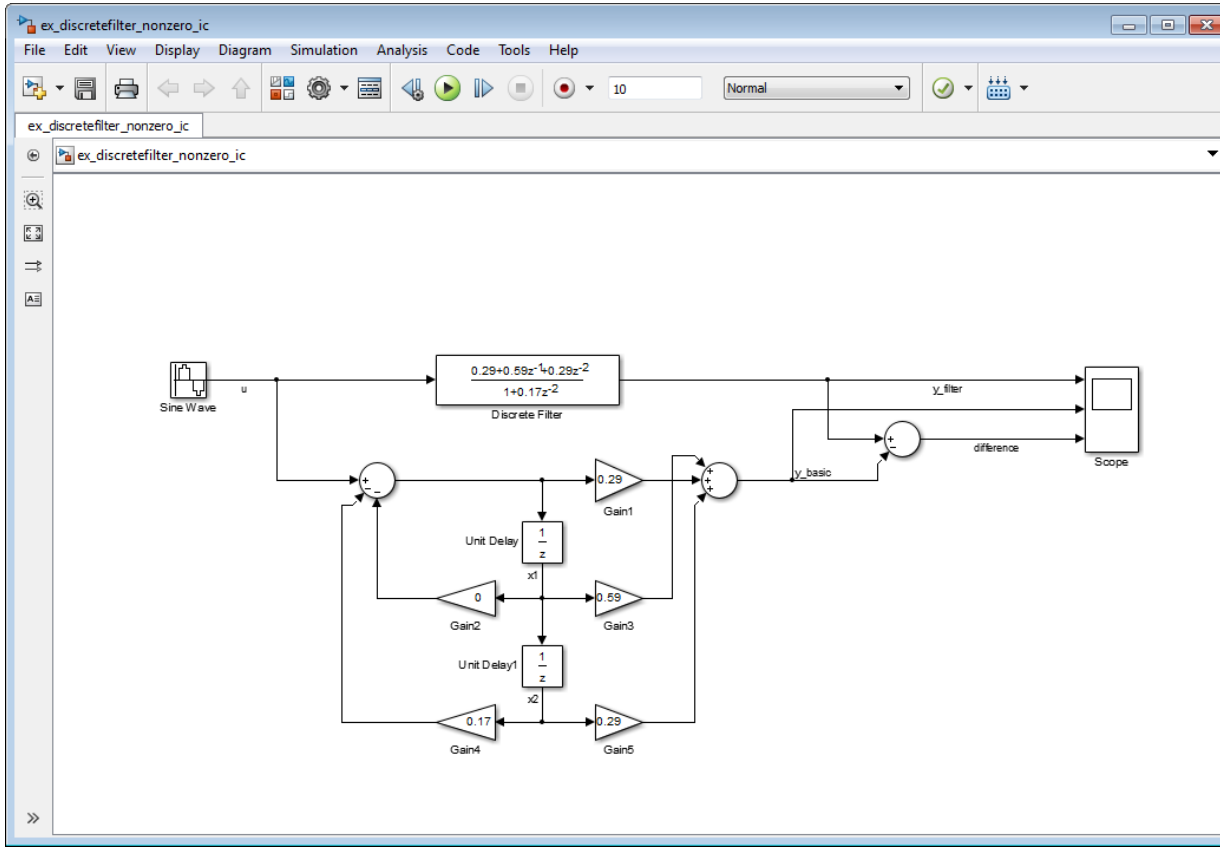


To see an example of how to set initial conditions as a vector:

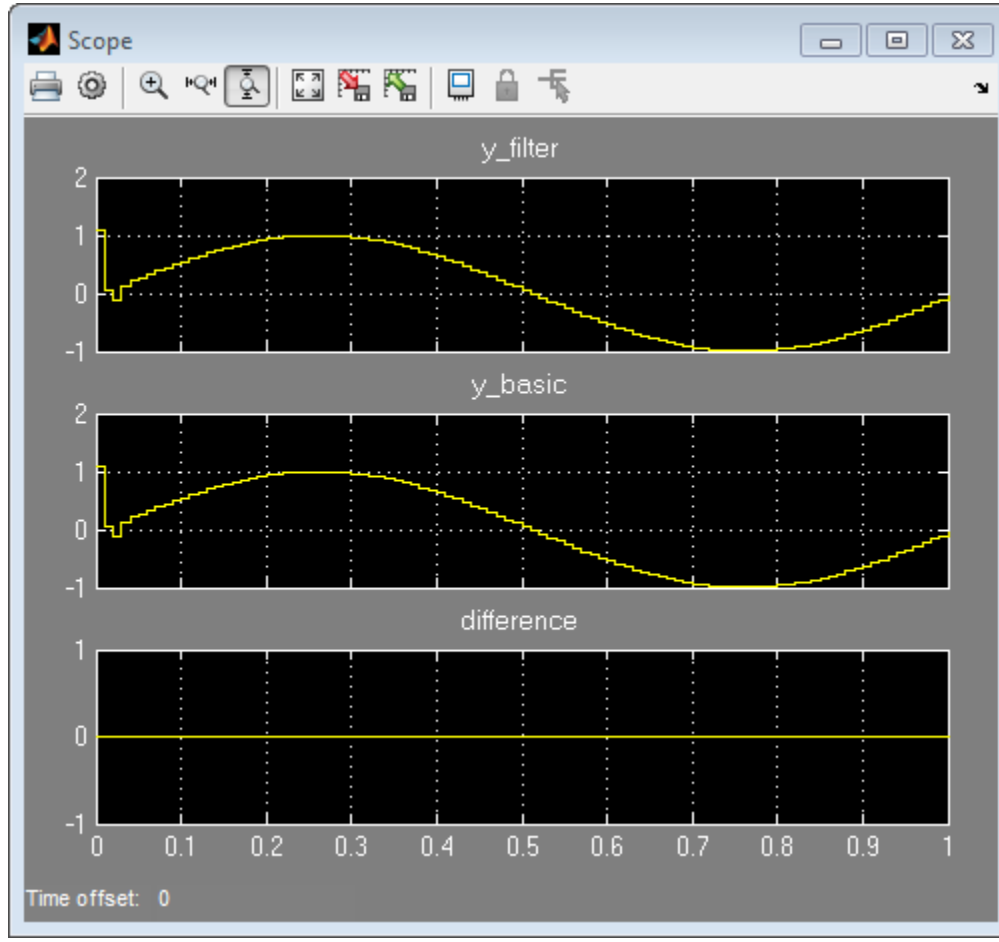
- Click on the model `ex_discretefilter_nonzero_ic`, or type it at the MATLAB command prompt.
- Double-click on the Discrete Filter block, and set the parameters. The following shows how to set the initial conditions of the Discrete Filter block to [1 2].



- Simulate the model, by left-clicking the green simulation icon.



- Double-click the scope. You can see that the difference between the signal filtered by the Discrete Filter block, and the signal from the filter's building blocks, is zero.



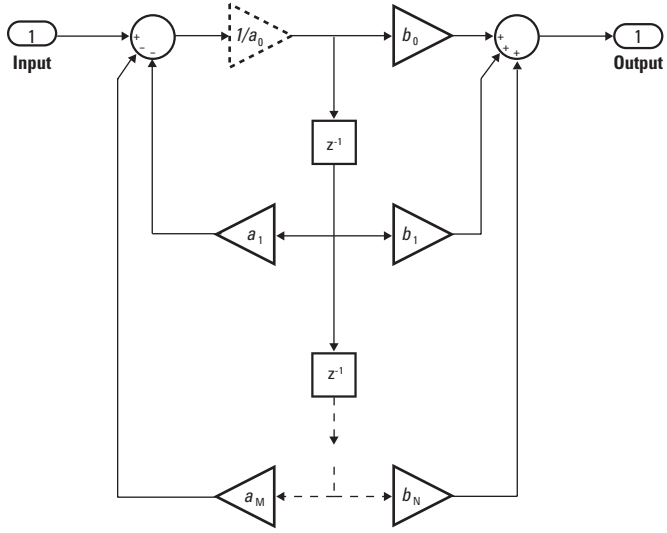
This demonstrates that you can enter the initial conditions of the Discrete Filter block as a vector of [1 2]. You can also set the initial condition of the first Unit Delay to 1 and the second Unit Delay to 2. The resulting outputs are the same.

Data Type Support

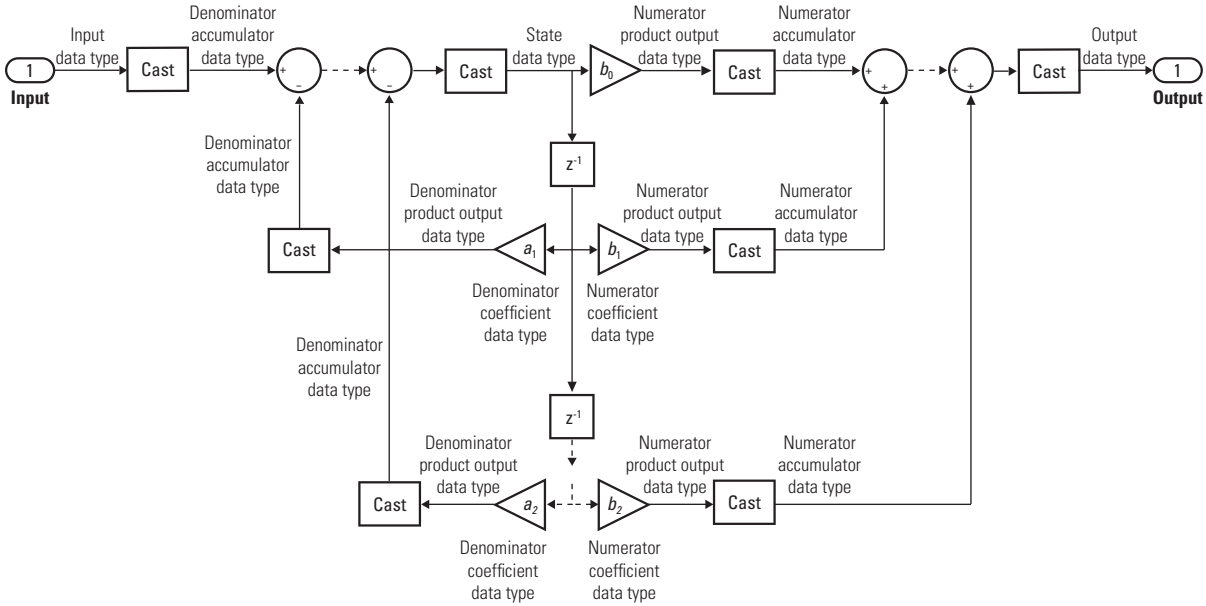
The Discrete Filter block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

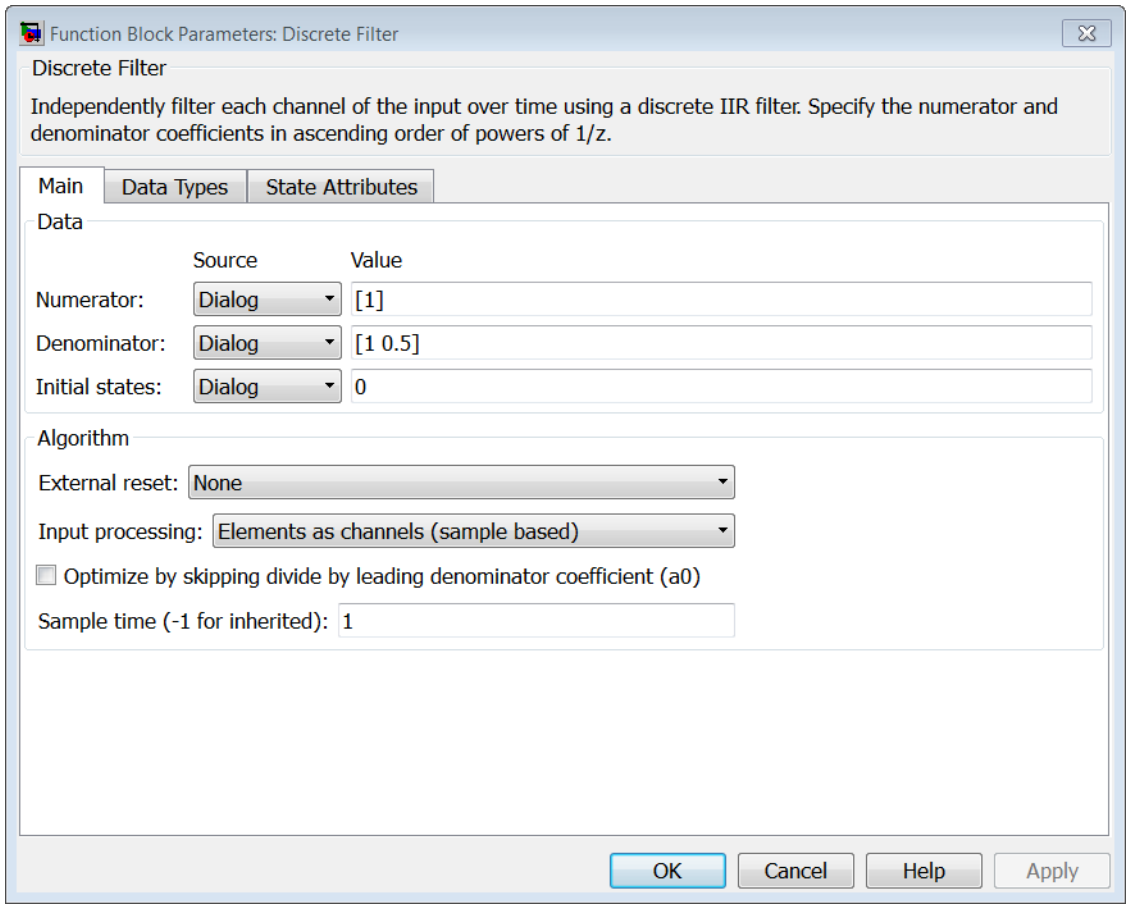
The following diagrams show the filter structure and the data types used within the Discrete Filter block for fixed-point signals.



The block omits the dashed divide when you select the **Optimize by skipping divide by leading denominator coefficient (a0)** parameter.


1
Input
Input data type
Cast
Denominator accumulator data type
Cast
State data type
b_0
Numerator product output data type
Cast
Numerator accumulator data type
Cast
Output data type
1
Output
z^{-1}
Denominator accumulator data type
Cast
Denominator product output data type
a_1
Denominator coefficient data type
b_1
Numerator coefficient data type
Numerator product output data type
Cast
Numerator accumulator data type
z^{-1}
Denominator accumulator data type
Cast
Denominator product output data type
a_2
Denominator coefficient data type
b_2
Numerator coefficient data type
Numerator product output data type
Cast
Numerator accumulator data type

Parameters and Dialog Box



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 1-446.

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 the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
`Level hold`	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Input processing

Specify whether the block performs sample- or frame-based processing.

- `Elements as channels (sample based)` — Process each element of the input as an independent channel.
- `Columns as channels (frame based)` — Process each column of the input as an independent channel.

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

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

Optimize by skipping divide by leading denominator coefficient (a0)

Select when the leading denominator coefficient, a_0, equals 1. This parameter optimizes your code.

When you select this check box, the block does not perform a divide-by-a_0 either in simulation or in the generated code. An error occurs if a_0 is not equal to one.

When you clear this check box, the block is fully tunable during simulation. It performs a divide-by-a_0 in both simulation and code generation.

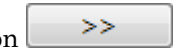
Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in "How Simulink Works" in the *Simulink User's Guide*.

State

Specify the state data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **State** parameter.

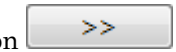
See "Specify Data Types Using Data Type Assistant" for more information.

Numerator coefficients

Specify the numerator coefficient data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See "Specify Data Types Using Data Type Assistant" for more information.

Numerator coefficient minimum

Specify the minimum value that a numerator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges")
- Automatic scaling of fixed-point data types

Numerator coefficient maximum

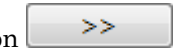
Specify the maximum value that a numerator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges")
- 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)`

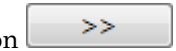
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator product output** parameter.

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)`

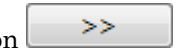
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator accumulator** parameter.

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 "Control Block Parameter Value Ranges")
- 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:

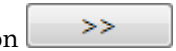
- Parameter range checking (see "Control Block Parameter Value Ranges")

- Automatic scaling of fixed-point data types

Denominator product output

Specify the product output data type for the denominator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

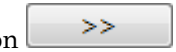
to display the **Data Type Assistant**, which helps you set the **Denominator product output** parameter.

See "Specify Data Types Using Data Type Assistant" for more information.

Denominator accumulator

Specify the accumulator data type for the denominator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Denominator accumulator** parameter.

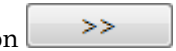
See "Specify Data Types Using Data Type Assistant" for more information.

Output

Specify the output data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See "Control Signal Data Types" for more information.

Output minimum

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

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

Output maximum

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

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

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

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Lock the Output Data Type Setting" 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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
			Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see "Discrete Block State Naming in Generated Code" 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**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name resolves
- Minimum and maximum values of the signal object

For more information, see "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation.

Characteristics

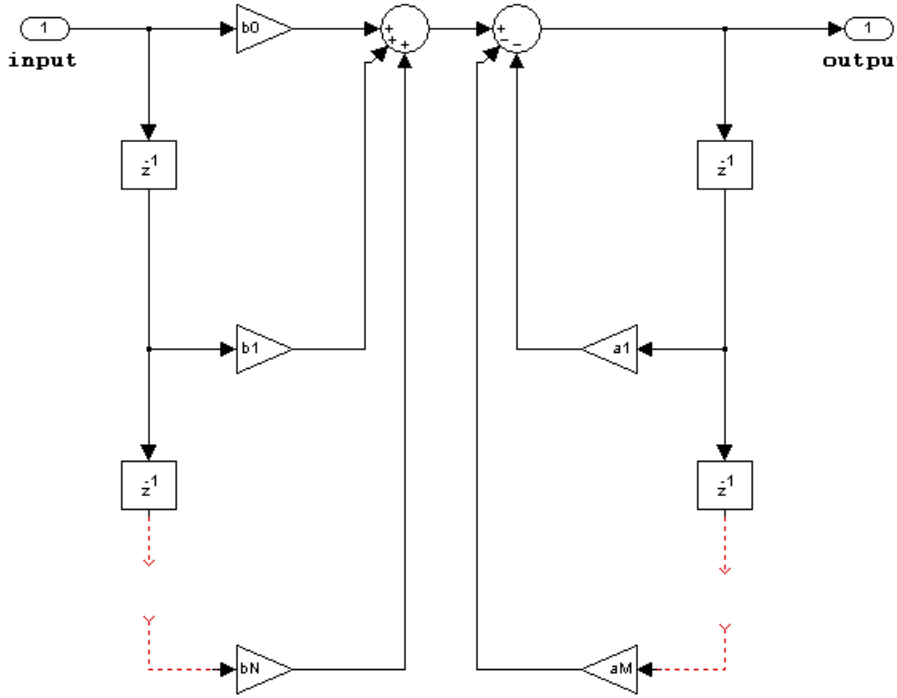
Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Only when the leading numerator coefficient does not equal zero
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Code Generation	Yes

Filter Structure Diagrams

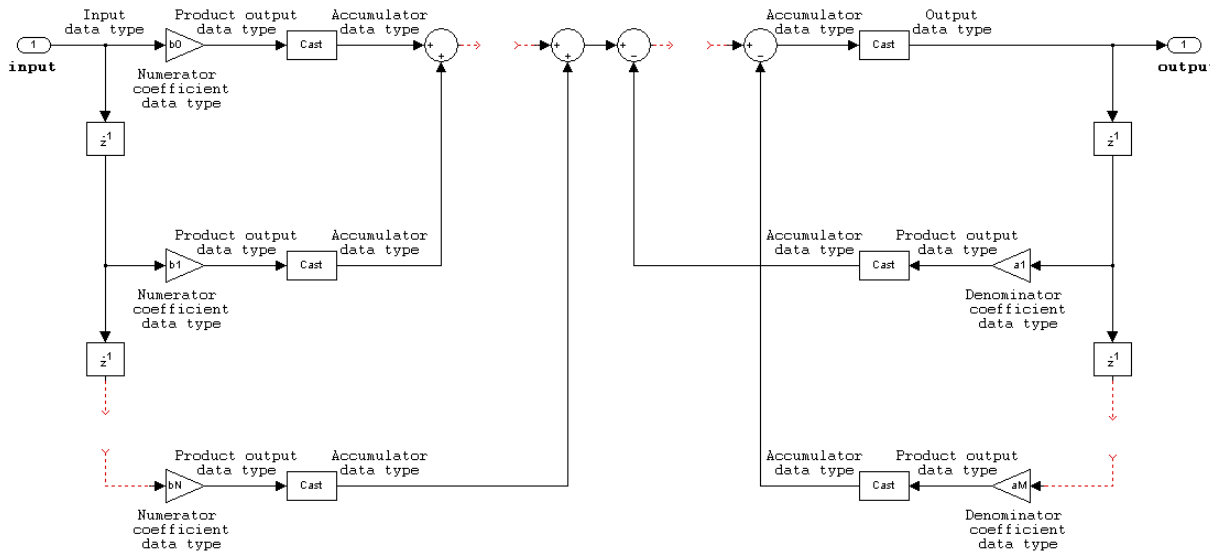
The diagrams in the following sections show the filter structures supported by the Digital Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block dialog.

IIR direct form I

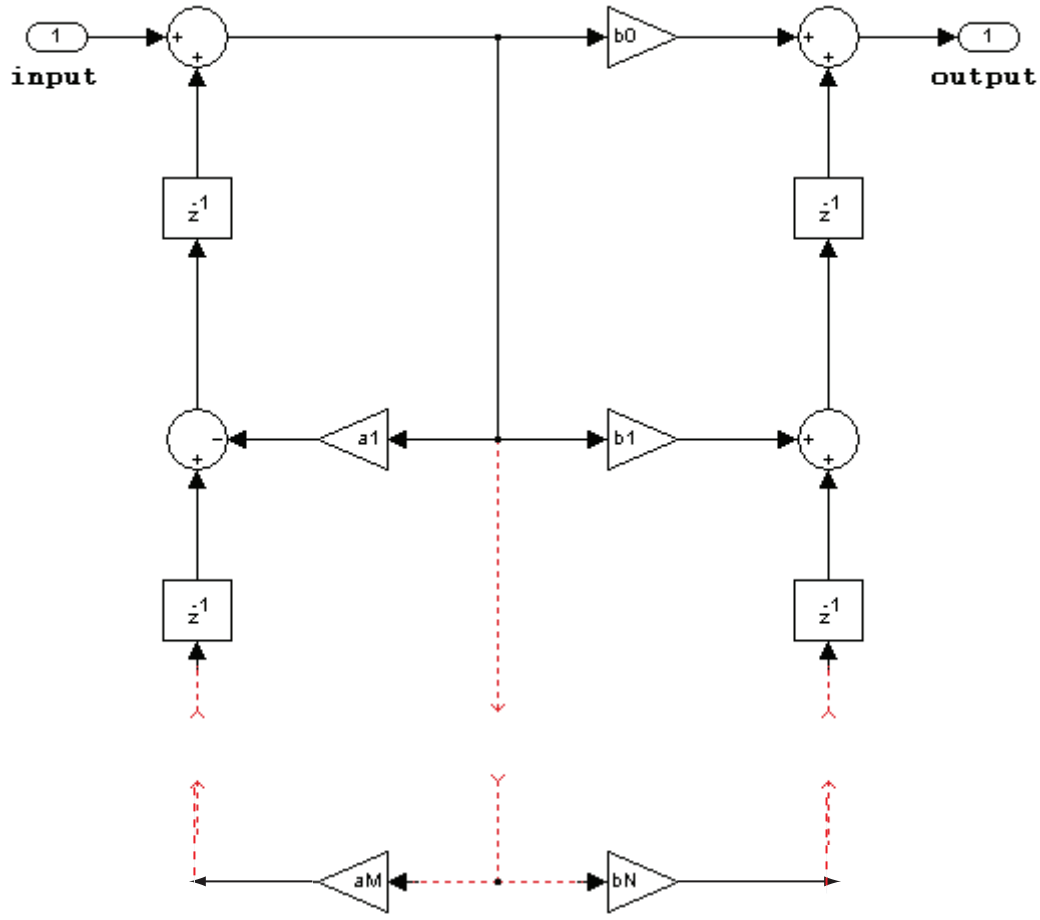


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- 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.



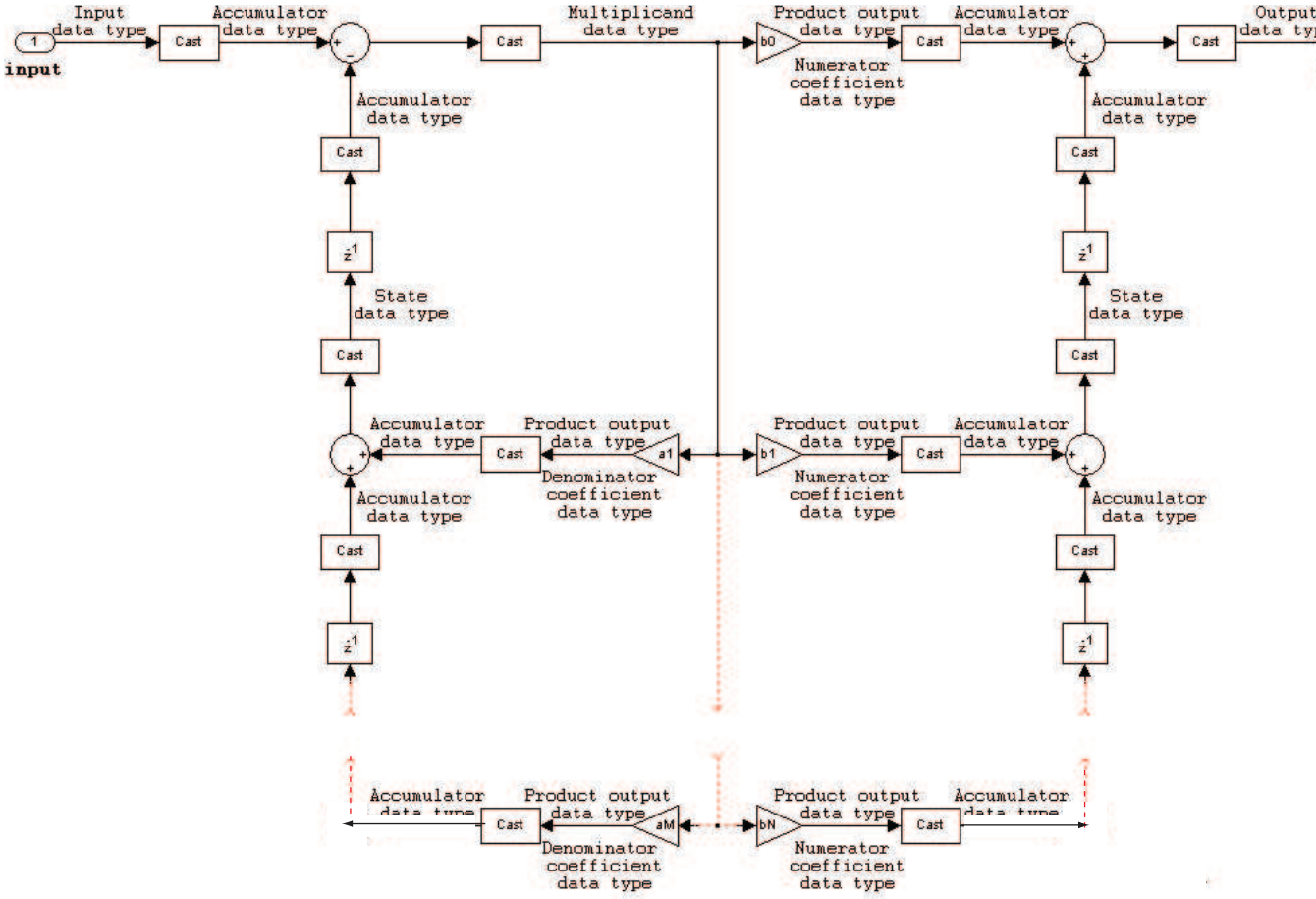
IIR direct form I transposed



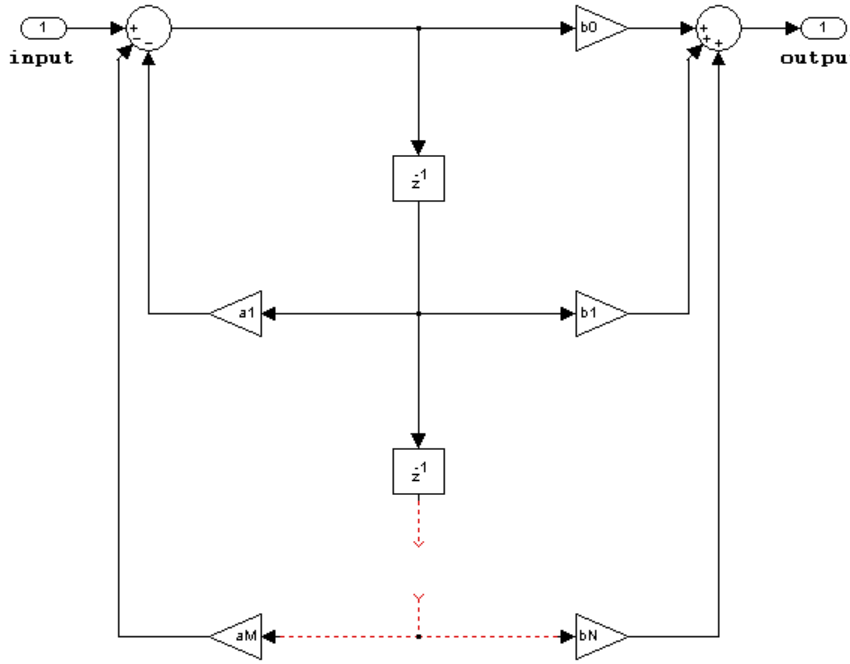
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- 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.


1
input
Input data type
Cast
Accumulator data type
Cast
Multiplicand data type
b0
Product output data type
Cast
Numerator coefficient data type
Accumulator data type
Cast
Output data ty
Accumulator data type
Cast
z-1
State data type
Cast
Accumulator data type
Cast
Product output data type
a1
Denominator coefficient data type
Accumulator data type
Cast
z-1
b1
Product output data type
Cast
Numerator coefficient data type
Accumulator data type
Cast
z-1
State data type
Cast
Accumulator data type
Cast
z-1
Accumulator data type
Cast
Product output data type
aM
Denominator coefficient data type
bN
Product output data type
Cast
Numerator coefficient data type
Accumulator data type

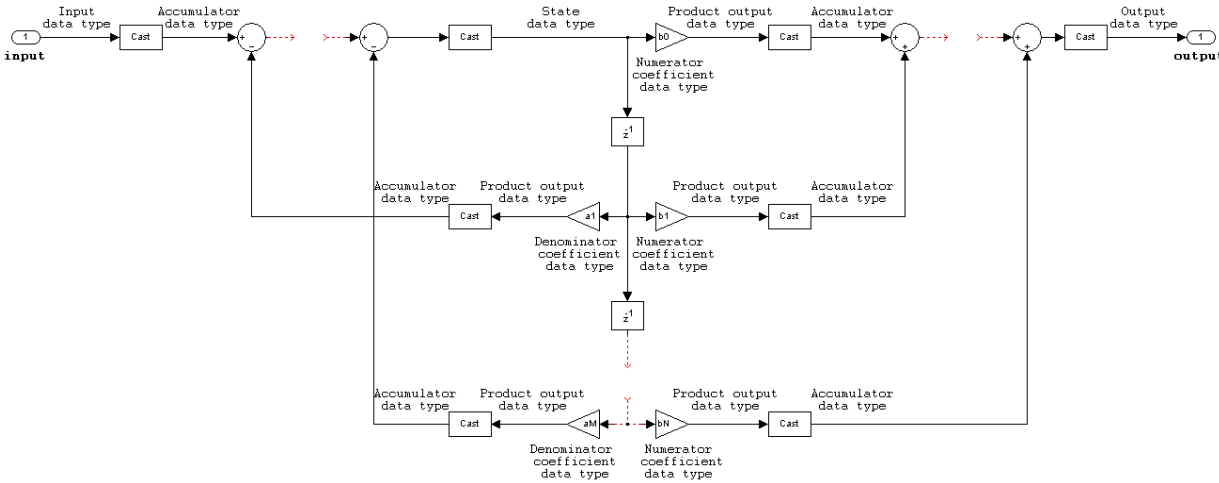
IIR direct form II



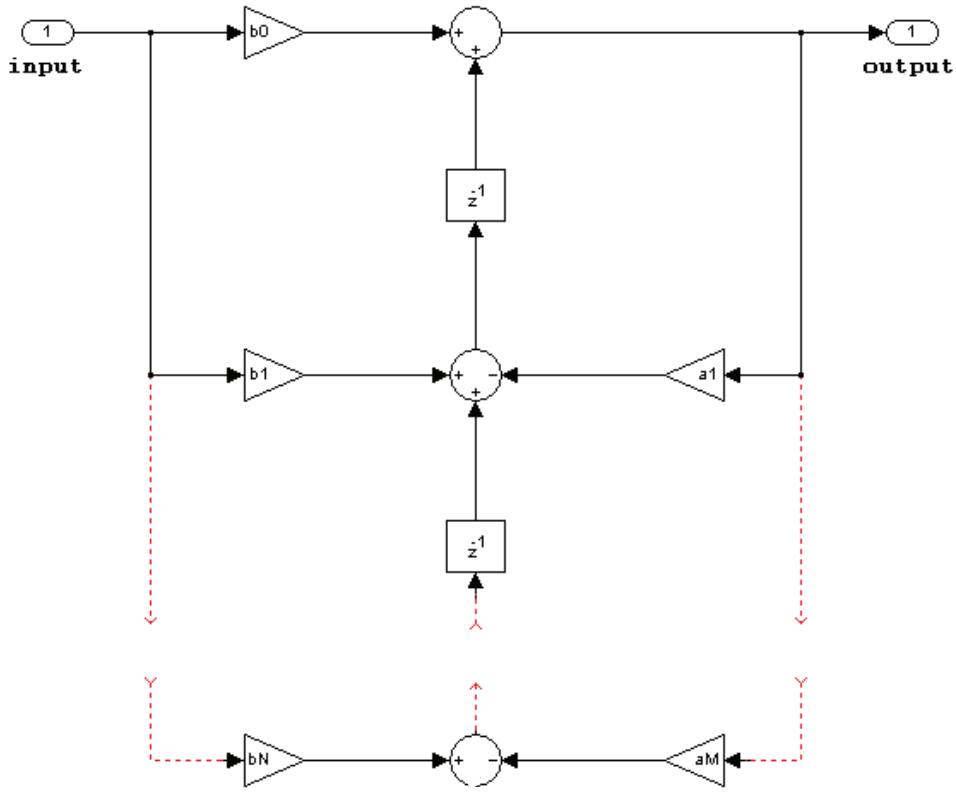
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- 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.



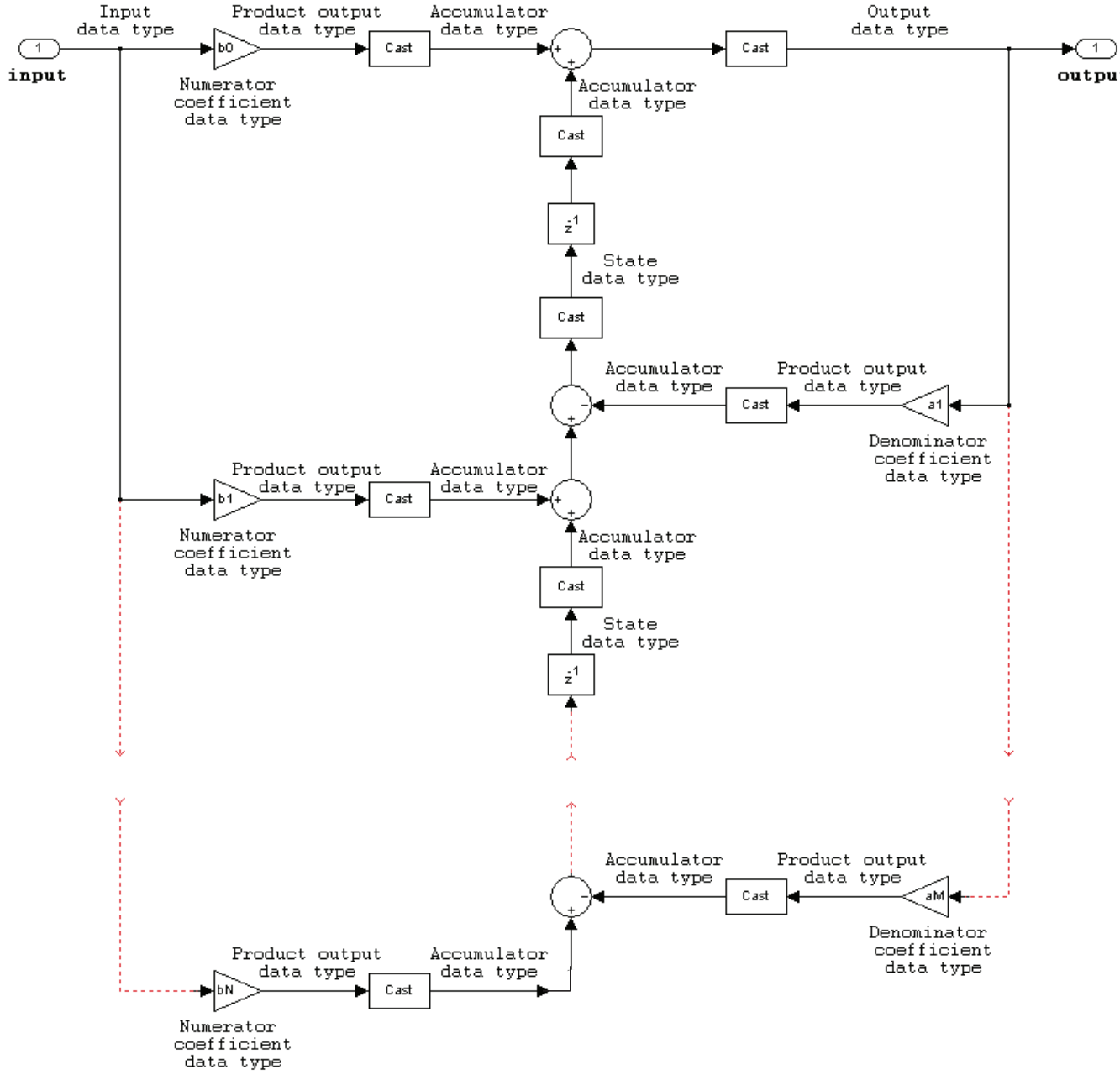
IIR direct form II transposed



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
Product output
data type
Accumulator
data type
Output
data type
1
input
b0
Numerator
coefficient
data type
Cast
Cast
Cast
1
output
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
Accumulator
data type
b1
Numerator
coefficient
data type
Cast
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
Accumulator
data type
bN
Numerator
coefficient
data type
Cast

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

`Allpole Filter`	DSP System Toolbox
`Digital Filter Design`	DSP System Toolbox
`Discrete FIR Filter`	Simulink
`Filter Realization Wizard`	DSP System Toolbox
dsp.IIRFilter	DSP System Toolbox
dsp.AllpoleFilter	DSP System Toolbox
`fdatool`	DSP System Toolbox
`fvtool`	Signal Processing Toolbox

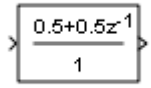
Introduced before R2006a

Discrete FIR Filter

Model FIR filters

Library

Discrete



Description

The Discrete FIR Filter block independently filters each channel of the input signal with the specified digital FIR filter. The block can implement static filters with fixed coefficients, as well as time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an independent channel (sample-based processing), or each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal those of the input, except when you specify a matrix of filter taps for the **Coefficients** parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

The outputs of this block numerically match the outputs of the DSP System Toolbox `Digital Filter Design` block 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`

Initial Condition	Description
	(`#_of_reflection_coeffs` for `Lattice MA`), and must have one column for each channel of the input signal.

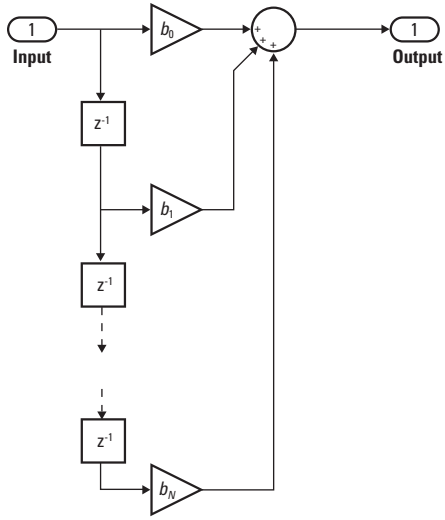
Data Type Support

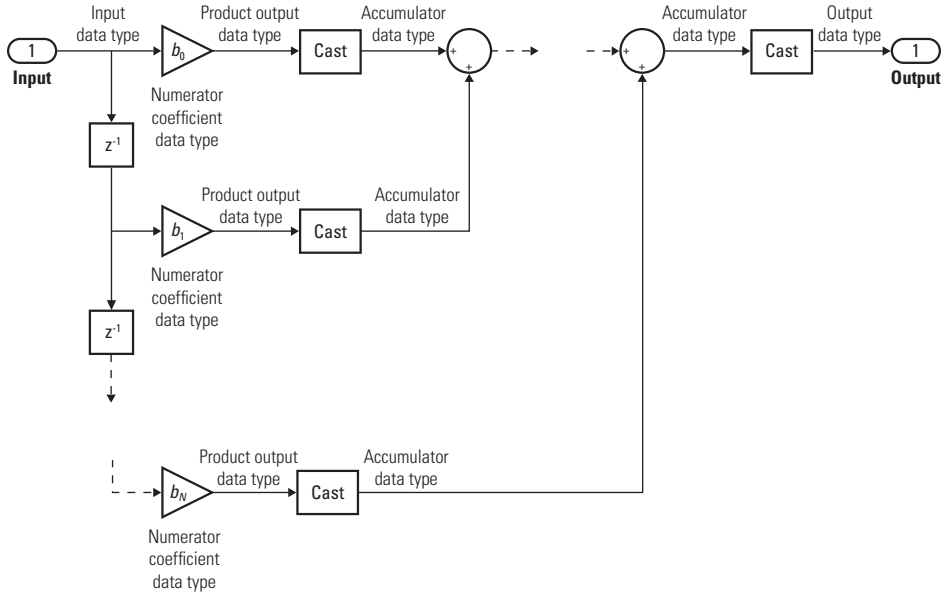
The `Discrete FIR Filter` block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.

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.

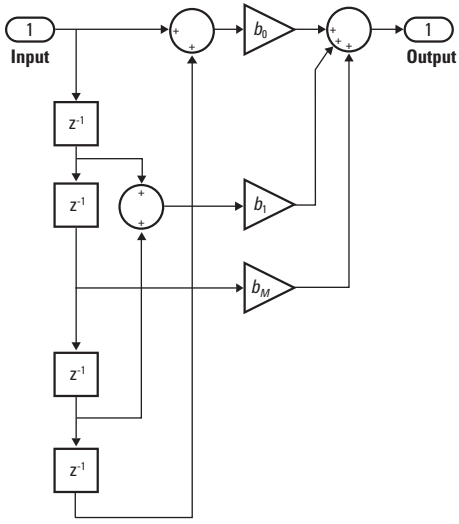



1
Input
Input data type
b_0
Product output data type
Cast
Accumulator data type
Numerator coefficient data type
z^{-1}
b_1
Product output data type
Cast
Accumulator data type
Numerator coefficient data type
z^{-1}
b_N
Product output data type
Cast
Accumulator data type
Numerator coefficient data type
Accumulator data type
Cast
Output data type
1
Output

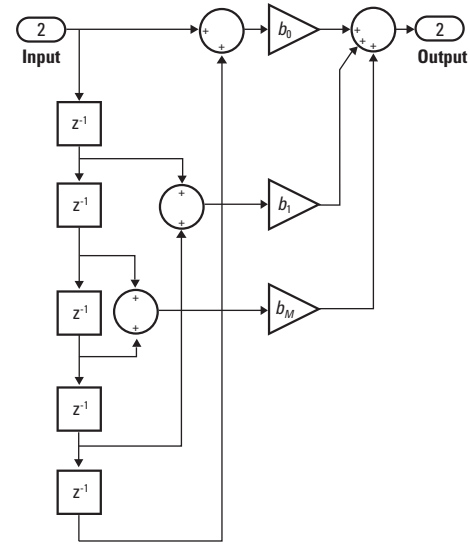
Direct Form Symmetric

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.

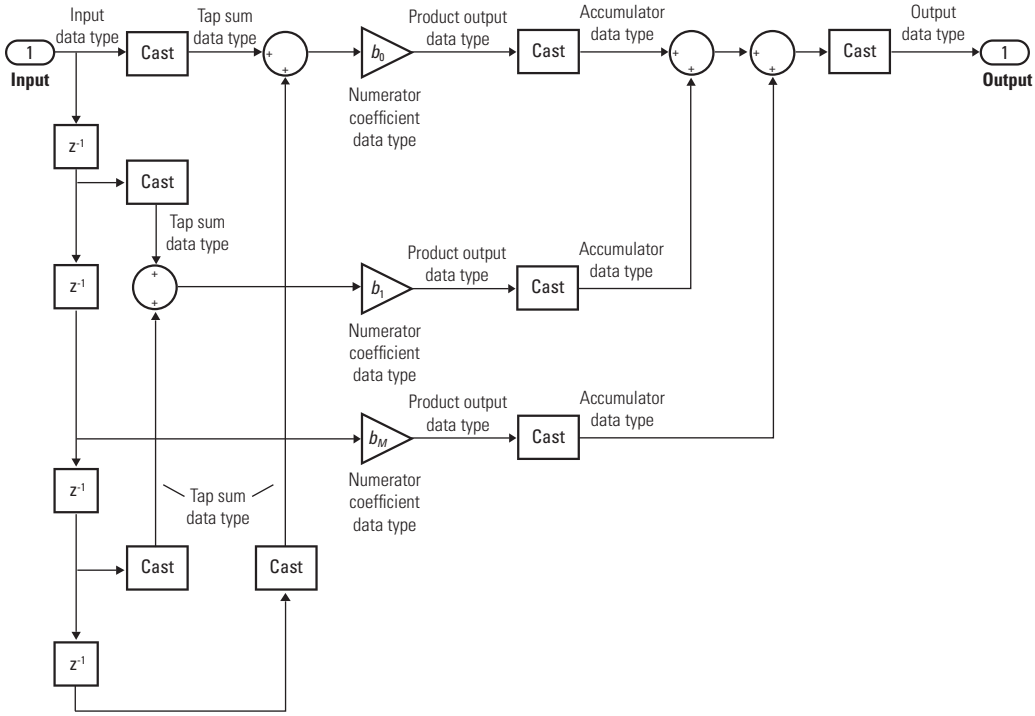
It is assumed that the filter coefficients are symmetric. The block only uses the first half of the coefficients for filtering.



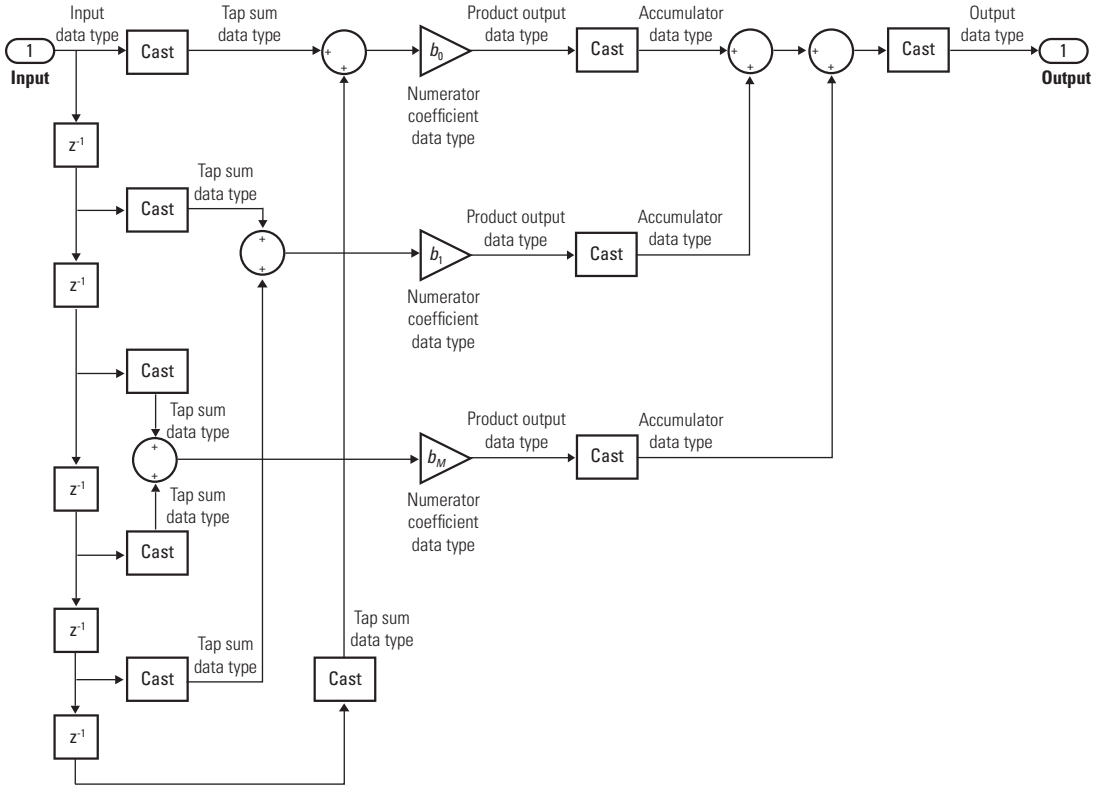
Even Order - Type I



Odd Order - Type II



Even Order - Type I

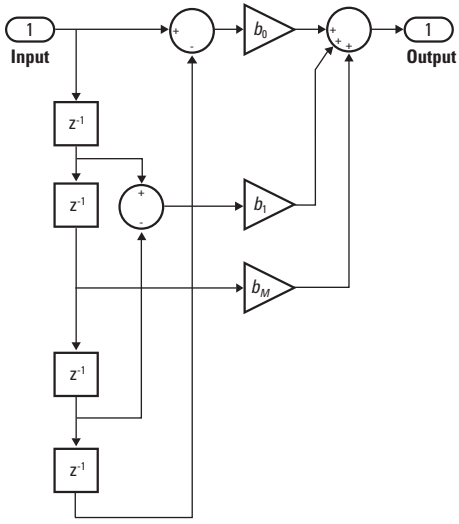


Odd Order - Type II

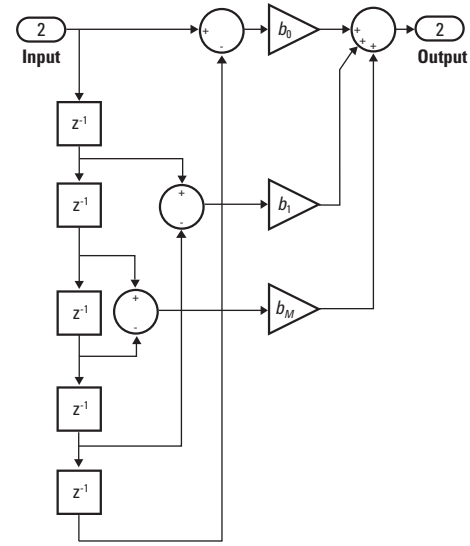
Direct Form Antisymmetric

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.

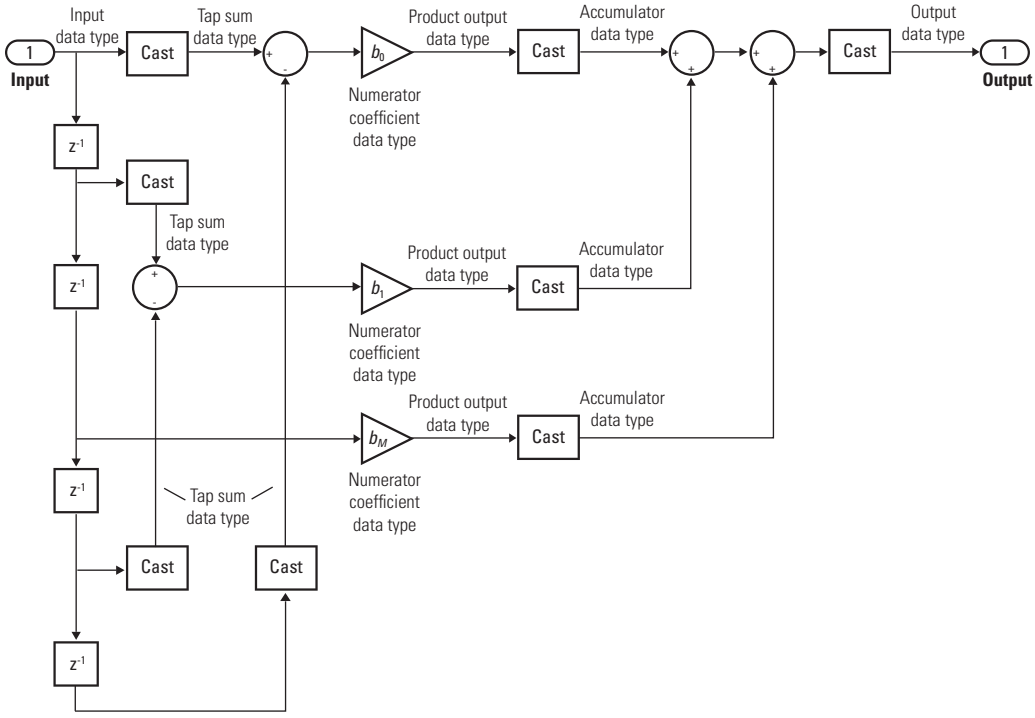
It is assumed that the filter coefficients are antisymmetric. The block only uses the first half of the coefficients for filtering.



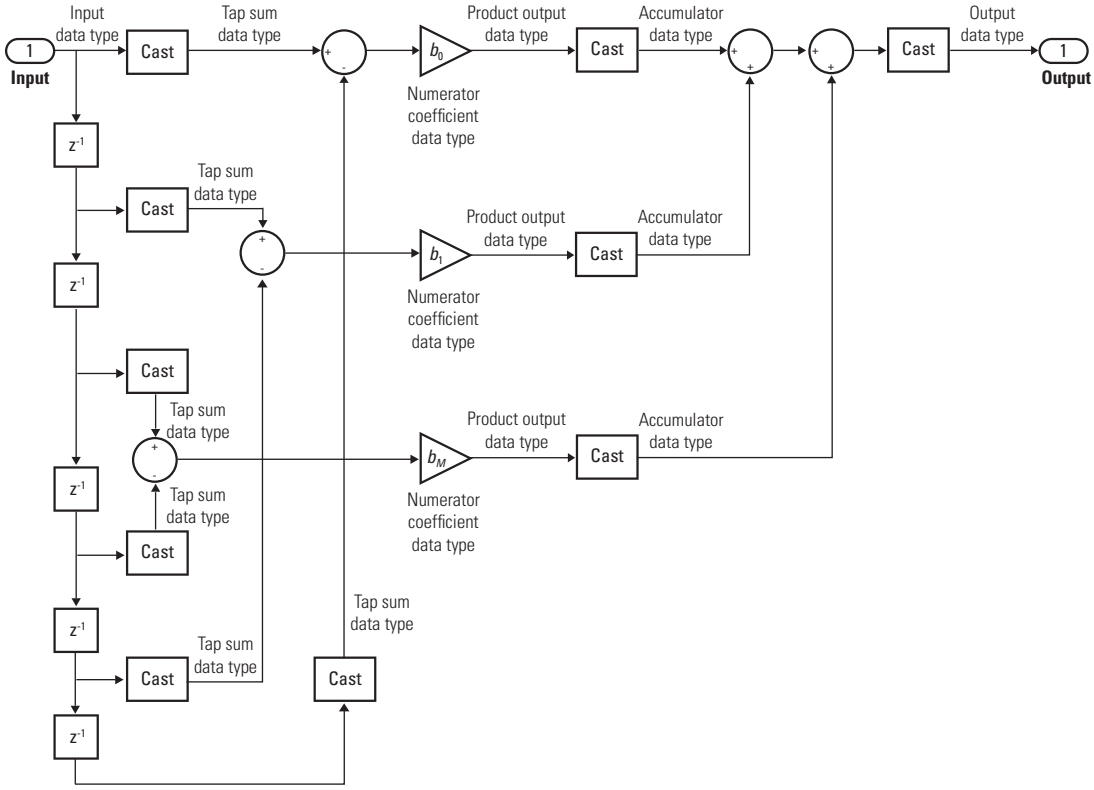
Even Order - Type III



Odd Order - Type IV



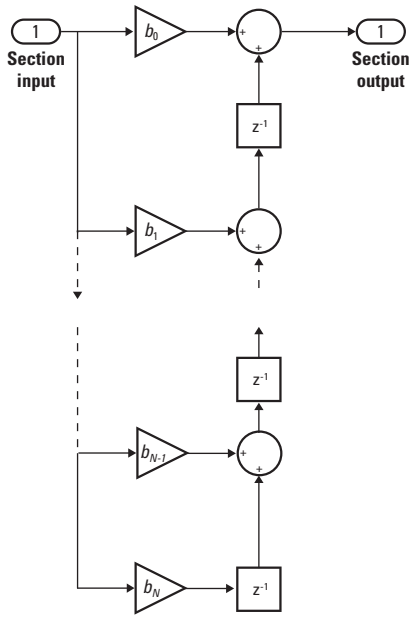
Even Order - Type III

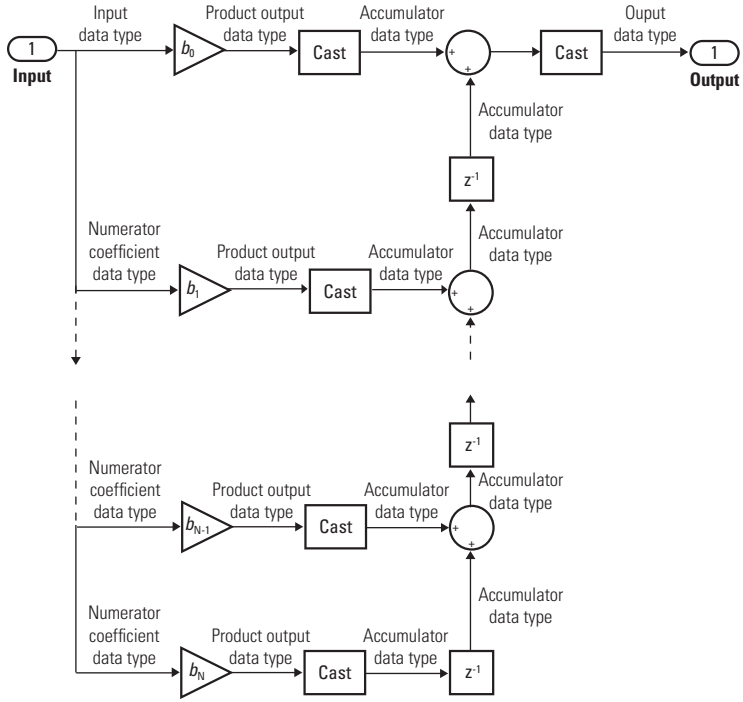


Odd Order - Type IV

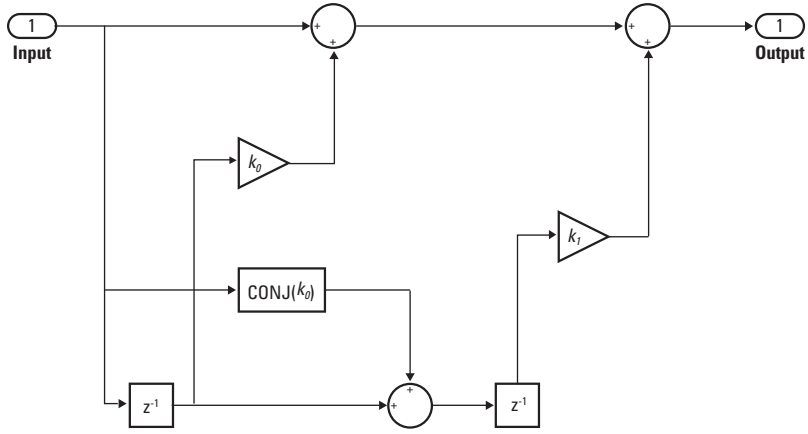
Direct Form Transposed

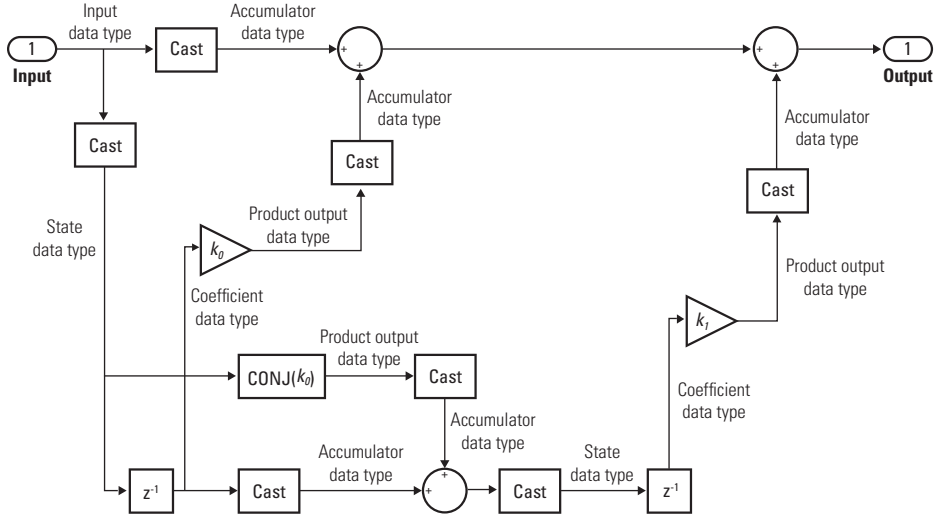
States are complex when either the inputs or the coefficients are complex.



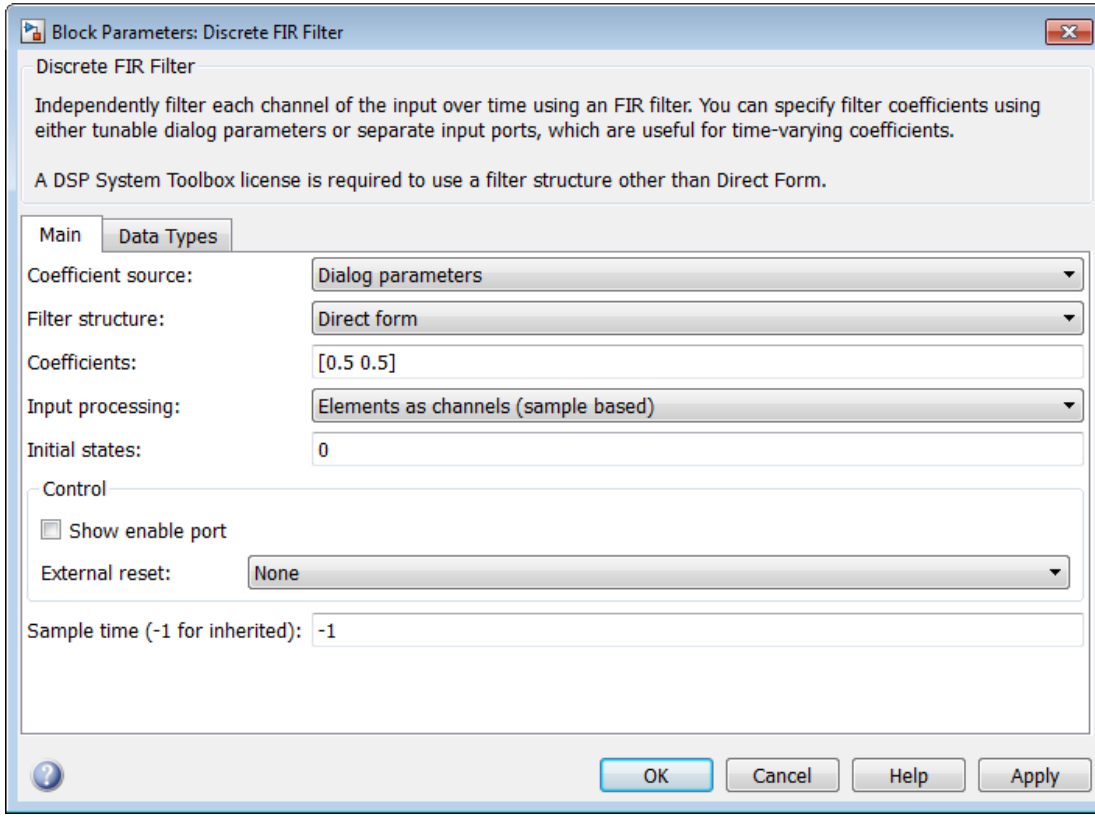

1
Input
Input data type
b_0
Product output data type
Cast
Accumulator data type
Cast
Ouput data type
1
Output
Accumulator data type
z^{-1}
Numerator coefficient data type
b_1
Product output data type
Cast
Accumulator data type
Accumulator data type
z^{-1}
Numerator coefficient data type
b_{N-1}
Product output data type
Cast
Accumulator data type
Accumulator data type
Numerator coefficient data type
b_N
Product output data type
Cast
Accumulator data type
z^{-1}

Lattice MA


1
Input
1
Output
k_0
k_1
CONJ(k_0)
z-1
z-1


1
Input
Input data type
Cast
Accumulator data type
1
Output
Cast
Accumulator data type
Cast
Accumulator data type
Cast
State data type
Product output data type
k_0
Coefficient data type
Product output data type
k_1
CONJ(k_0)
Cast
Accumulator data type
Coefficient data type
Accumulator data type
State data type
z-1
Cast
Cast
z-1

Parameters and Dialog Box



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 1-477.

Show enable port

Select to show an enable port for this block. This port can control execution of the block. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
`None`	No reset.
`Rising`	Reset on a rising edge.
`Falling`	Reset on a falling edge.
`Either`	Reset on either a rising or falling edge.

Reset Mode	Behavior
`Level`	Reset in either of these cases: • when the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
`Level hold`	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Sample time (-1 for inherited)

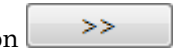
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in "How Simulink Works" in the *Simulink User's Guide*.

Tap sum

Specify the tap sum data type of a direct form symmetric or direct form antisymmetric filter, which is the data type the filter uses when it sums the inputs prior to multiplication by the coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is either `Direct form symmetric` or `Direct form antisymmetric`.

Click the **Show data type assistant** button 

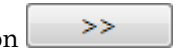
to display the **Data Type Assistant**, which helps you set the **Tap sum** parameter.

See "Specify Data Types Using Data Type Assistant" for more information.

Coefficients

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

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Coefficients** parameter.

See "Specify Data Types Using Data Type Assistant" for more information.

Coefficients minimum

Specify the minimum value that a filter coefficient should have. The default value is `[ ]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges")
- Automatic scaling of fixed-point data types

Coefficients maximum

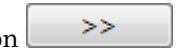
Specify the maximum value that a filter coefficient should have. The default value is `[ ]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges")
- Automatic scaling of fixed-point data types

Product output

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

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

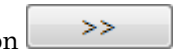
to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Accumulator

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

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

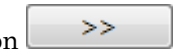
See “Specify Data Types Using Data Type Assistant” for more information.

State

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

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is `Lattice MA`.

Click the **Show data type assistant** button 

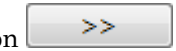
to display the **Data Type Assistant**, which helps you set the **State** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Output

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

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

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

Output minimum

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

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

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 “Lock 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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit

Action	Reasons for Taking This Action	What Happens for Overflows	Example
			integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes
Multidimensional Signals	No

Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

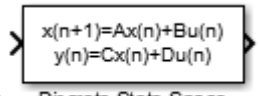
Introduced in R2008a

Discrete State-Space

Implement discrete state-space system

Library

Discrete



Description

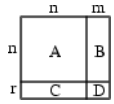
Block Behavior for Non-Empty Matrices

The Discrete State-Space block implements the system described by

$$x(n+1) = Ax(n) + Bu(n)$$
$$y(n) = Cx(n) + Du(n),$$

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.

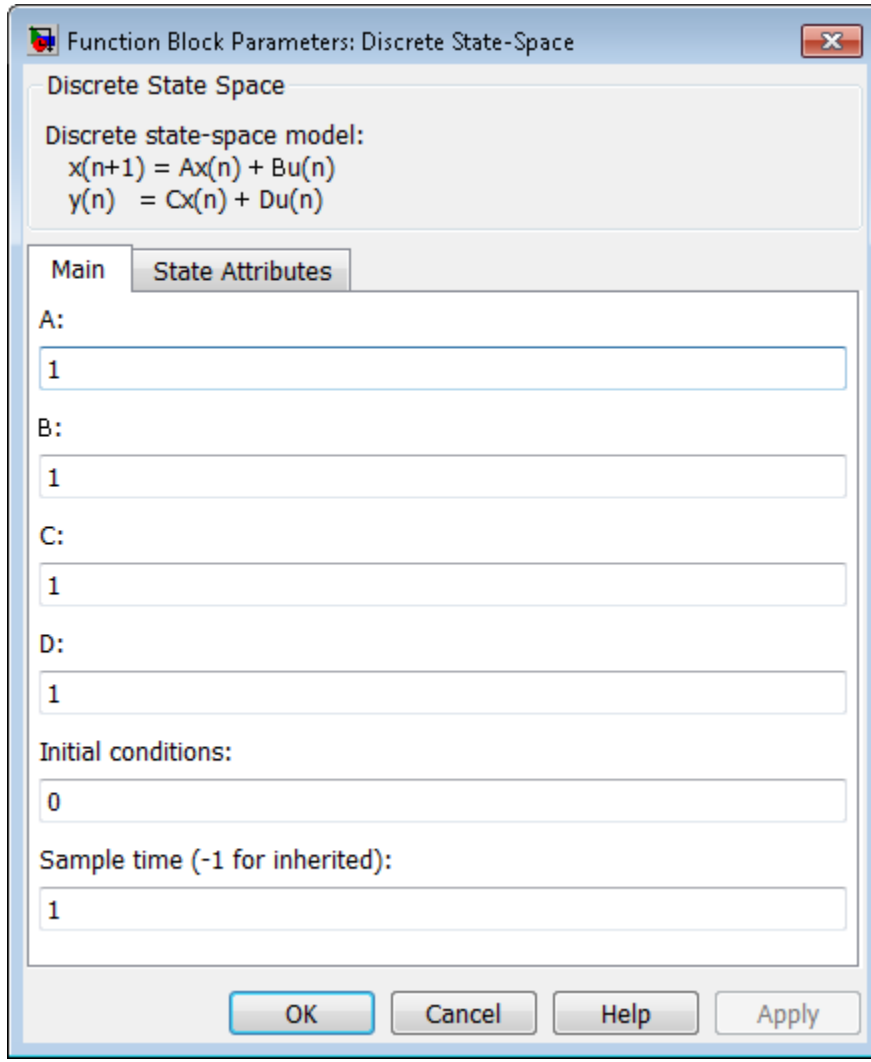
Block Behavior for Empty Matrices

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



A, B, C, D

Specify the matrix coefficients, as defined in the Description section.

Initial conditions

Specify the initial state vector. The default value is `0`. Simulink does not allow the initial states of this block to be `inf` or `NaN`.

Sample time (–1 for inherited)

Specify the time interval between samples. See "Specify Sample Time".

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see "Discrete Block State Naming in Generated Code" 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**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

For more information, see "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation.

Characteristics

Data Types	Double | Single

Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Only if $D \neq 0$
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

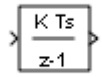
Introduced before R2006a

Discrete-Time Integrator

Perform discrete-time integration or accumulation of signal

Library

Discrete



Description

Capabilities of the Discrete-Time Integrator Block

You can use the Discrete-Time Integrator block in place of the `Integrator` block to create a purely discrete system. With the Discrete-Time Integrator block, you can:

- Define initial conditions on the block dialog box or as input to the block.
- Define an input gain (K) value.
- Output the block state.
- Define upper and lower limits on the integral.
- Reset the state depending on an additional reset input.

Output Equations

The block starts from the first time step, `n = 0`, with either initial output `y(0) = IC` or initial state `x(0) = IC`, depending on the **Initial condition setting** parameter value.

For a given step `n > 0` with simulation time `t(n)`, Simulink updates output `y(n)` as follows:

- Forward Euler method:

```
y(n) = y(n-1) + K*[t(n)-t(n-1)]*u(n-1)
```

- Backward Euler method:

  ```
  y(n) = y(n-1) + K*[t(n)-t(n-1)]*u(n)
  ```

- Trapezoidal method:

  ```
  y(n) = y(n-1) + K*[t(n)-t(n-1)]*[u(n)+u(n-1)]/2
  ```

Simulink automatically selects a state-space realization of these output equations depending on the block sample time, which can be explicit or triggered. When using explicit sample time, `t(n)-t(n-1)` reduces to the sample time `T` for all `n > 0`. For more information on these methods, see "Integration and Accumulation Methods" on page 1-504.

Integration and Accumulation Methods

The block can integrate or accumulate using the forward Euler, backward Euler, and trapezoidal methods. Assume that `u` is the input, `y` is the output, and `x` is the state. For a given step `n`, Simulink updates `y(n)` and `x(n+1)`. In integration mode, `T` is the block sample time (delta `T` in the case of triggered sample time). In accumulation mode, `T = 1`. The block sample time determines when the output is computed but not the output value. `K` is the gain value. Values clip according to upper or lower limits.

- Forward Euler method (default), also known as forward rectangular, or left-hand approximation

  For this method, the software approximates `1/s` as `T/(z-1)`. The expressions for the output of the block at step `n` are:

  ```
  x(n+1) = x(n) + K*T*u(n)
  y(n) = x(n)
  ```

  The block uses the following steps to compute the output:

  ```
  Step 0:   y(0)   = IC (clip if necessary)
     x(1)   = y(0) + K*T*u(0)

  Step 1:   y(1)   = x(1)
     x(2)   = x(1) + K*T*u(1)

  Step n:   y(n)   = x(n)
  ```

```
x(n+1) = x(n) + K*T*u(n) (clip if necessary)
```

Using this method, input port 1 does not have direct feedthrough.

- Backward Euler method, also known as backward rectangular or right-hand approximation

  For this method, the software approximates `1/s` as `T*z/(z-1)`. The resulting expression for the output of the block at step `n` is

  ```
  y(n) = y(n-1) + K*T*u(n).
  ```

  Let `x(n) = y((n)-1)`. The block uses these steps to compute the output.

  If the parameter **Initial condition setting** is set to `Output`:

  ```
  Step 0:   y(0)  = IC (clipped if necessary)
     x(1)   = y(0)
  ```

  If the parameter **Initial condition setting** is set to `State (most efficient)`:

  ```
  Step 0:   x(0)   = IC (clipped if necessary)
     x(1)   = y(0) = x(0) + K*T*u(0)

  Step 1:   y(1)   = x(1) + K*T*u(1)
     x(2)   = y(1)

  Step n:   y(n)   = x(n) + K*T*u(n)
     x(n+1) = y(n)
  ```

  Using this method, input port 1 has direct feedthrough.

- Trapezoidal method

  For this method, the software approximates `1/s` as `T/2*(z+1)/(z-1)`.

  When `T` is fixed (equal to the sampling period), the expressions to compute the output are:

  ```
  x(n) = y(n-1) + K*T/2 * u(n-1)
  y(n) = x(n) + K*T/2*u(n)
  ```

  If the **Initial condition setting** parameter is set to `Output`:

  ```
  Step 0:   y(0) = IC (clipped if necessary)
  ```

```
   x(1)   = y(0) + K*T/2*u(0)
```

If the **Initial condition setting** parameter is set to `State (most efficient)`:

```
Step 0:   x(0)   = IC (clipped if necessary)
   y(0)   = x(0) + K*T/2*u(0)
   x(1)   = y(0) + K*T/2*u(0)

Step 1:   y(1)   = x(1) + K*T/2*u(1)
   x(2)   = y(1) + K*T/2*u(1)

Step n:   y(n)   = x(n) + K*T/2*u(n)
   x(n+1) = y(n) + K*T/2*u(n)
```

Here, `x(n+1)` is the best estimate of the next output. It is not the same as the state, in that `x(n)` is not equal to `y(n)`.

If `T` is variable (for example, obtained from the triggering times), the block uses these steps to compute the output.

If the **Initial condition setting** parameter is set to `Output`:

```
Step 0:   y(0) = IC (clipped if necessary)
   x(1)   = y(0)
```

If the **Initial condition setting** parameter is set to `State (most efficient)`:

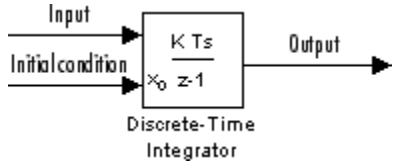
```
Step 0:   x(0)   = IC (clipped if necessary)
   x(1)   = y(0) = x(0) + K*T/2*u(0)

Step 1:   y(1)   = x(1) + T/2*(u(1) + u(0))
   x(2)   = y(1)

Step n:   y(n)   = x(n) + T/2*(u(n) + u(n-1))
   x(n+1) = y(n)
```

Using this method, input port 1 has direct feedthrough.

Define Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, set the **Initial condition source** parameter to `internal` and enter the value in the **Initial condition** text box.
- To provide the initial conditions from an external source, set the **Initial condition source** parameter to `external`. An additional input port appears on the block.

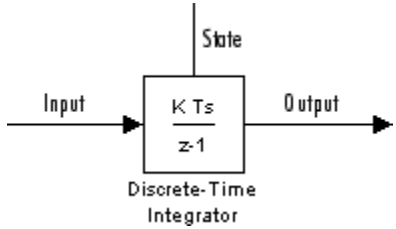


When to Use the State Port

Use the state port instead of the output port:

- When the output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example, see the `sldemo_bounce_two_integrators` model.
- When you want to pass the state from one conditionally executed subsystem to another, which can cause timing problems. For an example, see the `sldemo_clutch` model.

You can work around these problems by passing the state through the state port rather than the output port. Simulink generates the state at a slightly different time from the output, which protects your model from these problems. To output the block state, select the **Show state port** check box. The state port appears on the top of the block



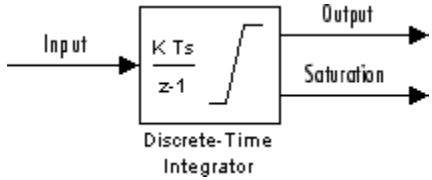
Limit the Integral

To keep the output within certain levels, select the **Limit output** check box and enter the limits in the corresponding text box. Doing so causes the block to function as a limited integrator. When the output reaches the limits, the integral action turns off to

prevent integral windup. During a simulation, you can change the limits but you cannot change whether the output is limited. The table shows how the block determines output.

Integral	Output
Less than or equal to the **Lower saturation limit** and the input is negative	Held at the **Lower saturation limit**
Between the **Lower saturation limit** and the **Upper saturation limit**	The integral
Greater than or equal to the **Upper saturation limit** and the input is positive	Held at the **Upper saturation limit**

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A new saturation port appears below the block output port:

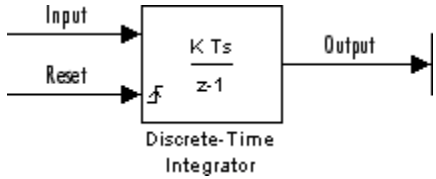


The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

Reset the State

The block can reset its state to the initial condition you specify, based on an external signal. To cause the block to reset its state, select one of the **External reset** parameter options. A reset port appears that indicates the reset trigger type:



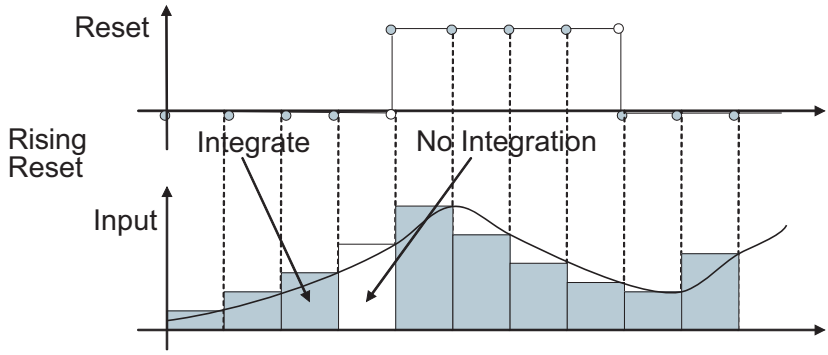
The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results.

To resolve this loop, feed the output of the block state port into the reset port instead. To access the block state, select the **Show state port** check box.

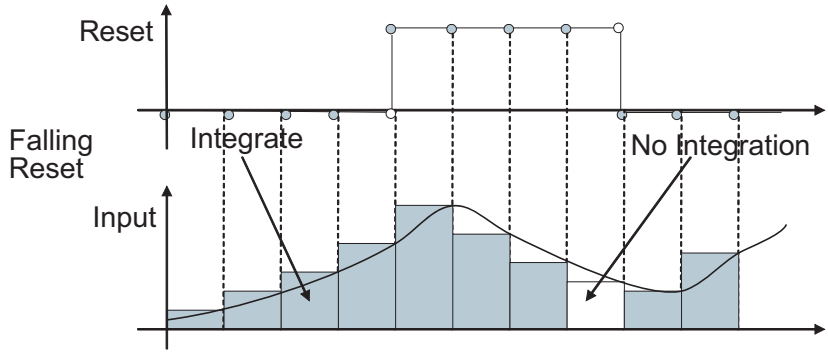
Reset Trigger Types

The **External reset** parameter lets you determine the attribute of the reset signal that triggers the reset. The trigger options include:

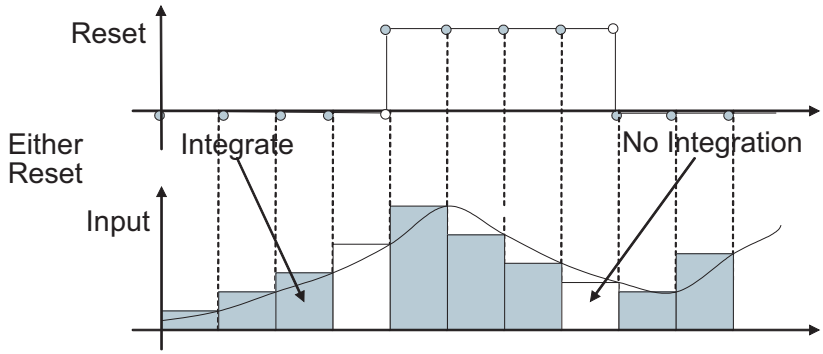
- `rising` – Resets the state when the reset signal has a rising edge. For example, this figure shows the effect that a rising reset trigger has on backward Euler integration.



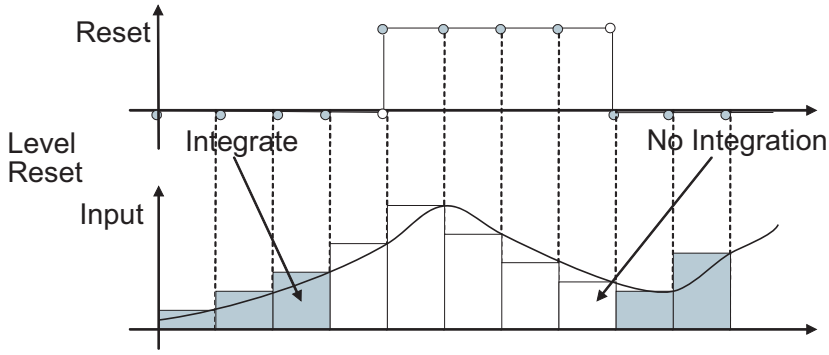
- `falling` – Resets the state when the reset signal has a falling edge. For example, this figure shows the effect that a falling reset trigger has on backward Euler integration.



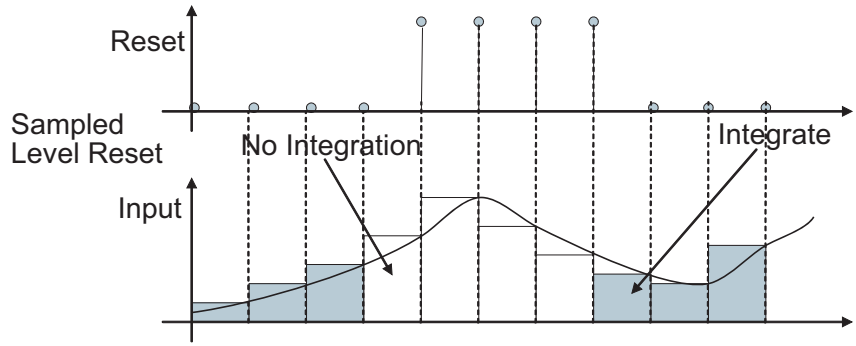
- `either` – Resets the state when the reset signal rises or falls. For example, the following figure shows the effect that an either reset trigger has on backward Euler integration.



- `level` – Resets and holds the output to the initial condition while the reset signal is nonzero. For example, this figure shows the effect that a level reset trigger has on backward Euler integration.



- `sampled level` – Resets the output to the initial condition when the reset signal is nonzero. For example, this figure shows the effect that a sampled level reset trigger has on backward Euler integration.



The `sampled level` reset option requires fewer computations, making it more efficient than the `level` reset option. However, the `sampled level` reset option can introduce a discontinuity when integration resumes.

Note: For the discrete-time integrator block, all trigger detections are based on signals with positive values. For example, a signal changing from -1 to 0 is not considered a rising edge, but a signal changing from 0 to 1 is.

Behavior in Simplified Initialization Mode

Simplified initialization mode is enabled when you set **Configuration Parameters > All Parameters > Underspecified initialization detection** to `Simplified`. If you use simplified initialization mode, the behavior of the Discrete-Time Integrator block differs from classic initialization mode. The new initialization behavior is more robust and provides more consistent behavior in these cases:

- In algebraic loops
- On enable and disable
- When comparing results using triggered sample time against explicit sample time, where the block is triggered at the same rate as the explicit sample time

Simplified initialization mode enables easier conversion from Continuous-Time Integrator blocks to Discrete-Time Integrator blocks, because the initial conditions have the same meaning for both blocks.

For more information on classic and simplified initialization modes, see “Underspecified initialization detection”.

Enable and Disable Behavior with Initial Condition Setting set to Output

When you use simplified initialization mode with **Initial condition setting** set to `Output`, the enable and disable behavior of the block is simplified as follows:

At disable time t_d:

$$y(t_d) = y(t_d-1)$$

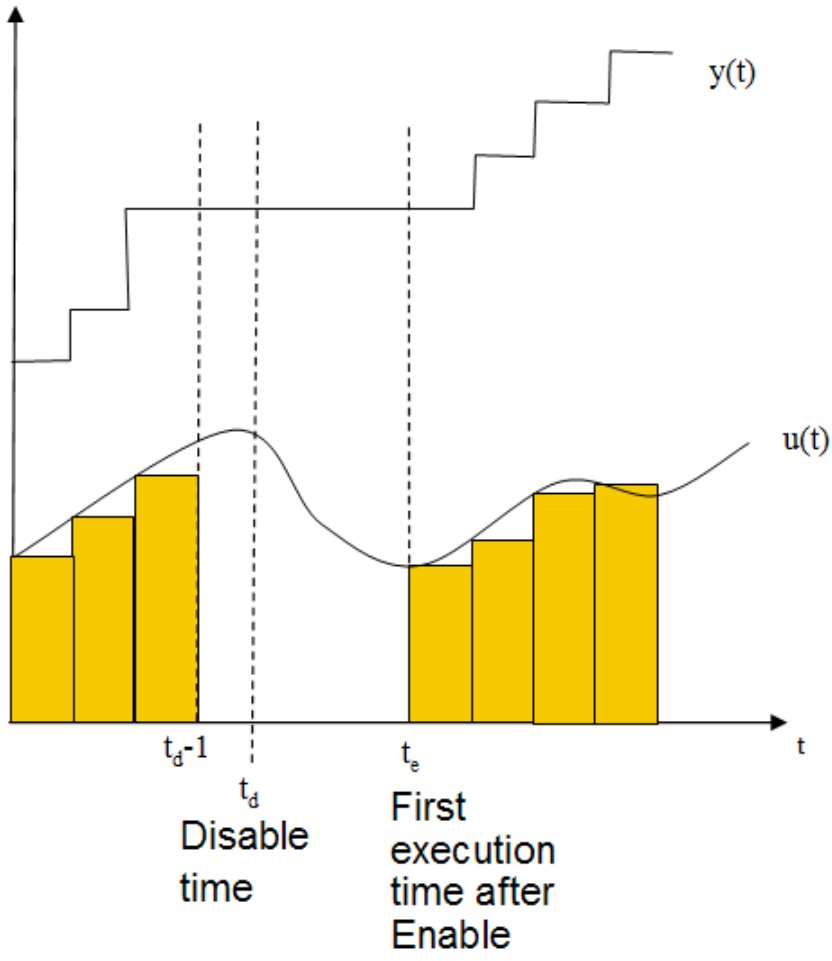
At enable time t_e:

- If the parent subsystem control port has **States when enabling** set to `reset`:

  $y(t_e) = IC.$
- If the parent subsystem control port has **States when enabling** set to `held`:

  $y(t_e) = y(t_d).$

  The following figure shows this condition.



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. Thus, if a Discrete-Time Integrator block, which needs elapsed time, is placed inside an Iterator Subsystem block, Simulink reports an error.

Triggered Subsystems and Function-Call Subsystems

Simulink does not support model simulation when all the following conditions are true:

- A Discrete-Time Integrator block is placed within a triggered subsystem or a function-call subsystem.
- The block's **Initial condition setting** parameter is set to `State (most efficient)`.
- Simplified initialization mode is enabled.

Behavior in an Enabled Subsystem Inside a Function-Call Subsystem

Suppose that you have a function-call subsystem that contains an enabled subsystem, which contains a Discrete-Time Integrator block. The following behavior applies.

Integrator Method	Sample Time Type of Function-Call Trigger Port	Value of `delta T` When Function-Call Subsystem Executes for the First Time After Enabled	Reason for Behavior
Forward Euler	Triggered	`t - tstart`	When the function-call subsystem executes for the first time, the integrator algorithm uses `tstart` as the previous simulation time.
Backward Euler and Trapezoidal	Triggered	`t - tprevious`	When the function-call subsystem executes for the first time, the integrator algorithm uses `tprevious` as the previous simulation time.
Forward Euler, Backward Euler, and Trapezoidal	Periodic	Sample time of the function-call generator	In periodic mode, the Discrete-Time Integrator block uses sample time of the

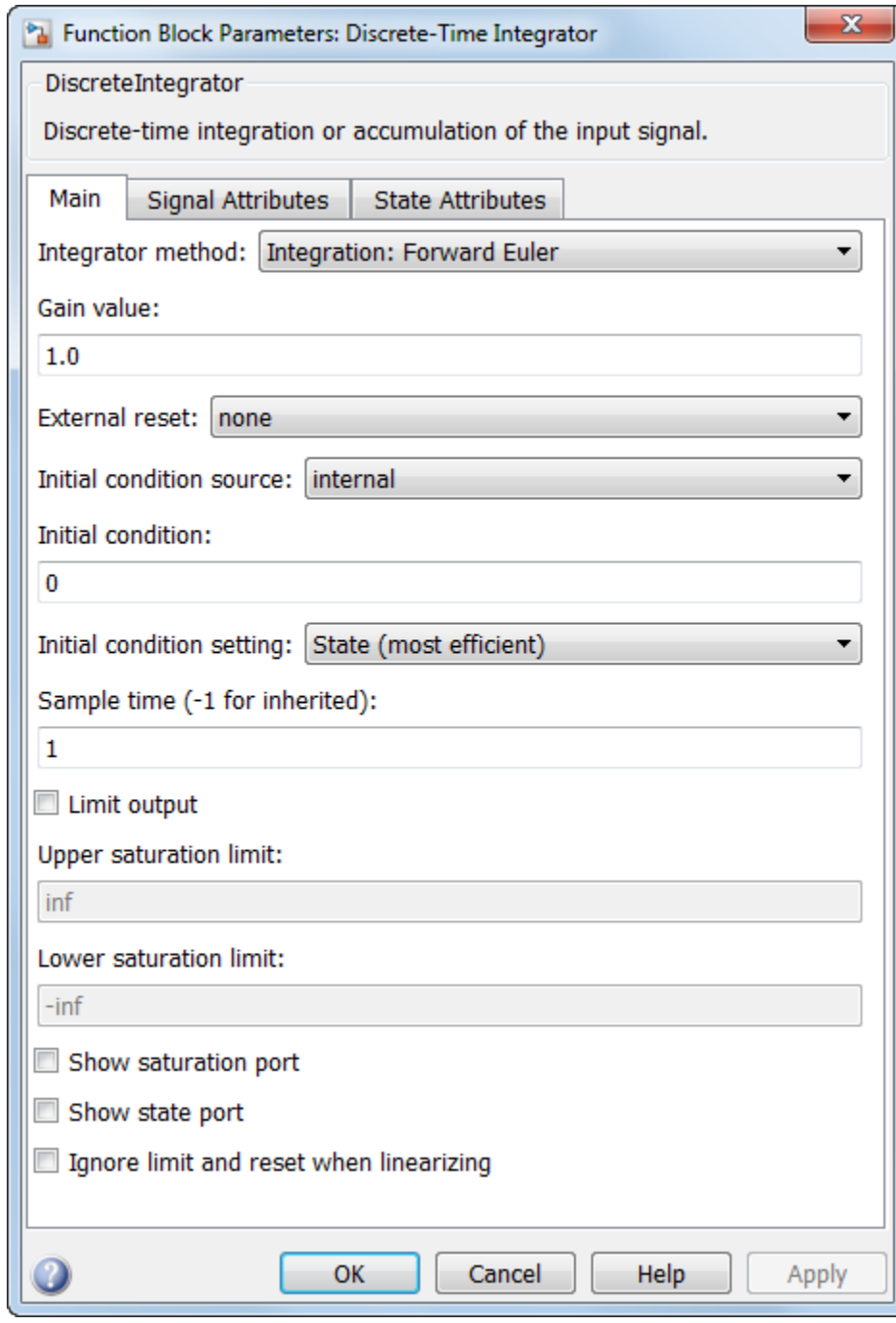
Integrator Method	Sample Time Type of Function-Call Trigger Port	Value of `delta T` When Function-Call Subsystem Executes for the First Time After Enabled	Reason for Behavior
			function-call generator for `delta T`.

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.


Function Block Parameters: Discrete-Time Integrator
DiscreteIntegrator
Discrete-time integration or accumulation of the input signal.
Main
Signal Attributes
State Attributes
Integrator method:
Integration: Forward Euler
Gain value:
1.0
External reset:
none
Initial condition source:
internal
Initial condition:
0
Initial condition setting:
State (most efficient)
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
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 "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation.

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 “Control Signal Data Types”.

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

Parameter: `IntegratorMethod`
Type: string
Value: `'Integration: Forward Euler' | 'Integration: Backward Euler' | 'Integration: Trapezoidal' | 'Accumulation: Forward Euler' | 'Accumulation: Backward Euler' | 'Accumulation: Trapezoidal'`
Default: `'Integration: Forward Euler'`

Gain value

Specify a scalar, vector, or matrix by which to multiply the integrator input. Each element of the gain must be a positive real number.

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. However, this parameter must be nontunable to realize this benefit. If you want to tune the input gain, set this parameter to 1.0 and use an external Gain block to specify the input gain.

Command-Line Information

Parameter: `gainval`
Type: string
Value: `'1.0'`
Default: `'1.0'`

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

Parameter: `ExternalReset`
Type: string
Value: `'none' | 'rising' | 'falling' | 'either' | 'level' | 'sampled level'`
Default: `'none'`

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

Parameter: `InitialConditionSource`
Type: string
Value: `'internal' | 'external'`
Default: `'internal'`

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

Parameter: `InitialCondition`
Type: scalar or vector
Value: `'0'`
Default: `'0'`

Initial condition setting

Specify whether to apply the **Initial condition** parameter to the block state or output. This initial condition is also used as the reset value. This parameter was named **Use initial condition as initial and reset value for** in Simulink before R2014a.

Settings

Default: `State (most efficient)`

`State (most efficient)`

Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and **Integrator method** is set to an integration method.

Set the following initial conditions:

```
x(0) = IC
```

At reset:

```
x(n) = IC
```

`Output`

Use this option when the block is in a triggered subsystem or a function-call subsystem and **Integrator method** is set to an integration method.

Set the following initial conditions:

```
y(0) = IC
```

At reset:

```
y(n) = IC
```

`Compatibility`

This option is present to provide backward compatibility. You cannot select this option for Discrete-Time Integrator blocks in Simulink models but you can select it for Discrete-Time Integrator blocks in a library. Use this option to maintain compatibility with Simulink models created before R2014a.

Prior to R2014a, the option `State (most efficient)` was known as `State only (most efficient)`. The option `Output` was known as `State and output`. The behavior of the block with the option `Compatibility` is as follows.

- If **Configuration Parameters** > **All Parameters** > **Underspecified initialization detection** is set to `Classic`, the **Initial condition setting** parameter behaves as `State (most efficient)`.
- If **Configuration Parameters** > **All Parameters** > **Underspecified initialization detection** is set to `Simplified`, the **Initial condition setting** parameter behaves as `Output`.

Command-Line Information

Parameter: `InitialConditionSetting`
Type: string
Value: `'State (most efficient)' | 'Output' | 'Compatibilty'`
Default: `'Output'`

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

Parameter: `SampleTime`
Type: string
Value: `'1'`
Default: `'1'`

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

Parameter: `LimitOutput`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `UpperSaturationLimit`
Type: scalar or vector
Value: `'inf'`
Default: `'inf'`

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

Parameter: `LowerSaturationLimit`
Type: scalar or vector
Value: `'-inf'`
Default: `'-inf'`

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

Parameter: `ShowSaturationPort`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `ShowStatePort`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Ignore limit and reset when linearizing

Cause Simulink linearization commands to treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

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

Parameter: `IgnoreLimit`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

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

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation.

Command-Line Information

Parameter: `StateIdentifier`
Type: string
Value: `' '`
Default: `' '`

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

Parameter: `StateMustResolveToSignalObject`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

Settings

`Simulink.Signal`

Use custom storage classes from the built-in package `Simulink`.

`Package.Class`

Use custom storage classes from the package that defines the class that you select.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Settings

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

State name enables this parameter.

Command-Line Information

Command-Line Information
Parameter: `StateStorageClass`
Type: string
Value: `'Auto' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'SimulinkGlobal' | 'Custom'`
Default: `'Auto'`

Code generation storage class

Select state storage class for code generation.

Settings

Default: `Auto`

`Auto`

> `Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

> Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".
>
> Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Settings

- **Default:** `' '` (empty string)
- `const`
- `volatile`

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Command-Line Information

Parameter Name: `RTWStateStorageTypeQualifier`
Value Type: string
Default: `' '` (empty string)

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the minimum to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges") 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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

Parameter: `OutDataTypeStr`
Type: string
Value: `'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'`
Default: `'Inherit: Inherit via internal rule'`

See Also

For more information, see “Control Signal 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

Parameter: `OutDataTypeStr`
Type: string
Value: `'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'`
Default: `'Inherit: Inherit via internal rule'`

See 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.

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.

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**

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.

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.

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

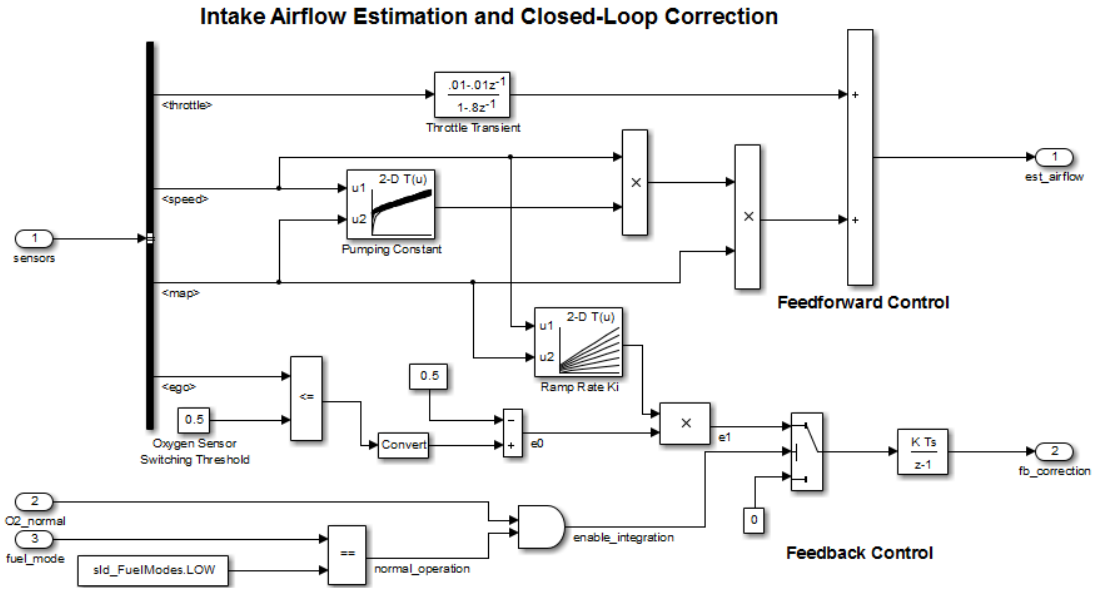
Selecting **Scaling** > `Slope and bias` enables this parameter.

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.



When the Switch block feeds a nonzero value into the Discrete-Time Integrator block, integration occurs. Otherwise, integration does not occur.

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, of the reset and external initial condition source ports. The input has direct feedthrough for every integration method except Forward Euler and accumulation Forward Euler.
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Integrator`

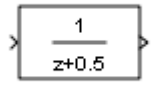
Introduced before R2006a

Discrete Transfer Fcn

Implement discrete transfer function

Library

Discrete



Description

The Discrete Transfer Fcn block implements the *z*-transform transfer function:

$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^m + num_1 z^{m-1} + ... + num_m}{den_0 z^n + den_1 z^{n-1} + ... + den_n}$$

where *m+1* and *n+1* are the number of numerator and denominator coefficients, respectively. *num* and *den* contain the coefficients of the numerator and denominator in descending powers of *z*. *num* can be a vector or matrix, *den* must be a vector, 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) • Row vector (1-by-*M*)	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*)
• Row vector (1-by-*N*)	*N*	• Scalar	• Scalar

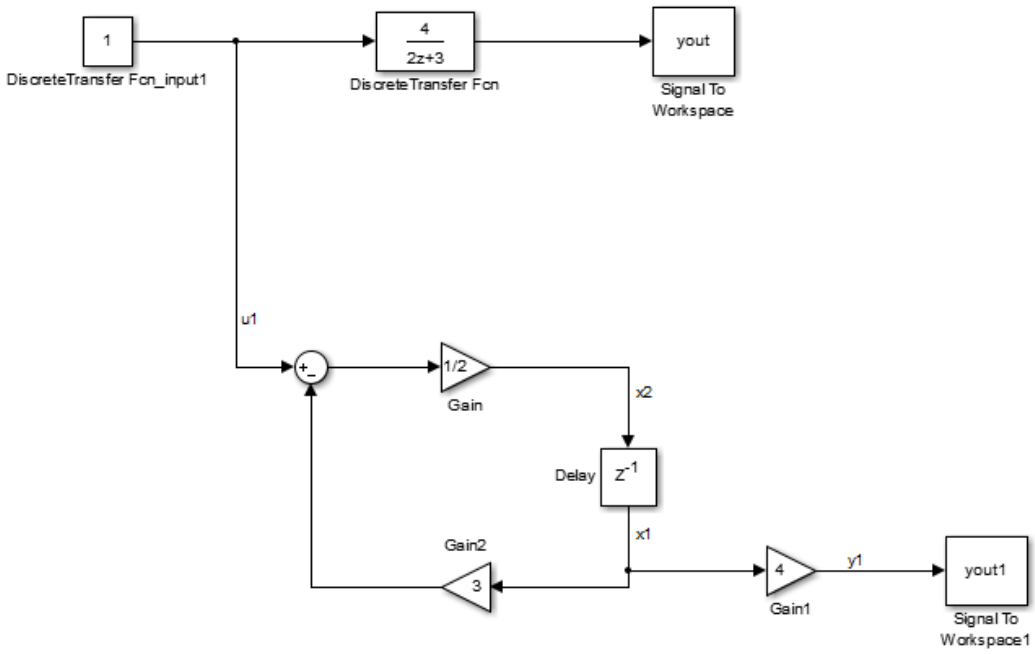
Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
• Column vector (*N*-by-1) • Unoriented vector (*N*)		• Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-*N*)	
• Matrix (*K*-by-*N*)	*K*×*N*	• Scalar • Column vector (*M*-by-1) • Row vector (1-by-*M*) • Matrix (*M*-by-(*K*×*N*))	• Scalar

When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. Enter `0` to initialize all states to zero. When the **Initial states** is a vector or a matrix, each vector or matrix element specifies a unique initial state for a corresponding delay element in a corresponding channel:

- The vector length must equal the number of delay elements in the filter, `M = max(number of zeros, number of poles)`.
- The matrix must have the same number of rows as the number of delay elements in the filter, `M = max(number of zeros, number of poles)`. The matrix must also have one column for each channel of the input signal.

The following example shows the relationship between the initial filter output and the initial input and state. Given an initial input u_1, the first output y_1 is related to the initial state $[x_1, x_2]$ and initial input by:

$$y_1 = 4x_1$$
$$x_2 = 1/2(u_1 - 3x_1)$$



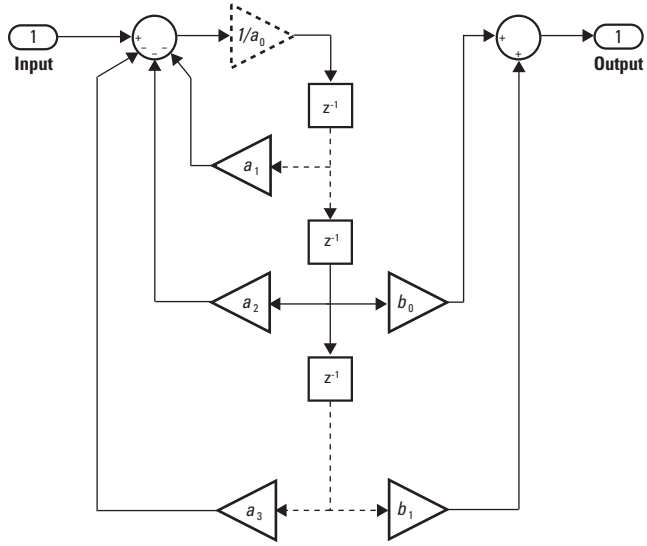
Data Type Support

The Discrete Transfer Function block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

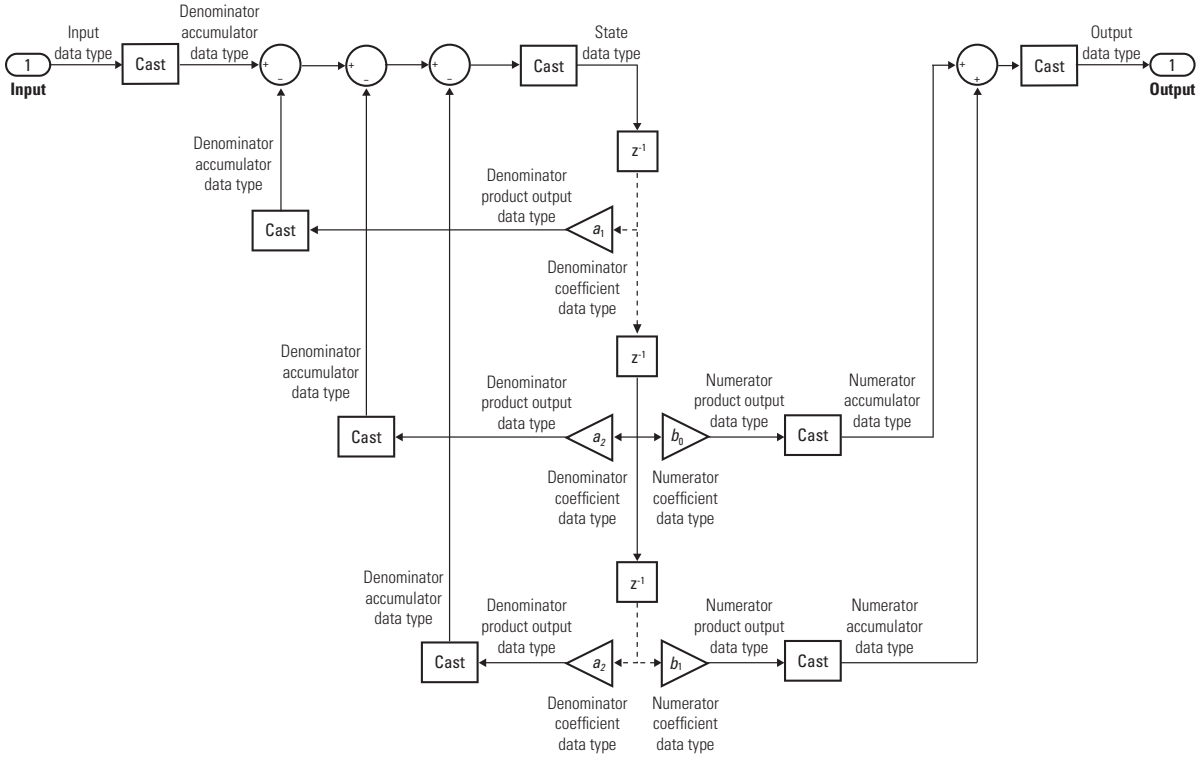
Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

States are complex when either the input or the coefficients are complex.

The following diagrams show the filter structure and the data types that the block uses for floating-point and fixed-point signals.

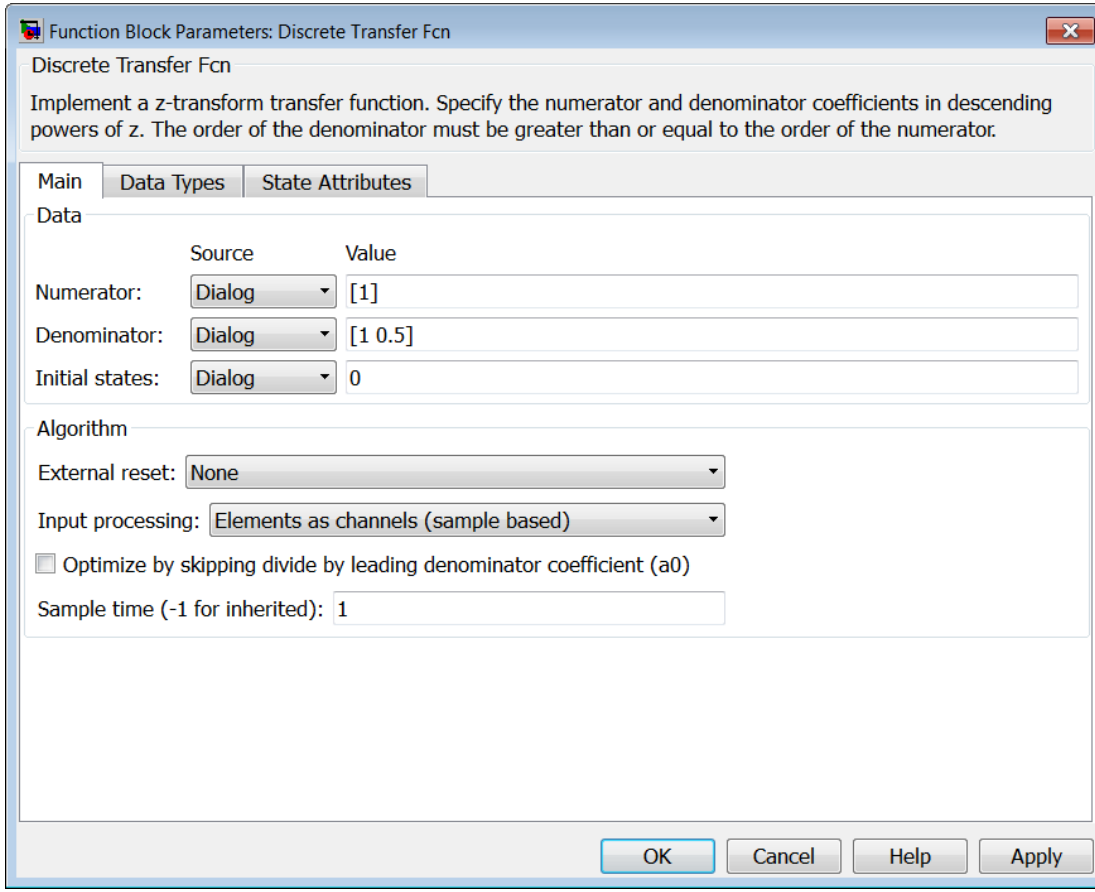


The block omits the dashed divide when you select the **Optimize by skipping divide by leading denominator coefficient (a0)** parameter.



Parameters and Dialog Box

The **Main** pane of the Discrete Transfer Fcn block dialog box appears as follows.



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 1-558.

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 the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
`Level hold`	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Input processing

Specify whether the block performs sample- or frame-based processing.

- `Elements as channels (sample based)` — Process each element of the input as an independent channel.
- `Columns as channels (frame based)` — Process each column of the input as an independent channel.

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

For more information, see "Sample- and Frame-Based Concepts" 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.

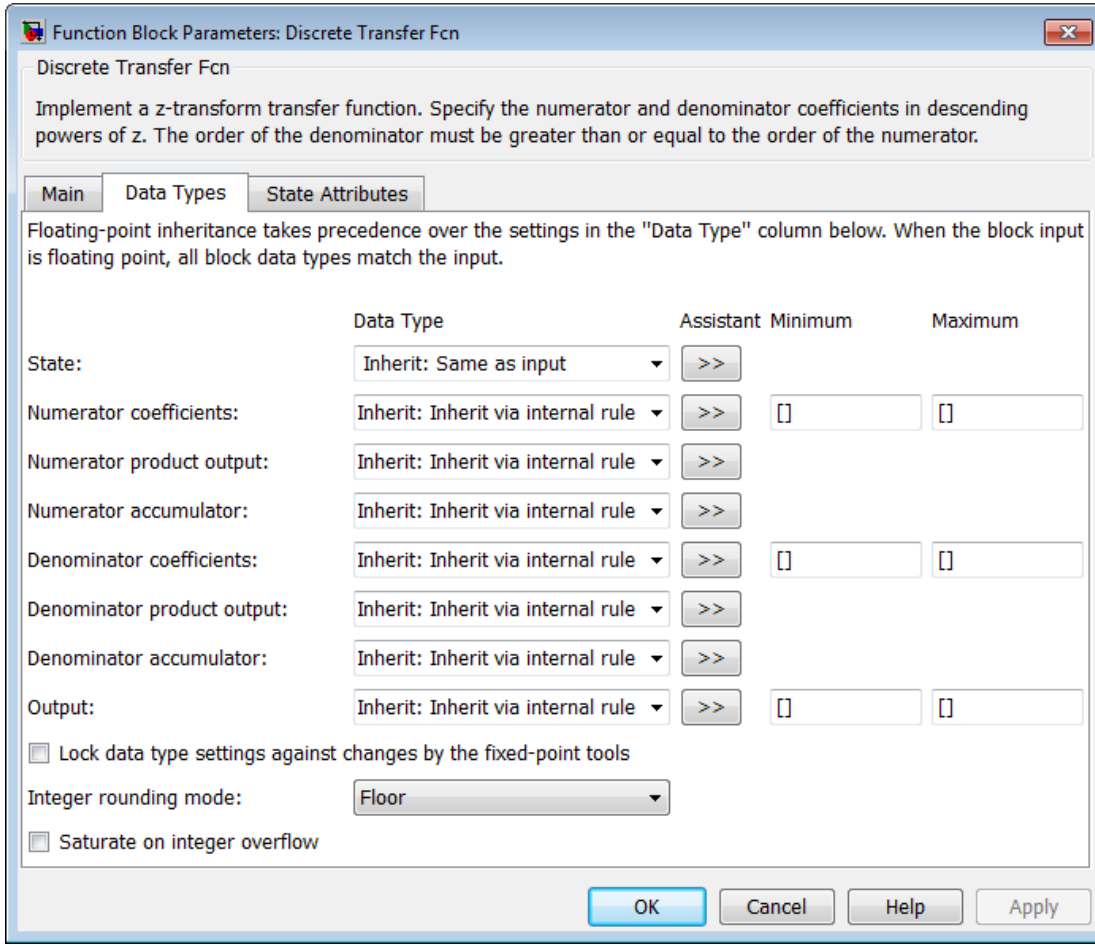
When you select this check box, the block does not perform a divide-by-a_0 either in simulation or in the generated code. An error occurs if a_0 is not equal to one.

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*.

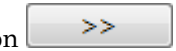
The **Data Types** pane of the Discrete Transfer Function block dialog box appears as follows.



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)`

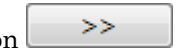
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **State** parameter.

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 “Control Block Parameter Value Ranges”)
- 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:

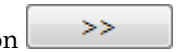
- Parameter range checking (see “Control Block Parameter Value Ranges”)
- Automatic scaling of fixed-point data types

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)`

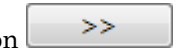
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator product output** parameter.

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)`

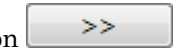
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator accumulator** parameter.

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 “Control Block Parameter Value Ranges”)
- Automatic scaling of fixed-point data types

Denominator coefficient maximum

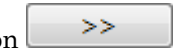
Specify the maximum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”)
- 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.

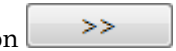
See “Specify Data Types Using Data Type Assistant” in theSimulink User's Guide for more information.

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)`

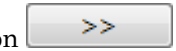
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Denominator accumulator** parameter.

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 "Control Signal Data Types" in the Simulink User's Guide for more information.

Output minimum

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

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

Output maximum

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

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

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

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Lock the Output Data Type Setting" in the Fixed-Point Designer documentation.

Integer rounding mode

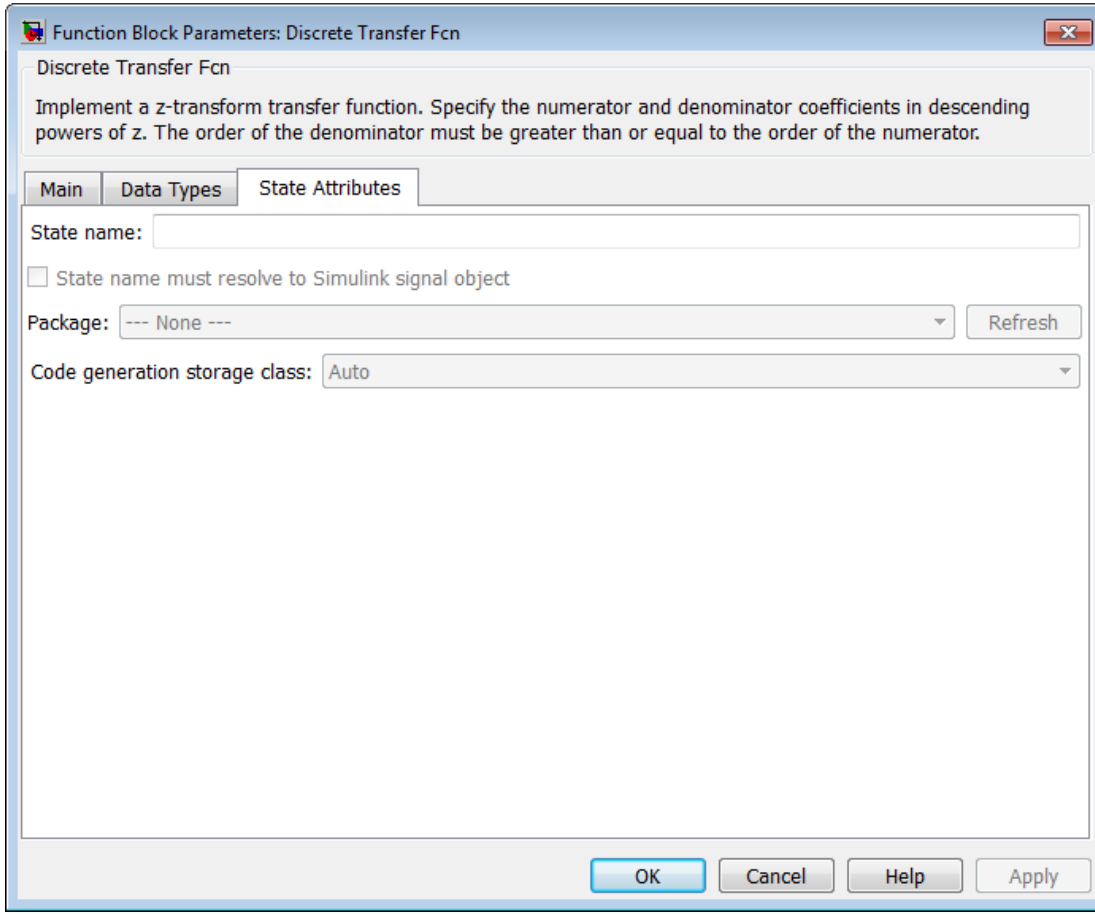
Specify the rounding mode for fixed-point operations. For more information, see "Rounding". in the Fixed-Point Designer documentation.

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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

The **State Attributes** pane of the Discrete Filter block dialog box appears as follows.



State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see "Discrete Block State Naming in Generated Code" 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**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name resolves
- Minimum and maximum values of the signal object

For more information, see "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation.

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Only when the leading numerator coefficient is not equal to zero and the numerator order equals the denominator order
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Code Generation	Yes

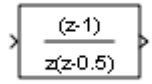
Introduced before R2006a

Discrete Zero-Pole

Model system defined by zeros and poles of discrete transfer function

Library

Discrete



Description

The Discrete Zero-Pole block models a discrete system defined by the zeros, poles, and gain of a *z*-domain transfer function. This block assumes that the transfer function has the following form:

$$H(z) = K\frac{Z(z)}{P(z)} = K\frac{(z-Z_1)(z-Z_2)...(z-Z_m)}{(z-P_1)(z-P_2)...(z-P_n)},$$

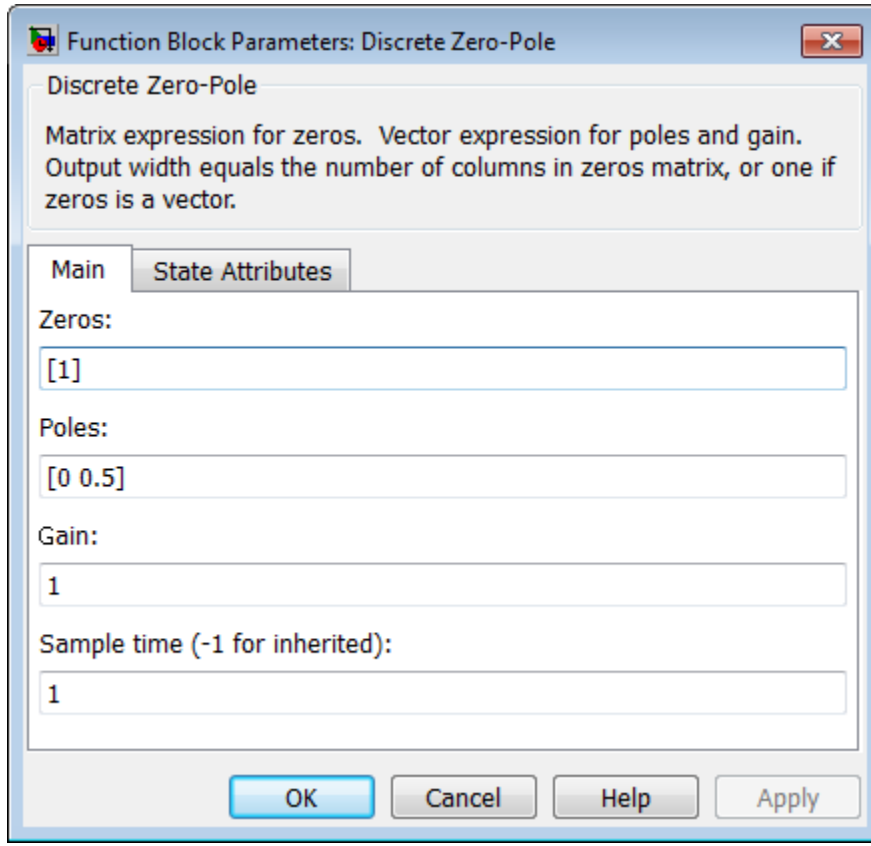
where Z represents the zeros vector, P the poles vector, and K the gain. The number of poles must be greater than or equal to the number of zeros ($n \geq m$). If the poles and zeros are complex, they must be complex conjugate pairs.

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



Zeros

Specify the matrix of zeros. The default is `[1]`.

Poles

Specify the vector of poles. The default is `[0 0.5]`.

Gain

Specify the gain. The default is `1`.

Sample time

Specify the time interval between samples. See Specifying Sample Time in the "How Simulink Works" chapter of the Simulink documentation.

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see "Discrete Block State Naming in Generated Code" 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**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

For more information, see "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation.

Characteristics

Data Types	Double | Single

Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, if the number of zeros and poles are equal
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

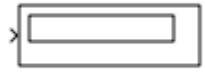
Introduced before R2006a

Display

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 "Simulink Enumerations" and "Define Simulink Enumerations"):

- The block displays enumerated values, not the values of underlying integers.
- Setting **Format** to any of the `Stored Integer` settings causes an error.

Display Abbreviations

The following abbreviations appear on the Display block to help you identify the format of the value.

When you see...	The value that appears is...
`(SI)`	The stored integer value **Note:** `(SI)` does not appear when the signal is of an integer data type.
`hex`	In hexadecimal format
`bin`	In binary format
`oct`	In octal format

Frequency of Data Display

The amount of data that appears and the time steps at which the data appears depend on the **Decimation** block parameter and the `SampleTime` property:

- The **Decimation** parameter enables you to display data at every *n*th sample, where *n* is the decimation factor. The default decimation, `1`, displays data at every time step.

  Note: The Display block updates its display at the initial time, even when the **Decimation** value is greater than one.

- The `SampleTime` property, which you can set with `set_param`, enables you to specify a sampling interval at which to display points. This property is useful when you are using a variable-step solver where the interval between time steps is not the same. The default sample time, `-1`, causes the block to ignore the sampling interval when determining the points to display.

Note: If the block inherits a sample time of `Inf`, the **Decimation** parameter is ignored.

Resizing Options

If the block input is an array, you can resize the block to show more than just the first element. You can resize the block vertically or horizontally, 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

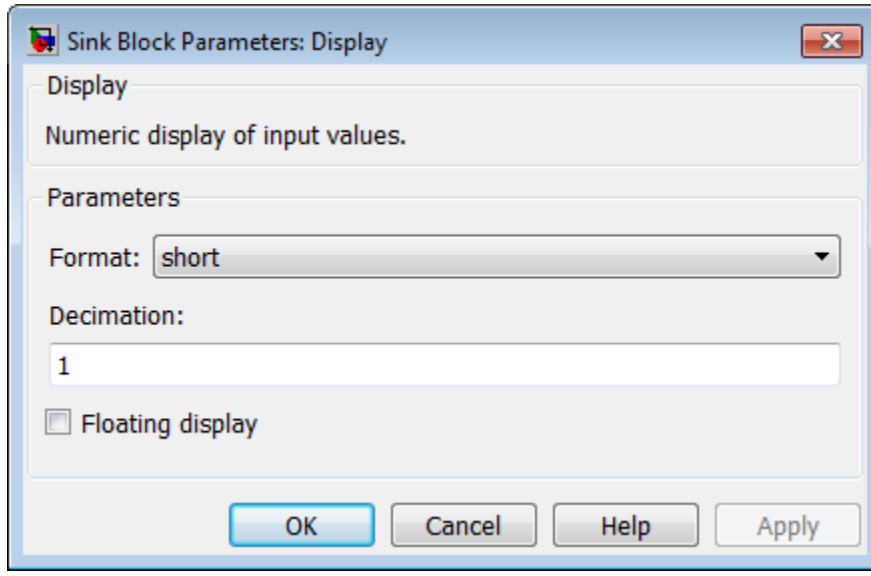
The Display block accepts real or complex signals of the following data types:

- Floating point
- Built-in integer

- Fixed point
- Boolean
- Enumerated

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

Parameters and Dialog Box



Format

Specify the format of the data that appears, as discussed in "Format Options" on page 1-581. The default is `short`.

Decimation

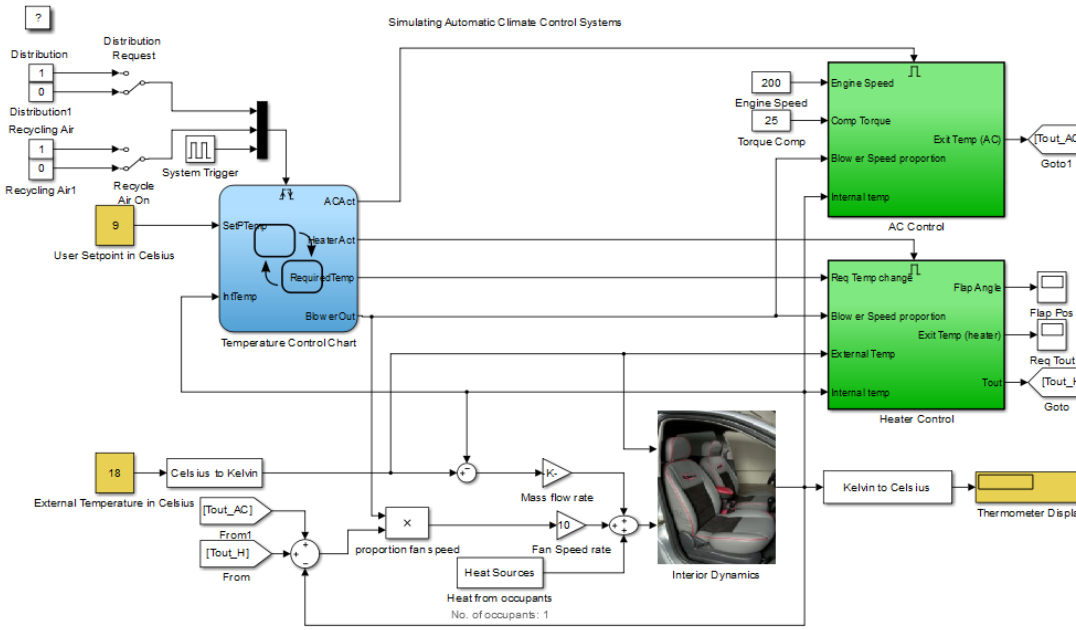
Specify how often to display data, as discussed in "Frequency of Data Display" on page 1-582. The default is `1`.

Floating display

Select to use the block as a floating display, as discussed in “Floating Display” on page 1-583.

Examples

The `sldemo_auto_climatecontrol` model shows how you can use the `Display` block.



Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Use `set_param` to specify the `SampleTime` property
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No

Code Generation	No

See Also

Scope

Introduced before R2006a

Divide

Divide one input by another

Library

Math Operations

Description

The Product and Product of Elements blocks are variants of the Divide block.

- For information on the Product block, see `Product`.
- For information on the Product of Elements block, see `Product of Elements`.

Supported Block Operations

The Divide block outputs the result of dividing its first input by its second. The inputs can be scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The Divide block is functionally a `Product` block that has two block parameter values preset:

- **Multiplication:** `Element-wise(.*)`
- **Number of Inputs:** `*/`

Setting non-default values for either of those parameters can change a Divide block to be functionally equivalent to a `Product` block or a `Product of Elements` block. See the documentation of those two blocks for more information.

Expected Differences Between Simulation and Code Generation

If any of the `Divide` block inputs contains a `NaN` or `inf` value, or if the block generates `NaN` or `inf` during execution, you might see different results when you compare the block

simulation results with the generated code. This difference is due to the nonfinite `NaN` or `inf` values. In such cases, inspect your model configuration and eliminate the conditions that produce `NaN` or `inf`.

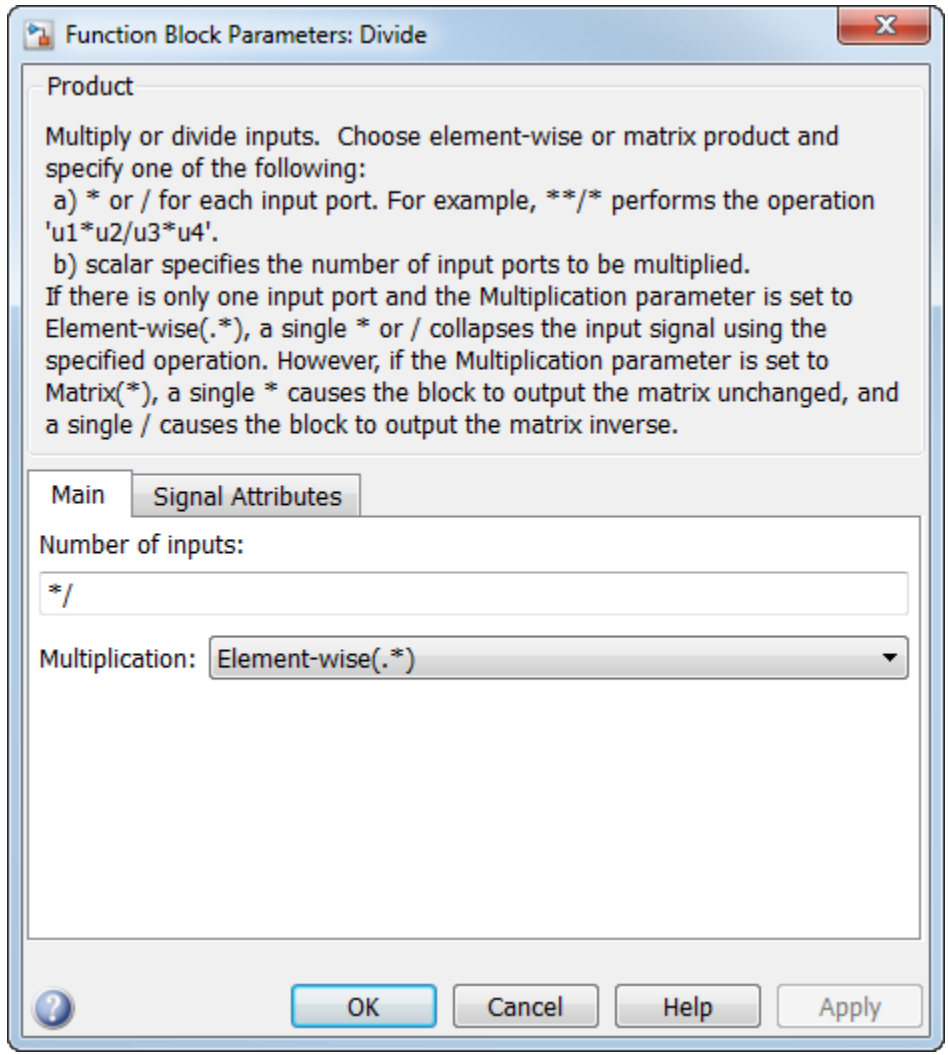
Code Optimizations

The Simulink Coder build process provides efficient code for matrix inverse and division operations. The following summary describes the benefits and when each benefit is available:

Benefit	Small matrices (2-by-2 to 5-by-5)	Medium matrices (6-by-6 to 20-by-20)	Large matrices (larger than 20-by-20)
Faster code execution time, compared to R2011a and earlier releases	Yes	No	Yes
Reduced ROM and RAM usage, compared to R2011a and earlier releases	Yes, for real values	Yes, for real values	Yes, for real values
Reuse of variables	Yes	Yes	Yes
Dead code elimination	Yes	Yes	Yes
Constant folding	Yes	Yes	Yes
Expression folding	Yes	Yes	Yes
Consistency with MATLAB Coder results	Yes	Yes	Yes

For blocks that have three or more inputs of different dimensions, the code might include an extra buffer to store temporary variables for intermediate results.

Parameters and Dialog Box



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 “Control Signal Data Types”.

Number of inputs

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

Settings

Default: * /

- **1 or * or /**

  Has one input. In element-wise mode, processes the input as described for the `Product of Elements` block. In matrix mode, if the parameter value is `1` or `*`, the block outputs the input value. If the value is `/`, the input must be a square matrix (including a scalar as a degenerate case) and the block outputs the matrix inverse. See “Element-wise Mode” on page 1-1459 and “Matrix Mode” on page 1-1460 for more information.

- **Integer value > 1**

  Has the number of inputs given by the integer value. The inputs are multiplied together in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1459 and “Matrix Mode” on page 1-1460 for more information.

- **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 1-1459 and “Matrix Mode” on page 1-1460 for more information.

Dependency

Setting **Number of inputs** to `*` and selecting `Element-wise(.*)` for **Multiplication** enables the parameter **Multiply over**.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 6-101 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 6-101 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 6-101 for the command-line information.

Sample time

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

Require all inputs to have the same data type

Require that all inputs have the same data type.

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 6-101 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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the minimum to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges") 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges") 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 6-101 for the command-line information.

See Also

For more information, see "Control Signal 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 6-101 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.

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.

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**

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.

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.

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

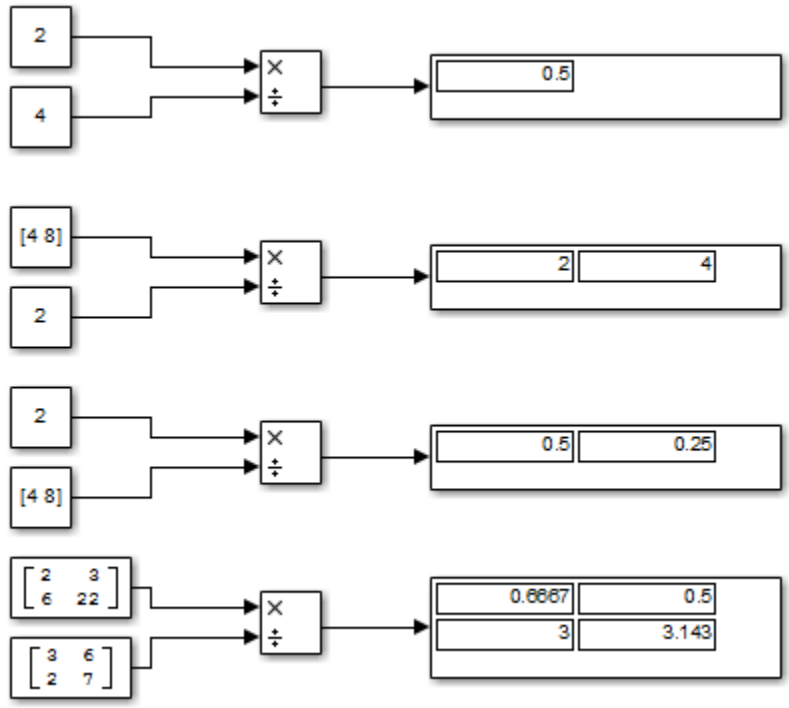
Selecting **Scaling** > `Slope and bias` enables this parameter.

See Also

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

Examples

The following examples show the output of the Divide block for some typical inputs using default block parameter values.



Introduced before R2006a

DocBlock

Create text that documents model and save text with model

Library

Model-Wide Utilities

Description

The DocBlock allows you to create and edit text that documents a model, and save that text with the model. Double-clicking an instance of the block creates a temporary file containing the text associated with this block and opens the file in an editor. Use the editor to modify the text and save the file. Simulink software stores the contents of the saved file in the model file.

The DocBlock supports HTML, Rich Text Format (RTF), and ASCII text document types. The default editors for these different document types are

- HTML — Microsoft® Word (if available). Otherwise, the DocBlock opens HTML documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- RTF — Microsoft Word (if available). Otherwise, the DocBlock opens RTF documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- Text — The DocBlock opens text documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.

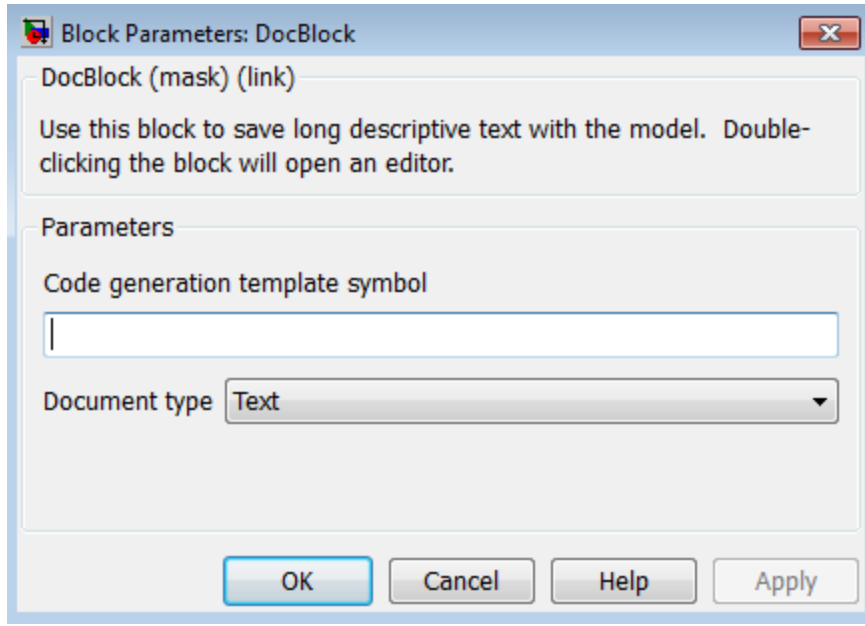
Use the `docblock` command to change the default editors.

Data Type Support

Not applicable.

Parameters 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.



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

Data Types	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Dot Product

Generate dot product of two vectors

Library

Math Operations

Description

The Dot Product block generates the dot product of the vectors at its inputs. The scalar output, `y`, is equal to the MATLAB operation

```
y = 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

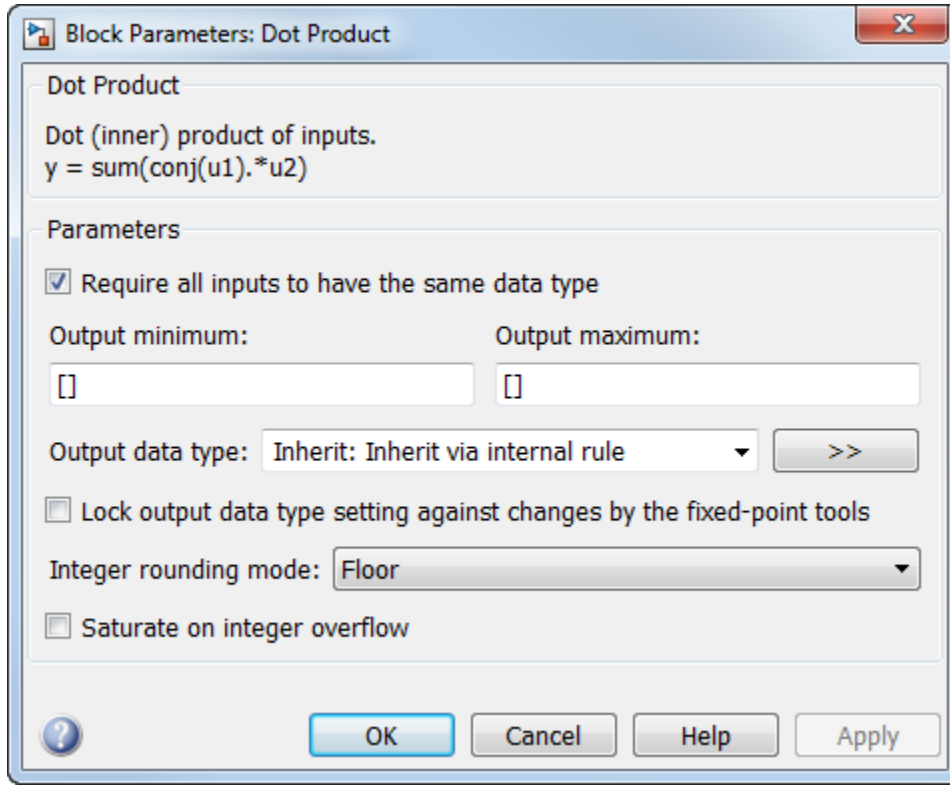
To perform element-by-element multiplication without summing, use the `Product` block.

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



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

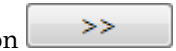
Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- 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 "Control Signal Data Types" for more information.

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

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

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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

Sample time

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

Characteristics

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

See Also

`Product`

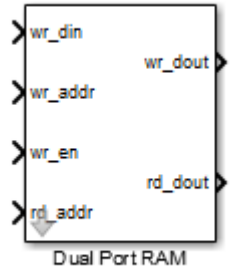
Introduced before R2006a

Dual Port RAM

Dual port RAM with two output ports

Library

HDL Coder / HDL Operations



Description

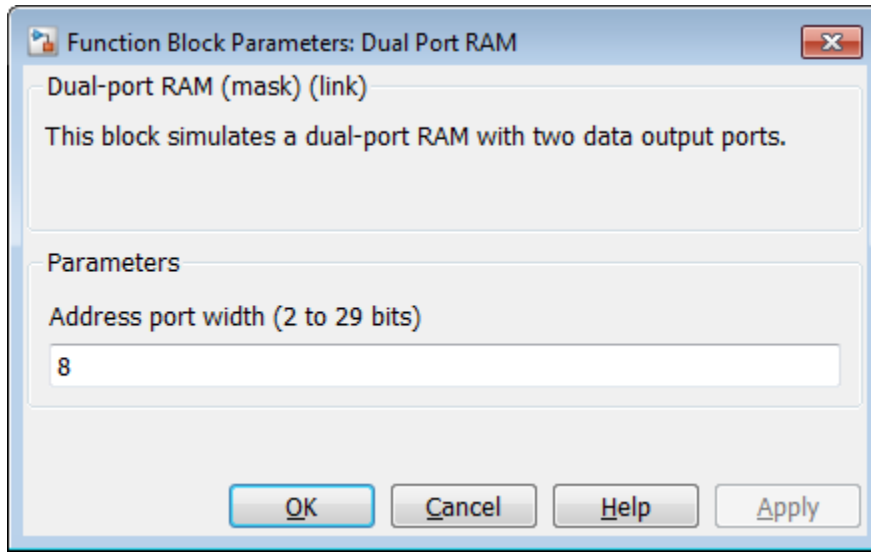
The `Dual Port RAM` block models a RAM that supports simultaneous read and write operations, and has both a read data output port and write data output port. You can use this block to generate HDL code that maps to RAM in most FPGAs.

If you do not need to use the write output data, `wr_dout`, you can achieve better RAM inference with synthesis tools by using the `Simple Dual Port RAM` block.

Read-During-Write Behavior

During a write, new data appears at the output of the write port (`wr_dout`) of the Dual Port RAM block. If a read operation occurs simultaneously at the same address as a write operation, old data appears at the read output port (`rd_dout`).

Dialog Box and Parameters



Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

Ports

The block has the following ports:

`wr_din`

Write data input. The data can be any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`wr_addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of `0`

`wr_en`

Write enable.

Data type: Boolean

`rd_addr`

Read address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of `0`

`wr_dout`

Output data from write address, `wr_addr`.

`rd_dout`

Output data from read address, `rd_addr`.

See Also

`Dual Rate Dual Port RAM` | `Simple Dual Port RAM` | `Single Port RAM`

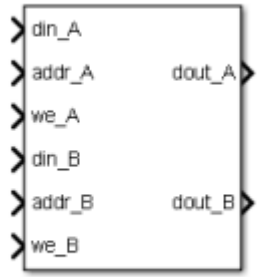
Introduced in R2014a

Dual Rate Dual Port RAM

Dual Port RAM that supports two rates

Library

HDL Coder / HDL Operations



Description

The `Dual Rate Dual Port RAM` block models a RAM that supports simultaneous read and write operations to different addresses at two clock rates. Port A of the RAM can run at one rate, and port B can run at a different rate.

In high-performance hardware applications, you can use this block to access the RAM twice per clock cycle. If you generate HDL code, this block maps to a dual-clock dual-port RAM in most FPGAs.

Simultaneous Access

You can access different addresses from ports A and B simultaneously. You can also read the same address from ports A and B simultaneously.

However, do not access an address from one RAM port while it is being written from the other RAM port. During simulation, if you access an address from one RAM port at the

same time as you write that address from the other RAM port, the software reports an error.

Read-During-Write Behavior

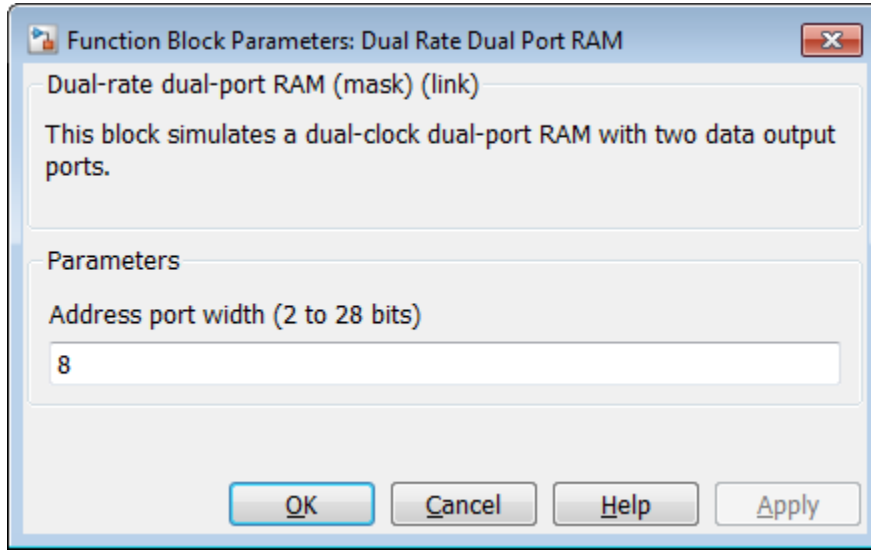
The RAM has write-first behavior. When you write to the RAM, the new write data is immediately available at the output port.

HDL Code Generation

For simulation results that match the generated HDL code, in the Configuration Parameters dialog box, in the Solver pane, **Tasking mode for periodic sample times** must be `SingleTasking`.

If you simulate this block using `MultiTasking` mode, the output data can update in the same cycle, but in the generated HDL code, the output data is updated one cycle later.

Dialog Box and Parameters



Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 28. The default value is 8.

Ports

The block has the following ports:

`din_A`

Write data input for RAM port A. The data can be any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`addr_A`

Write address for RAM port A.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of `0`

`we_A`

Write enable for RAM port A. Set `we_A` to `true` for a write operation, or `false` for a read operation.

Data type: Boolean

`din_B`

Write data input for RAM port B. The data can be of any width, and inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`addr_B`

Write address for RAM port B.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of `0`

`we_B`

Write enable for RAM port B. Set `we_B` to `true` for a write operation, or `false` for a read operation.

Data type: Boolean

`dout_A`

Output data from RAM port A address, `addr_A`.

`dout_B`

Output data from RAM port B address, `addr_B`.

See Also

`Dual Port RAM` | `HDL FIFO` | `Simple Dual Port RAM` | `Single Port RAM`

Introduced in R2014a

Enable

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.

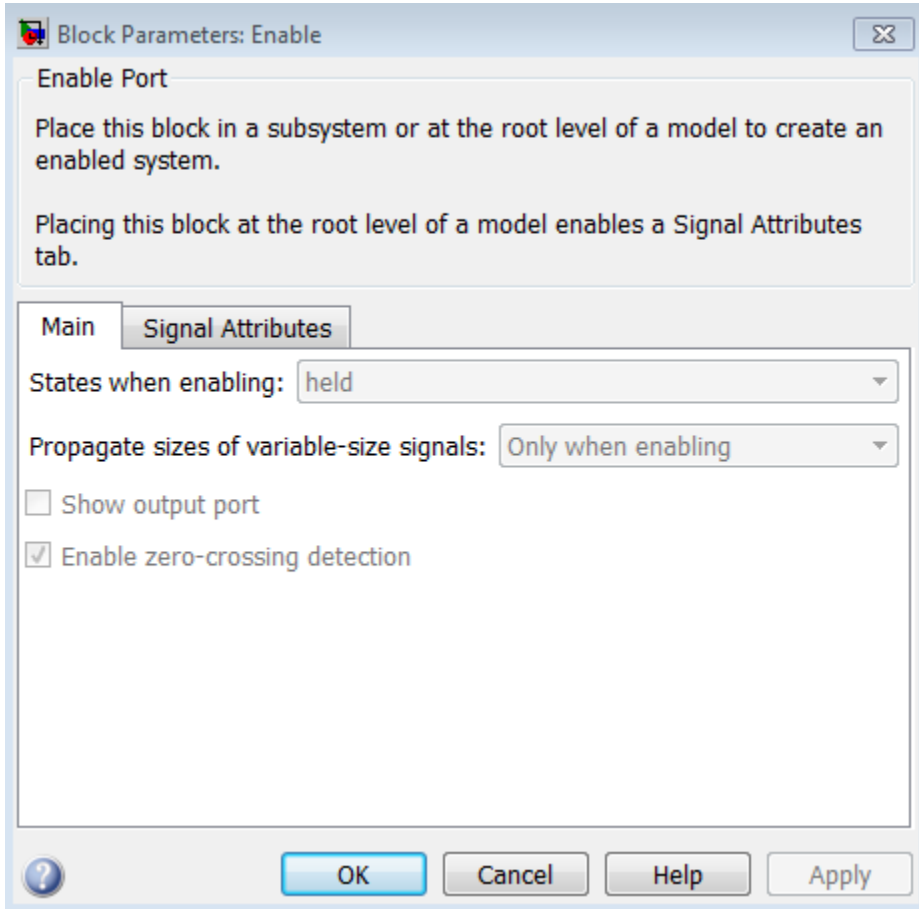
The Enable block supports signal label propagation.

Data Type Support

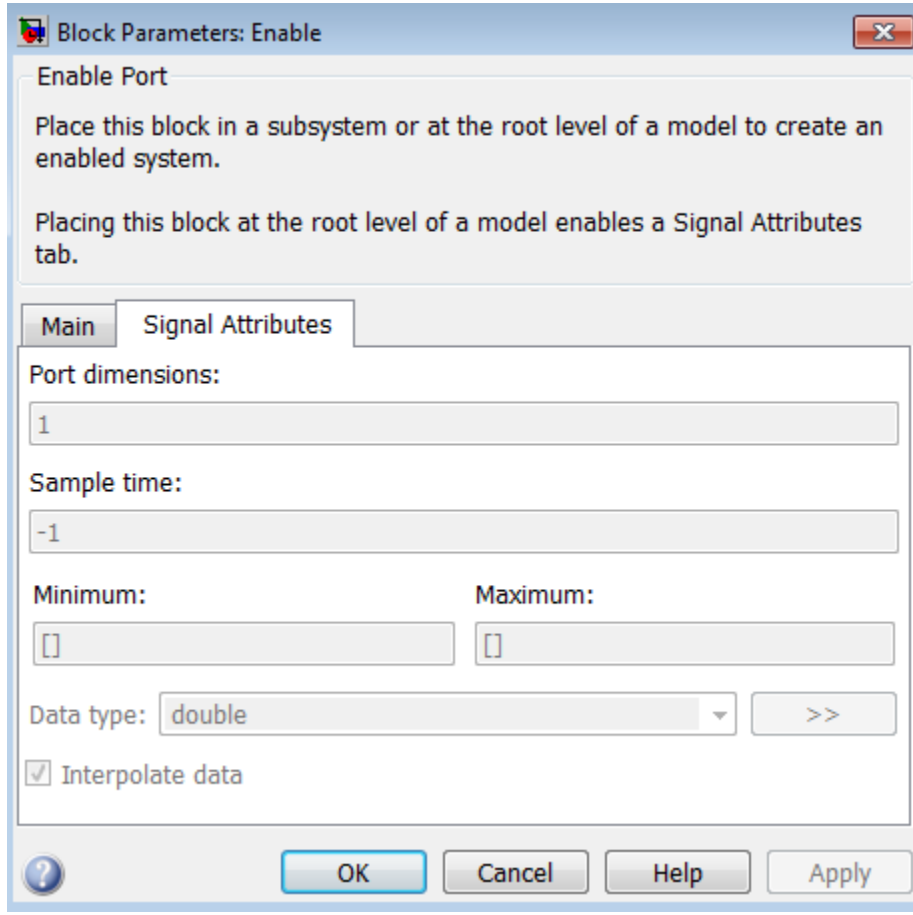
The Enable block accepts signals of supported Simulink numeric data types, including fixed-point data types. For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box

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



Placing the Enable block at the root of a model enables the **Signal Attributes** pane:



- "Data type" on page 1-633
- "Show data type assistant" on page 1-128
- "Mode" on page 1-636
- "Interpolate data" on page 1-638

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 6-101 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 6-101 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.

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 6-101 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.

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 6-101 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 6-101 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 “Control Signal Data Types”.

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 6-101 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 6-101 for the command-line information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Determined by the signal at the enable port
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

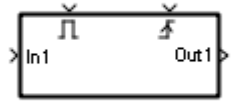
Introduced before R2006a

Enabled and Triggered Subsystem

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 "Create a Triggered and Enabled Subsystem" in the online Simulink help.

Characteristics

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

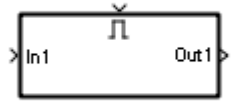
Introduced before R2006a

Enabled Subsystem

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 "Create an Enabled Subsystem" in the "Creating a Model" chapter of the Simulink documentation.

Characteristics

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

Introduced before R2006a

Enabled Synchronous Subsystem

Represent enabled subsystem that has synchronous reset and enable behavior

Library

HDL Coder / HDL Subsystems

Description 

An `Enabled Synchronous Subsystem` is an `Enabled Subsystem` that uses the `Synchronous` mode of the `State Control` block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create an `Enabled Synchronous Subsystem` block, add the block to your Simulink model from the HDL Subsystems block library. You can also add a `State Control` block with **State control** set to `Synchronous` inside an `Enabled subsystem`.

For more information, see `State Control` and “Create an Enabled Subsystem”.

Data Type Support

See `Inport` for information on the data types accepted by a subsystem's input ports. See `Outport` for information on the data types output by a subsystem's output ports.

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

Parameters

Show port labels

Cause Simulink software to display labels for the subsystem's ports on the subsystem's icon.

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 6-101 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 6-101 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 6-101 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 6-101 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 1-1911.
- "Sample time (-1 for inherited)" on page 1-1914
- "Function packaging" on page 1-1916 (requires a Simulink Coder license)

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from `Variant Source` blocks or to `Variant Sink` blocks.

Settings

Default: On

 On

Simulink treats the subsystem as a unit when propagating variant conditions from `Variant Source` blocks or to `Variant Sink` blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

 Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

Dependency

"Treat as grouped when propagating variant conditions" on page 1-1910 enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 1-1925 parameter setting. You can name the generated function and file using parameters "Function name" on page 1-1920 and "File name (no extension)" on page 1-1923. 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Characteristics

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

See Also

`Enable` | `State Control` | `Synchronous Subsystem`

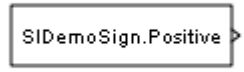
Introduced in R2016a

Enumerated Constant

Generate enumerated constant value

Library

Sources



Description

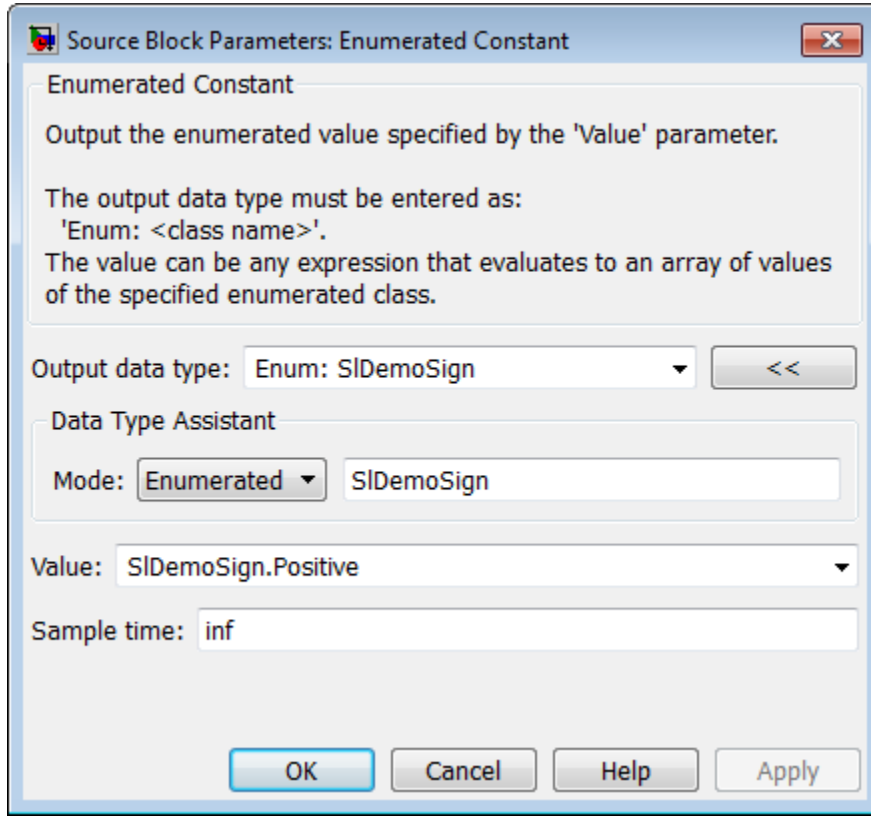
The Enumerated Constant block outputs a scalar, array, or matrix of enumerated values. You can also use the `Constant` block to output enumerated values, but it provides block parameters that do not apply to enumerated types, such as **Output minimum** and **Output maximum**. When you need a block that outputs only constant enumerated values, preferably use Enumerated Constant rather than Constant. For more information, see “Simulink Enumerations”.

Data Type Support

The Enumerated Constant block supports only enumerated data types. Use the `Constant` block to output constant data of other types. For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box

The Enumerated Constant block dialog box initially appears as follows:



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 6-101 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 6-101 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 6-101 for the command-line information.

Characteristics

Data Types	Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Not applicable

Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Related Examples

- “Use Enumerated Data in Simulink Models”

More About

- “Simulink Enumerations”

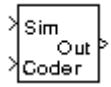
Introduced in R2009b

Environment Controller

Create branches of block diagram that apply only to simulation or only to code generation

Library

Signal Routing

Description 

This block outputs the signal at its Sim port only if the model that contains it is being simulated. It outputs the signal at its Coder port only if code is being generated from the model. This option enables you to create branches of a block diagram that apply only to simulation or code generation. The table below describes various scenarios where either the Sim or Coder port applies.

Scenario	Output
Normal mode simulation	Sim
Accelerator mode simulation	Sim
Rapid Accelerator mode simulation	Sim
Simulation of a referenced model (Normal or Accelerator modes)	Sim
Simulation of a referenced model (Processor-in-the-loop (PIL) mode)	Coder (uses the same code generated for a referenced model)
External mode simulation	Coder
Standard code generation	Coder
Code generation of a referenced model	Coder

Simulink Coder software does not generate code for blocks connected to the Sim port if these conditions hold:

- On the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box, you set **Default parameter behavior** to `Inlined`.
- The blocks connected to the Sim port do not have external signals.
- The Sim port input path does not contain 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.

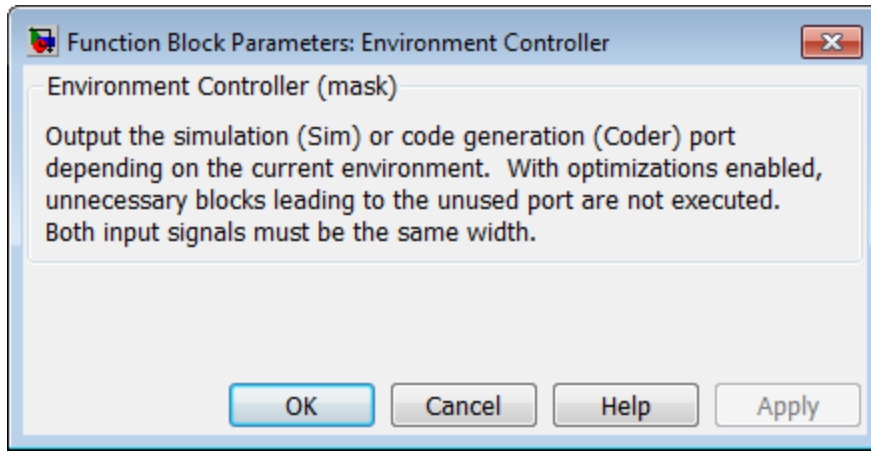
Note Simulink Coder code generation eliminates the blocks connected to the Sim branch only if the Sim branch has the same signal dimensions as the Coder branch. Regardless of whether it eliminates the Sim branch, Simulink Coder uses the sample times on the Sim branch as well as the Coder branch to determine 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.

Parameters and Dialog Box



Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Extract Bits

Output selection of contiguous bits from input signal

Library

Logic and Bit Operations

Description

The Extract Bits block allows you to output a contiguous selection of bits from the stored integer value of the input signal. Use the **Bits to extract** parameter to define the method for selecting the output bits.

- Select `Upper half` to output the half of the input bits that contain the most significant bit. If there is an odd number of bits in the input signal, the number of output bits is given by the equation
  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.

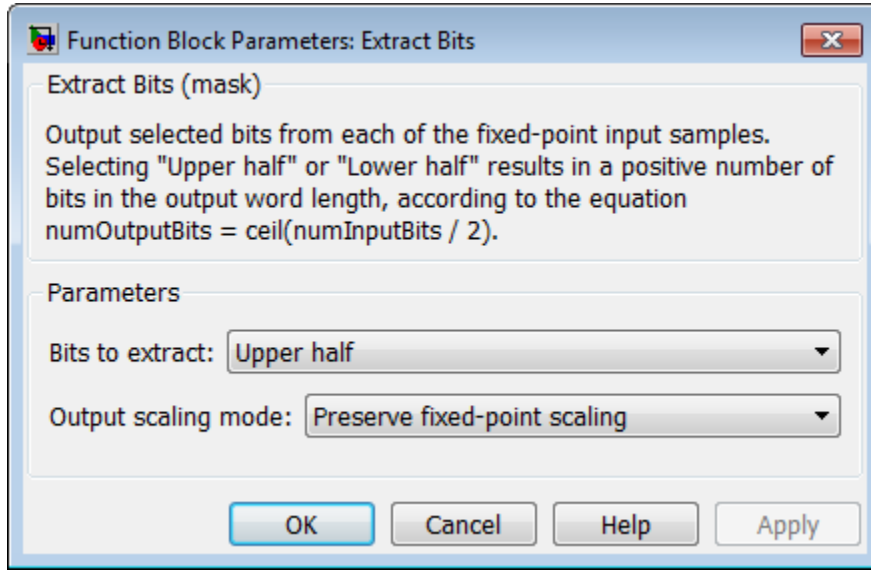
Data Type Support

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.

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 than two bits in them. Unsigned data integer types can be as short as a single bit.

This parameter is only visible if you select `Range starting with most significant bit` or `Range ending with least significant bit` for the **Bits to extract** parameter.

Bit indices

(Not shown on dialog above.) Specify a contiguous range of bits of the input signal to output. Specify the range in `[start end]` format. The indices are assigned to the input bits starting with 0 at the least significant bit.

This parameter is only visible if you select `Range of bits` for the **Bits to extract** parameter.

Output scaling mode

Select the scaling mode to use on the output bits selection:

- When you select `Preserve fixed-point scaling`, the fixed-point scaling of the input is used to determine the output scaling during the data type conversion.
- When you select `Treat bit field as an integer`, the fixed-point scaling of the input is ignored, and only the stored integer is used to compute the output data type.

Example

Consider an input signal that is represented in binary by `110111001`:

- If you select `Upper half` for the **Bits to extract** parameter, the output is `11011` in binary.
- If you select `Lower half` for the **Bits to extract** parameter, the output is `11001` in binary.
- If you select `Range starting with most significant bit` for the **Bits to extract** parameter, and specify `3` for the **Number of bits** parameter, the output is `110` in binary.
- If you select `Range ending with least significant bit` for the **Bits to extract** parameter, and specify `8` for the **Number of bits** parameter, the output is `10111001` in binary.

- If you select `Range of bits` for the **Bits to extract** parameter, and specify `[4 7]` for the **Bit indices** parameter, the output is `1011` in binary.

Characteristics

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

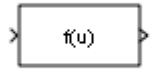
Introduced before R2006a

Fcn

Apply specified expression to input

Library

User-Defined Functions



Description

The Fcn block applies the specified mathematical expression to its input. The expression can include one or more of these components:

- `u` — The input to the block. If `u` is a vector, `u(i)` represents the `i`th element of the vector; `u(1)` or `u` alone represents the first element.
- Numeric constants.
- Arithmetic operators (`+ - * / ^`).
- Relational operators (`== != > < >= <=`) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Logical operators (`&& || !`) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Parentheses.
- Mathematical functions — `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `floor`, `hypot`, `log`, `log10`, `power`, `rem`, `sgn` (equivalent to `sign` in MATLAB), `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.

  Note: The `Fcn` block does not support `round` and `fix`. Use the `Rounding Function` block to apply these rounding modes.

- Workspace variables — Variable names that are not recognized in the preceding list of items are passed to MATLAB for evaluation. Matrix or vector elements must be specifically referenced (e.g., `A(1,1)` instead of `A` for the first element in the matrix).

The Fcn block observes the following rules of operator precedence:

1. `( )`
2. `^`
3. `+ -` (unary)
4. `!`
5. `* /`
6. `+ -`
7. `> < <= >=`
8. `== !=`
9. `&&`
10. `||`

The expression differs from a MATLAB expression in that the expression cannot perform matrix computations. Also, this block does not support the colon operator (:).

Block input can be a scalar or vector. The output is always a scalar. For vector output, consider using the `Math Function` block. If a block input is a vector and the function operates on input elements individually (for example, the `sin` function), the block operates on only the first vector element.

Limitations

The Fcn block has the following limitations:

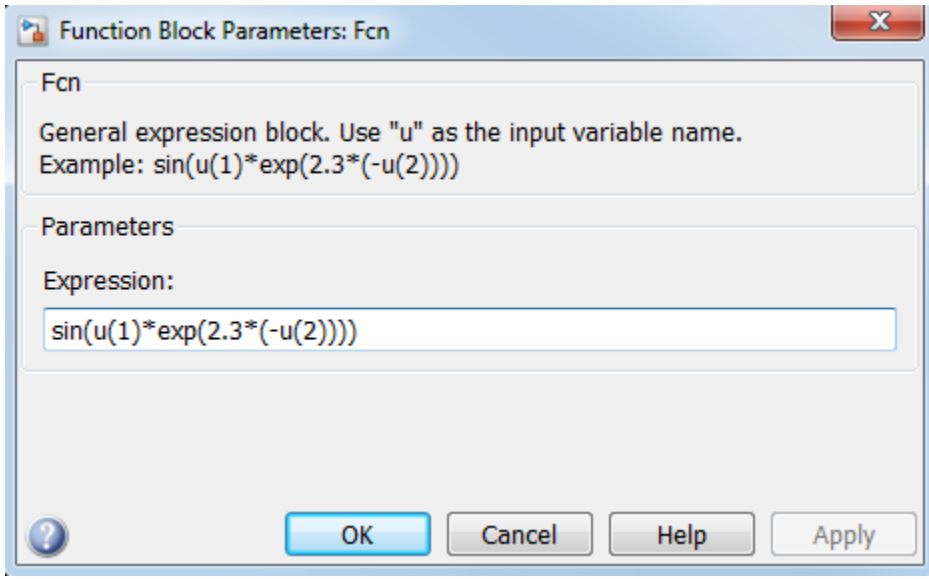
- You cannot tune the expression during simulation in Normal or Accelerator mode (see "How Acceleration Modes Work"), or in generated code. To implement tunable expressions, tune the expression outside the Fcn block. For example, use the `Relational Operator` block to evaluate the expression outside.
- The Fcn block does not support custom storage classes. See "Custom Storage Classes" 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



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 1-661)

Sample time

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

Examples

The following example models show how to use the `Fcn` block:

- `sldemo_absbrake`

- `sldemo_enginewc` (Throttle & Manifold/Throttle subsystem)

Characteristics

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

Introduced before R2006a

Find

Find nonzero elements in array

Library

Math Operations

Description

The Find block locates all nonzero elements of the input signal and returns the linear indices of those elements. If the input is a multidimensional signal, the Find block can also return the subscripts of the nonzero input elements. In both cases, you can show an output port with the nonzero input values.

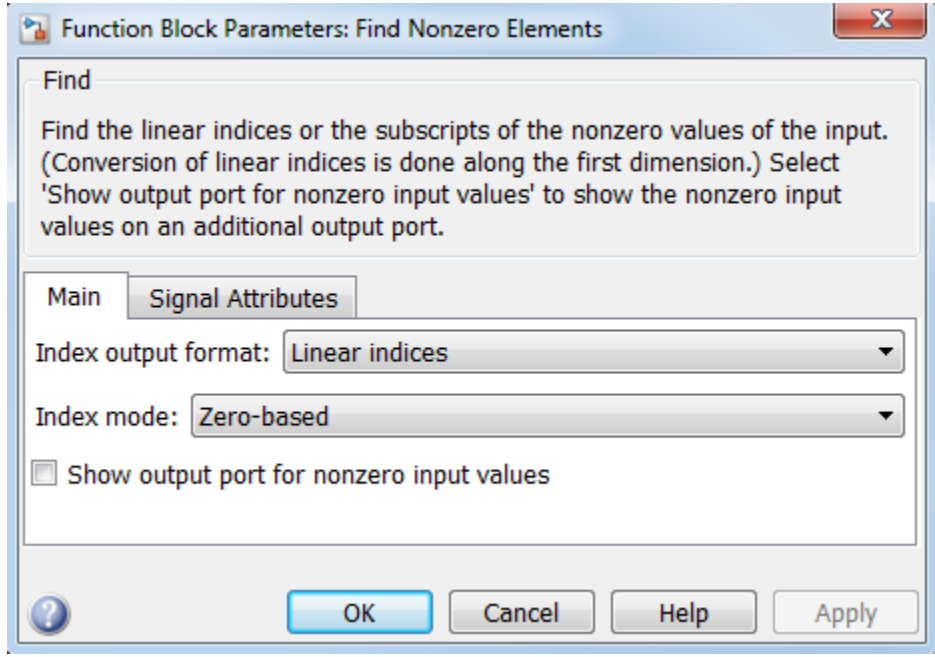
Data Type Support

The Find block accepts and outputs real values of any numeric data type that Simulink supports.

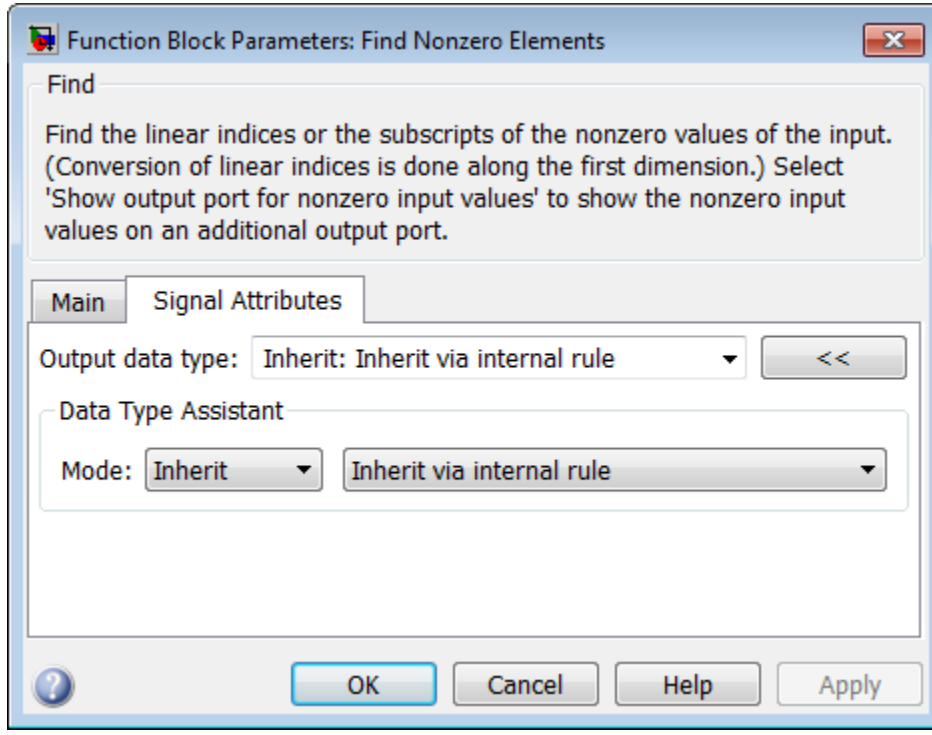
For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

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



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



- "Index output format" on page 1-668
- "Number of input dimensions" on page 1-669
- "Index mode" on page 1-670
- "Show output port for nonzero input values" on page 1-671
- "Sample time" on page 1-298
- "Output data type" on page 1-673
- "Mode" on page 1-675
- "Data type override" on page 1-230
- "Signedness" on page 1-678
- "Word length" on page 1-679
- "Scaling" on page 1-680

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

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

Output data type

Specify the output data type.

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`

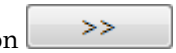
Output data type is `unt32`.

`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

"Control Signal 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

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

Introduced in R2010a

First-Order Hold

Implement first-order sample-and-hold

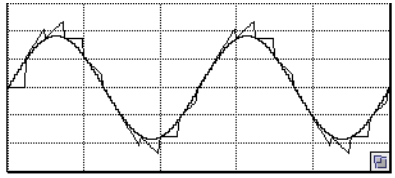
Library

Discrete

Description

The First-Order Hold block implements a first-order sample-and-hold that operates at the specified sampling interval. This block has little value in practical applications and is included primarily for academic purposes.

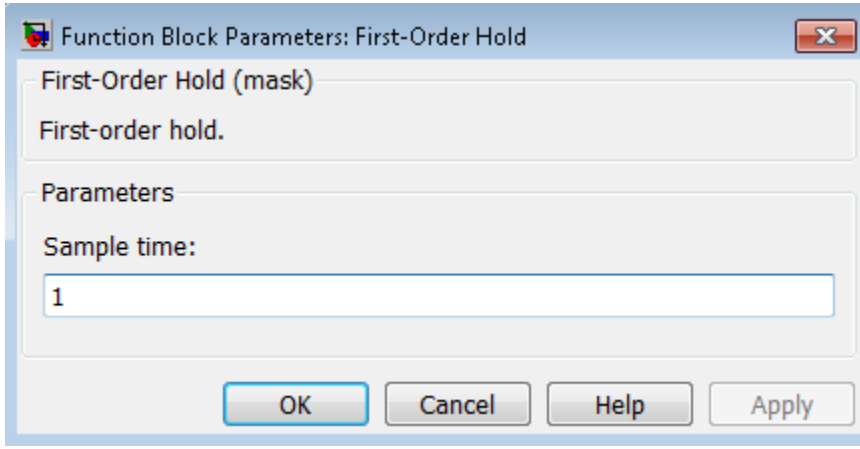
This figure compares the output from a Sine Wave block and a First-Order Hold block.



Data Type Support

The First-Order Hold block accepts and outputs signals of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Sample time

The time interval between samples. See “Specify Sample Time” in the online documentation for more information.

Characteristics

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

See Also

`Zero-Order Hold`

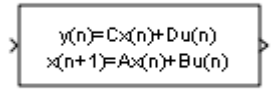
Introduced before R2006a

Fixed-Point State-Space

Implement discrete-time state space

Library

Additional Math & Discrete / Additional Discrete



Description

The Fixed-Point State-Space block implements the system described by

$y(n) = Cx(n) + Du(n)$

$x(n+1) = Ax(n) + Bu(n)$

where u is the input, x is the state, and y is the output. Both equations have the same data type.

The matrices A, B, C and D have the following characteristics:

- A must be an n-by-n matrix, where n is the number of states.
- B must be an n-by-m matrix, where m is the number of inputs.
- C must be an r-by-n matrix, where r is the number of outputs.
- D must be an r-by-m matrix.

In addition:

- The state x must be an n-by-1 vector.
- The input u must be an m-by-1 vector.
- The output y must be an r-by-1 vector.

The block accepts one input and generates one output. The block determines the input vector width by the number of columns in the B and D matrices. Similarly, the block determines the output vector width by the number of rows in the C and D matrices.

Data Type Support

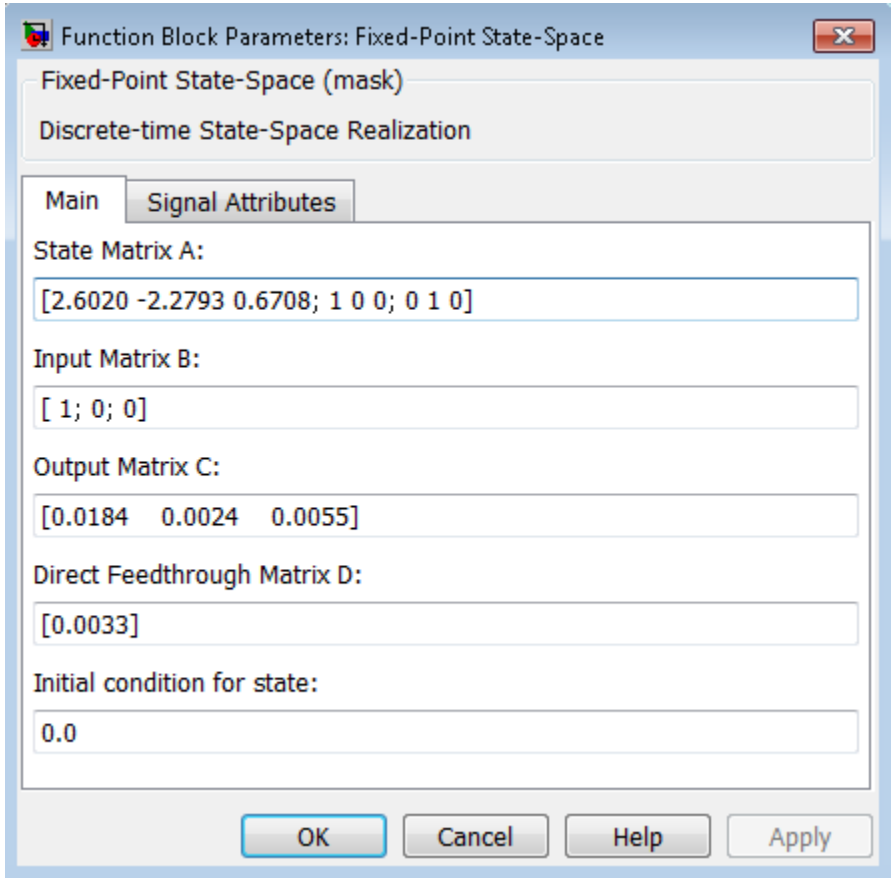
The Fixed-Point State-Space block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box

The **Main** pane of the Fixed-Point State-Space block dialog box appears as follows:



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.

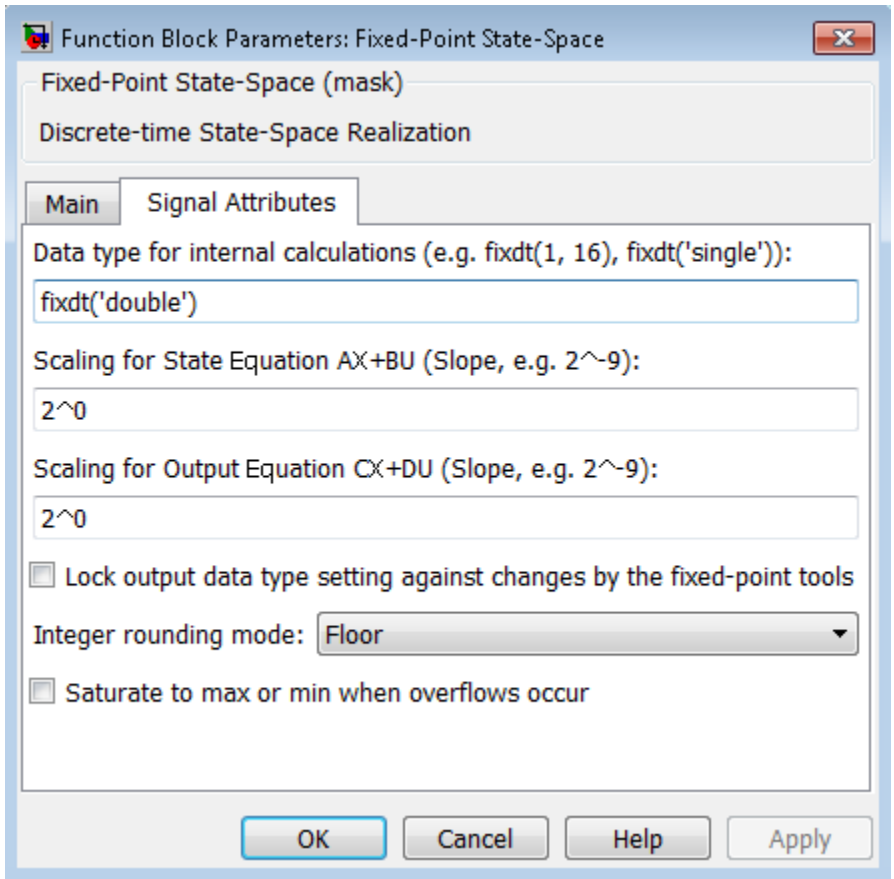
Direct Feedthrough Matrix D

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:



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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Discrete State-Space`

Introduced before R2006a

Floating Scope and Scope Viewer

Display signals generated during simulation

Library

Sinks

Description

The Simulink Floating Scope block and Scope Viewer display time domain signals with respect to simulation time.

Input signal characteristics:

- **Signal** — Continuous (sample-based) or discrete (sample-based and frame-based).
- **Signal data type** — Any data type that Simulink supports including real, complex, fixed-point, and enumerated data types. See “Data Types Supported by Simulink”.
- **Signal dimension** — Scalar, one-dimensional (vector), two dimensional (matrix), or multidimensional. Display multiple channels within a signal depending on its dimension. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

Floating Scope block characteristics:

- **Multiple y-axes (displays)** — Display multiple *y*-axes with multiple input ports. All of the *y*-axes have a common time range on the *x*-axis.
- **Multiple signals** — Show multiple signals on the same *y*-axis (display) from one or more input ports.
- **Modify parameters** — Modify scope parameter values before and during a simulation.
- **Display data after simulation** — If a Floating Scope is closed at the start of a simulation, scope data is still written to the scope during a simulation. As a result, if

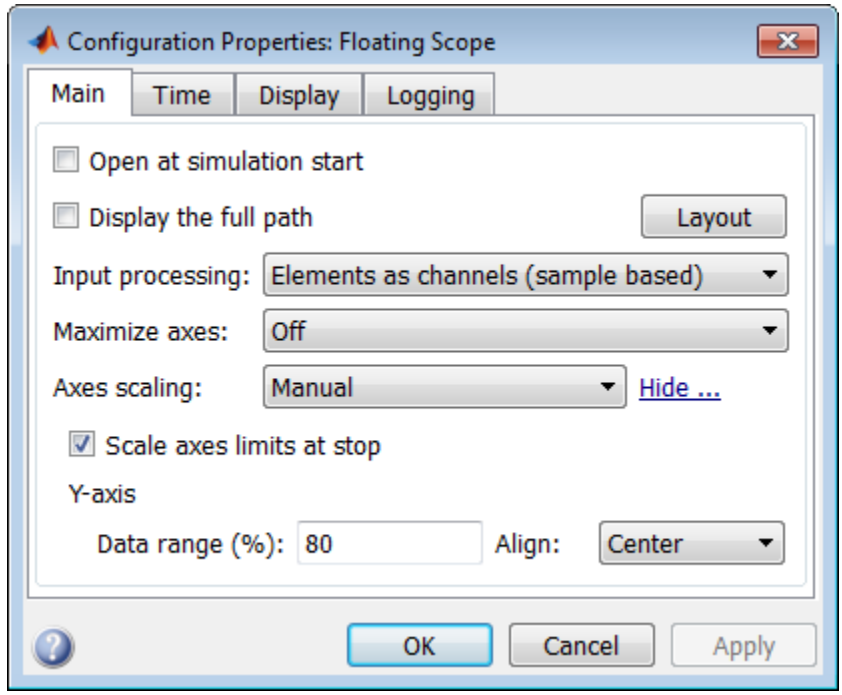
you open the Floating Scope after a simulation, the scope displays simulation results for input signals.

Note: A Floating Scope block and Scope Viewer have the same functionality as a Scope block, but they are not connected to signal lines. Use the Signal Selector to add and display signals on a Floating Scope.

Note: A Floating Scope does not have a toolbar Back button. If you step back using the Back button from your model, stepping back appears to work with Floating Scopes due to the data buffer, but it may not work in all cases.

Note: For information on controlling a Floating Scope programmatically, see “Control Scopes Programmatically” in the Simulink documentation.

Configuration Properties



Open at simulation start

Specify when a Scope window opens.

Settings

Default: Clear for Scope block. Select for Time Scope block.

☑ Select

Open Scope window when simulation starts.

☐ Clear

Do not open a closed Scope at the start of a simulation.

Display the full path

Display full block path on Scope title bar.

Settings

Default: Clear

 Select

Display block path and name.

 Clear

Display block name.

Scope Configuration property: No corresponding property.

Number of input ports

Specify number of input ports on a Scope block, specified by a positive integer string. Maximum number of input ports is 96. This property does not apply to floating scopes and scope viewers..

Default: 1

Scope Configuration property: `NumInputPorts`.

Layout button

Specify number of displays. The maximum layout dimension is four rows by four columns.

- If the number of *y*-axes are equal to the number of ports, signals from each port appear on separate displays.
- If the number of *y*-axes are less than the number of ports, signals from additional ports appear on the last *y*-axis.

Settings

Default: 1 display

Scope Configuration property: `LayoutDimensions`.

Sample time

Specify time interval between Scope block updates during a simulation, specified as a positive real string. This property does not apply to floating scopes and scope viewers.

Settings

Default: `-1` for inherited

Scope Configuration property: `SampleTime`.

Input processing

Specify sample-based or frame-based processing of signals.

Settings

Default: `Elements as channels (sample based)` for Scope block. `Columns as channels (frame based)` for Time Scope block.

`Elements as channels (sample based)`

Process signal values in a channel at each time interval.

`Columns as channels (frame based)`

Process signal values in a channel as a group of values from multiple time intervals. Frame-based processing is available only with discrete input signals.

Scope Configuration property: `FrameBasedProcessing`.

Maximize axes

Maximize size of signal plots. Each of the plots expands to fit the full display. Maximizing the size of signal plots removes the background area around the plots.

Settings

Default: `Off` for Scope block. `Auto` for Time Scope block.

`Auto`

If **Title** and **Y-label** properties are not specified, maximize all plots.

`On`

Maximize all plots. Values in **Title** and **Y-label** are hidden

`Off`

Do not maximize plots.

Scope Configuration property: `MaximizeAxes`.

Axes scaling

Specify when to scale *y*-axis to include all signal values.

Settings

Default: `Manual`

`Manual`

Manually scale y-axis range with **Scale Y-axis Limits** toolbar button.

`Auto`

Scale *y*-axis range during and after simulation. Selecting this option displays the **Do not allow Y-axis limits to shrink** check box.

`After N Updates`

Scale *y*-axes after specified number of block updates (time intervals). Selecting this option displays the **Number of updates** text box.

Do not allow Y-axis limits to shrink

Specify when *y*-axis range limits can change.

Settings

Default: Select

☑ Select

Allow *y*-axis range limits to increase but not decrease during a simulation.

☐ Clear

Allow y-axis range limits to increase and decrease..

Dependency

Click the **Configure** link to the right of the **Axes scaling** property and set the **Axes scaling** property to `Auto` to display this property.

Number of updates

Specify the number of updates that occur during a simulation before a Scope scales the *y*-axes, specified as a positive integer string.

Settings

Default: `10`

Dependency

Display and activate this property by clicking the **Configure** link to the right of the **Axes scaling** property and set the **Axes scaling** property to `After N Updates`.

Scope Configuration property: `AxesScalingNumUpdates`.

Scale axes limits at stop

Specify when to scale axes.

Settings

Default: Select

☑ Select

Scale axes when simulation stops.

☐ Clear

Always scale axes.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

The *y*-axes limits are always scaled. The *x*-axis limits are scaled only if you also select the **Autoscale X-axis limits** check box.

Y-axis Data range (%)

Specify percentage of *y*-axis range for plotting data. For example, if you set this property to `100`, plotted data uses the entire *y*-axis range.

Settings

Default: `80`

Values are 1 through 100.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Y-axis Align

Specify where to align plotted data along the *y*-axis data range when **Y-axis Data range** is set to less than 100 percent.

Settings

Default: `Center`

`Top`
: Align signals with maximum values at top of *y*-axis range.

`Center`
: Align signals with maximum and minimum values centered.

`Bottom`
: Align signals with minimum values at bottom of *y*-axis range.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Autoscale X-axis limits

Scale *x*-axis range limits when scaling axes.

Settings

Default: Clear

Select

Scale *x*-axis range to fit all signal values. If **Axes scaling** is set to `Auto`, scales the data currently within the axes, not the entire signal in the data buffer.

Clear

Do not scale *x*-axis range.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

X-axis Data range (%)

Specify percentage of *x*-axis range for plotting data. For example, if you set this property to `100`, plotted data uses the entire *x*-axis range.

Settings

Default: `100`

Values are 1 through 100.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

X-axis Align

Specify where to align plotted data along the *x*-axis when **X-axis Data range** is set to less than 100 percent.

Settings

Default: `Center`

`Top`
Align signals with maximum values at top of *x*-axis range.

`Center`
Align signals with maximum and minimum values centered.

`Bottom`
Align signals with minimum values at bottom of *x*-axis range.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Time span

Specify length of *x*-axis range to display.

The block calculates the beginning and end times of the time range using the **Time display offset** and **Time span** properties. For example, if you set the **Time display offset** to `10` and the **Time span** to `20`, the scope sets the time range from `10` to `30`.

Settings

Default: `Auto`

`Auto`
Difference between the simulation start and stop times.

`User defined`
Value less than the total simulation time.

Scope Configuration property: `TimeSpan`.

Time span overrun action

Specify how to display data beyond the visible *x*-axis range.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

Settings

Default: `Wrap`

`Wrap`

Draw a full screen of data from left to right, clear the screen, and then restart drawing of data.

`Scroll`

Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

Scope Configuration property: `TimeSpanOverrunAction`.

Time units

Display units for the *x*-axis.

Settings

Default: `None` for Scope block. `Metric` for Time Scope block.

`Metric`

Display time units based on the length of **Time span**.

`Seconds`

Display Time (seconds).

`None`

Do not display time units.

Scope Configuration property: `TimeUnits`.

Time display offset

Offset the *x*-axis by a specified time value, specified as a real number or vector of real numbers.

For input signals with multiple channels, you can enter a scaler or vector.

- Scaler — Offset all channels of an input signal by the same time value.
- Vector — Independently offset the channels.

Settings

Default: `0`

Scope Configuration property: `TimeDisplayOffset`.

Time-axis labels

Specify how *x*-axis (time) labels dispaly

Settings

Default: `Bottom Displays Only` for Scope block. `All` for Time Scope block.

`All`

Display *x*-axis labels on all *y*-axes.

`None`

Do not display labels. Selecting `None` also clears the **Show time-axis label** check box.

`Bottom Displays Only`

Display *x*-axis label on the bottom *y*-axis.

Dependency

Set **Active display** before setting this property. Activate this property by selecting **Show time-axis label** and setting **Maximize axes** to `off`.

Scope Configuration property: `TimeAxisLabels`.

Show time-axis label

Display or hide *x*-axis (time) labels.

Settings

Default: `Clear` for Scope block. `Select` for Time Scope block.

Select

Display *x*-axis label for the active display

Clear

Hide *x*-axis labels.

Dependency

Set **Active display** before setting this property. If you select this property and set the **Time-axis labels** is set to `None`, this property is deactivated.

Scope Configuration property: `ShowTimeAxisLabel`.

Active display

Display for setting display-specific properties, specified as a positive integer. The number of a display corresponds to its column-wise placement index.

Settings

Default: 1

Dependency

Setting this property selects the display for setting the properties **Show Grid**, **Show legend**, **Title**, **Plot signals as magnitude and phase**, **Y-label**, and **Y-Limits**.

Scope Configuration property: `ActiveDisplay`.

Title

Specify a title for display, specified as a character string. The default value `%<SignalLabel>` uses the input signal name for the title..

Settings

Default: `%<SignalLabel>`

Dependency

Set **Active display** before setting this property.

Scope Configuration property: `Title`.

Show legend

Show signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. See the `Scope` block reference for an example.

Settings

Default: Clear

☑ Select

Display signal legend. Continuous signals have straight lines before their names and discrete signals have step-shaped lines.

☐ Clear

Hide signal legend.

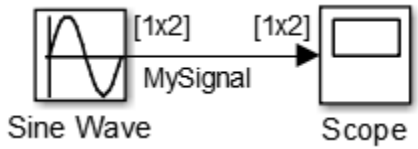
Dependency

Set **Active Display** before setting this property.

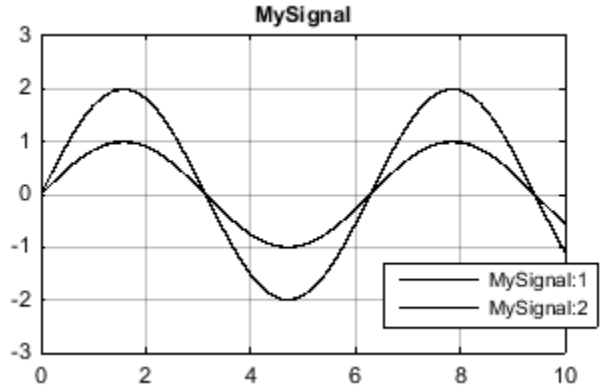
Scope Configuration property: `ShowLegend`.

Example

Connect a Sine Wave block to a Scope. Set the **Amplitude** parameter for the Sine Wave to `[1 2]`. Select the **Legends** check box for the Scope. Set the **Signal name** property for the signal to `MySignal`.



After simulating this model, the Scope window displays a sine wave for the two signal channels in `MySignal`, `MySignal:1`, and `MySignal:2`.



Edit the name of any channel in the legend by double-clicking the current name and entering a new channel name.

See also “Signal Dimensions”and “Determine Output Signal Dimensions”.

Show grid

Show vertical and horizontal grid lines.

Settings

Default: Select

☑ Select

Display grid lines.

☐ Clear

Hide grid lines.

Dependency

Set **Active Display** before setting this property.

Scope Configuration property: `ShowGrid`.

Plot signals as magnitude and phase

Split display into magnitude and phase plots.

Settings

Default: Clear

 Select

Display magnitude and phase plots. If the signal is real, plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values.

 Clear

Display signal plot. If the signal is complex, plots the real and imaginary parts on the same *y*-axis.

Dependency

Set **Active Display** before setting this property.

Scope Configuration property: `PlotAsMagnitudePhase`.

Y-limits (Minimum)

Specify minimum value of *y*-axis.

Settings

Default: `-10`

Dependency

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** applies this property value to the magnitude plot. The *y*-axis limits of the phase plot are always `[-180 180]`.

Scope Configuration property: `YLimits`.

Y-limits (Maximum)

Specify maximum value of *y*-axis, specified as real number.

Settings

Default: `+10`

Dependency

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** applies this property value to the magnitude plot. The *y*-axis limits of the phase plot are always `[-180 180]`.

Scope Configuration property: `YLimits`.

Y-label

Specify *y*-axis label, specified as a character string.

Settings

Default: No label for Scope block. `Amplitude` for Time Scope block.

Dependency

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** hides this property and plots are labeled `Magnitude` and `Phase`.

Scope Configuration property: `YLabel`.

Limit data points to last

Specify to limit buffered data values before plotting and saving signals.

Settings

Default: Clear, 5000

☑ Select

Save specified number of data values for each signal. If the signal is frame-based, the number of buffered data values is the specified number of data values multiplied by the frame size.

For simulations with **Stop time** set to `inf`, consider selecting **Limit data points to last**.

In some cases, for example where the sample time is small, selecting this parameter can have the effect of plotting signals for less than the entire time range of a simulation. If a scope plots a portion of your signals, consider increasing the number of data values to save.

☐ Clear

Save and plot all data values. Clearing **Limit data points to last** can cause an out-of-memory error for simulations that generate a large amount of data or for systems without enough available memory.

Dependency

If this property is selected, also specify the number of data points by entering a positive integer in the text box. This property limits the data values a scope plots and the data values saved in the MATLAB variable specified in **Variable name**. Data values are from the end of a simulation.

Scope Configuration properties: `DataLoggingLimitDataPoints` and `DataLoggingMaxPoints`.

Decimation

Reduce the amount of scope data to display and save.

Settings

Default: Clear, `2`

☑ Select

Plot and Log (save) scope data every `N`th data point, where `N` is the decimation factor entered in the text box.

☐ Clear

Save all scope data values.

Dependency

If this property is selected, also specify the decimation factor by entering a positive integer in the text box. The scope buffers every `N`th data point, where `N` is the decimation factor you specify. A value of `1` buffers all data values. This property limits the data

values a scope plots and the data values saved in the MATLAB variable specified in **Variable name**.

Log/Unlog Viewed Signals to Workspace

For signals selected with the Signal Selector, clicking this button selects the **Log signal data** check boxes in the Signals Properties dialog boxes.

Clicking the button a second time clears the **Log signal data** check boxes.

Style Properties

Open the Style dialog box:

- From the menu, select **View > Style**.
- From the Configuration Properties button arrow. select the Style button  .

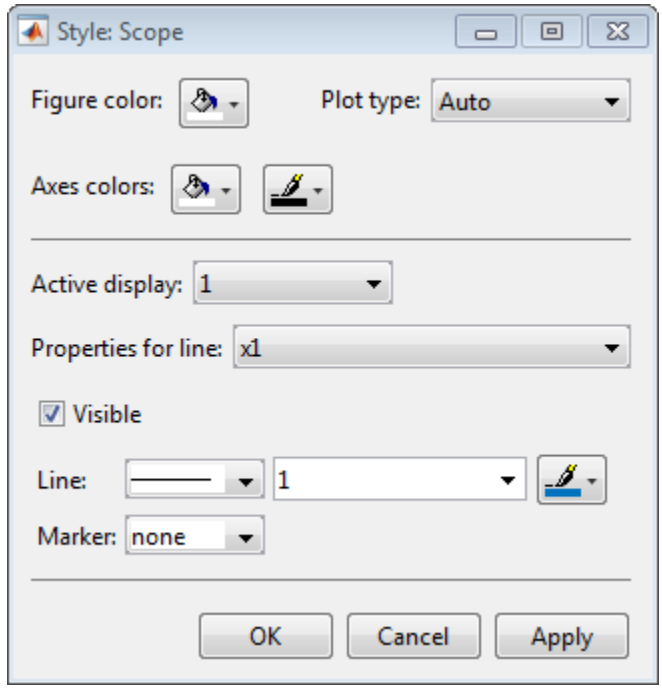


Figure color

Select background color for display.

Plot type

Specify how to plot a signal.

Default: `Auto` for Scope block. `Line` for Time Scope block.

- `Line` — Line graph.
- `Stairs` — Stair-step graph.
- `Auto` — Line graph if it is a continuous signal or a stair-step graph if it is a discrete signal.

Active display

Select active display for setting style properties.

Default: 1

Axes colors

Select the background color for axes (displays) with the first color pallet. Select the grid and label color with the second color pallet.

Properties for line

Select active line for setting line style properties.

Visible

Plot signal on active display.

Default: Select


Select

Plot signal.


Clear

Hide signal.

Line

Select line style, width, and color.

Marker

Select marker style.

Default: None

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	No

See Also

`Scope`

How To

- “Scope Blocks and Scope Viewer Overview”
- “Simulate a Model Interactively”
- “Step Through a Simulation”
- “Scope Tasks”
- “Floating Scope and Scope Viewer Tasks”
- “Scope Trigger Panel”
- “Scope Measurement Panels”
- “Control Scopes Programmatically”

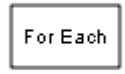
Introduced in R2015b

For Each

Enable blocks inside For Each Subsystem to process elements or subarrays of a mask parameter or input signal independently

Library

Ports & Subsystems



Description

The `For Each` block serves as a control block for the `For Each Subsystem` block. Specifically, the `For Each` block enables the blocks inside the `For Each Subsystem` to process elements of the mask parameters or 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.

Iterations in the For Each Subsystem

You can use a For Each subsystems to iteratively compute output after changing inputs or mask parameters. This is done by configuring the partitioning them in the For Each block dialog box.

Partition Input Signals to the Subsystem

In a For Each subsystem, you can specify which input signals to partition for each iteration, using the **Input Partition** tab in the dialog box of the For Each block. When specifying a signal to be partitioned, you also have to specify the **Partition Dimension** and **Partition Width** parameters. For more information, see “Select Partition Parameters” on page 1-711.

Partition Parameters in the For Each block

You can partition the mask parameters of For Each subsystems. Partitioning is useful for systems that have identical structures in each iteration but different parameter values. In this case, changing the model to partition extra input signals for each parameter is cumbersome. Instead, add a mask parameter to a For Each subsystems, see "Create a Simple Mask". To select the mask parameter for partitioning, use the Parameter Partition tab on the For Each block dialog box. For more information, see "Select Partition Parameters" on page 1-711

Concatenate Output

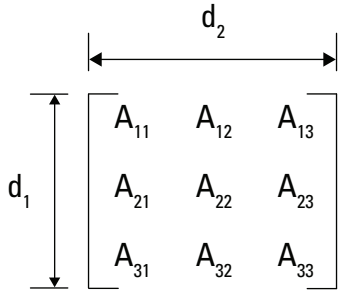
You define the dimension along which to concatenate the results by specifying the **Concatenation Dimension** in the Output Concatenation tab.

The results generated by the block for each subarray stack along the concatenation dimension, d_1 (y-axis). Whereas, if you specify d_2 by setting the concatenation dimension to 2, the results concatenate along the d_2 direction (x-axis). Thus if the process generates row vectors, then the concatenated result is a row vector.

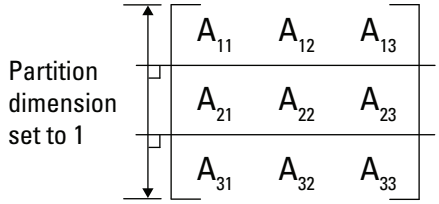
Select Partition Parameters

When selecting an input signal or subsystem mask parameter for partitioning, you need specify how to decompose it into elements or subarrays for each iteration. Do this by setting integer values for the **Partition Dimension** and the **Partition Width** parameters.

As an illustration, consider a mask parameter 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.



Mask parameter A decomposes into the following three row vectors:

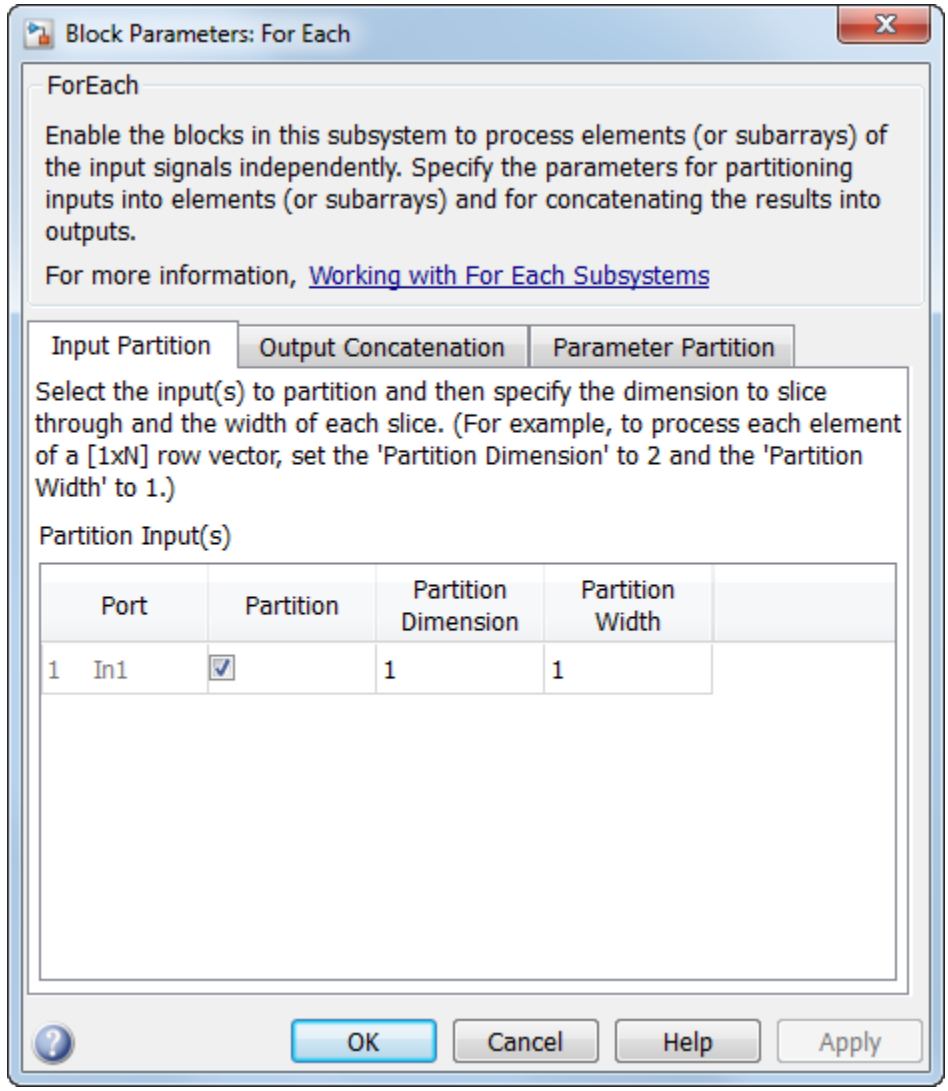
$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \end{bmatrix}$$
$$\begin{bmatrix} A_{21} & A_{22} & A_{23} \end{bmatrix}$$
$$\begin{bmatrix} A_{31} & A_{32} & A_{33} \end{bmatrix}$$

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:

$$\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix} \quad \begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix} \quad \begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix}$$

Note: Only signals are considered one-dimensional in Simulink. Mask parameters are row or column vectors according to their orientation. To partition a row vector, specify the partition dimension as 2 (along the columns). To partition a column vector, specify the partition dimension as 1 (along the rows).

Parameters and Dialog Box



- "Input Partition Tab" on page 1-714
- "Output Concatenation Tab" on page 1-714

• "Parameter Partition Tab" on page 1-715

Input Partition Tab

Use this tab to select each input signal you need to partition and to specify the corresponding **Partition Dimension** and **Partition Width** parameters. See the `Inport` block reference page for more information.

Port

The **Port** column displays the input index and the name of the input port connected to the For Each Subsystem block.

Partition

Select the check box beside each input signal that you want to partition into subarrays or elements. Selecting this check box enables **Partition Dimension** and **Partition Width** for that input signal.

Default: Off

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.

Default: 1

Minimum: 1

Partition Width

Specify the width of each partition slice of the input signal. The default width of 1 represents a width of one element.

Default: 1

Minimum: 1

Output Concatenation Tab

For each output port, specify the dimension along which to stack (concatenate) the For Each Subsystem results. See the `Outport` block reference page for more information.

Port

The **Port** column displays the output index and the name of the output port connected to the For Each Subsystem block. You can have any number of ports.

Concatenation Dimension

Specify the dimension along which to stack the results of the For Each Subsystem.

Default: 1

Minimum: 1

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.

Parameter Partition Tab

Use this tab to select each mask parameter to partition and to specify the corresponding **Partition Dimension** and **Partition Width** parameters. Parameters appear in the list only if you have added an editable parameter to the mask of the parent For Each subsystem.

Parameter

The **Parameter** column displays the name of the mask parameter of the For Each Subsystem block.

Partition

Select the check box next to each mask parameter that you want to partition into subarrays or elements. Selecting this check box makes the **Partition Dimension** and **Partition Width** parameters available for that mask parameter.

Default: Off

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.

Default: 1

Minimum: 1

Partition Width

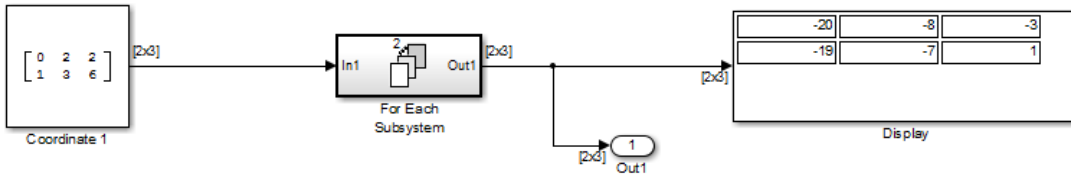
Specify the width of each partition slice of the input signal. The default width of 1 represents a width of one element.

Default: 1

Minimum: 1

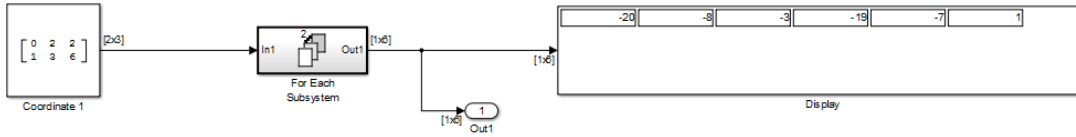
Examples

The following model demonstrates the partitioning of an input signal by a For Each block. Each row of this 2-by-3 input array contains three integers that represent the (*x*, *y*, *z*)-coordinates of a point. The goal is to translate each of these points based on a new origin at (–20, –10, –5) and to display the results.



By placing the process of summing an input signal and the new origin inside of a For Each Subsystem, you can operate on each set of coordinates by partitioning the input signal into two row vectors. To accomplish such partitioning, use the default settings of 1 for both the partition dimension and the partition width. If you also use the default concatenation dimension of 1, each new set of coordinates stacks in the d_1 direction, making your display a 2-by-3 array.

Alternatively, if you specify a concatenation dimension of 2, then you get a single row vector because each set of results stacks in the d_2 direction.



This example shows how to partition an input signal. To learn how the `For Each` block and subsystem handle a model with states, see the `For Each Subsystem` documentation.

See Also

`For Each Subsystem`

"Repeat an Algorithm Using a For Each Subsystem"

Introduced in R2010a

For Each Subsystem

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

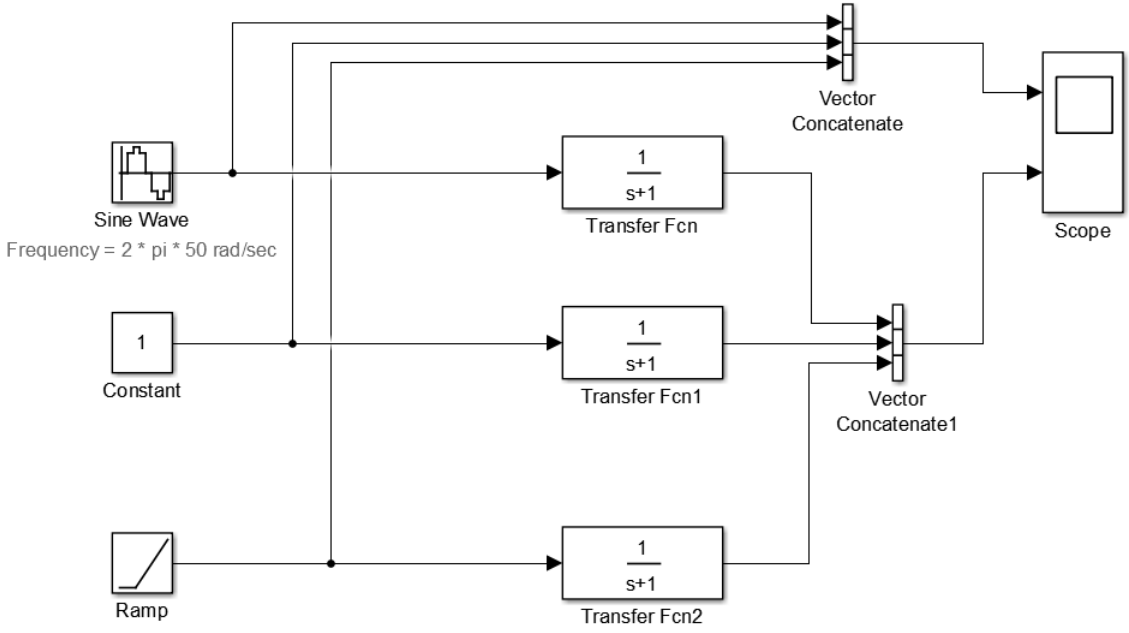
For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Examples of Working with For Each Subsystems

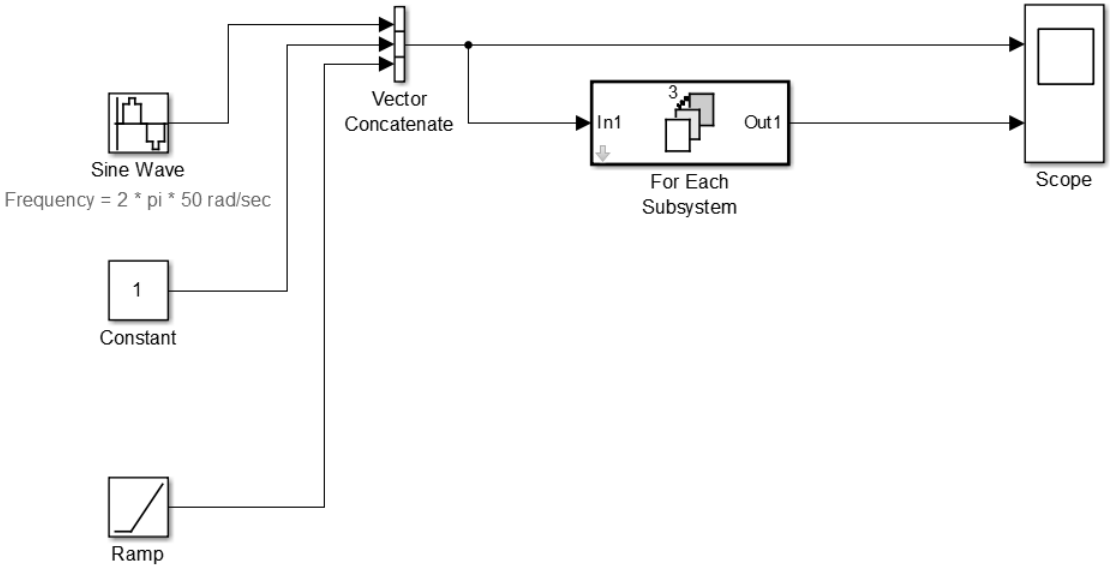
Vectorize Algorithms Using For Each Subsystems

This example shows how to simplify modeling of vectorized algorithms. Using For Each subsystem blocks simplifies a model where three input signals are filtered by three identical Transfer Fcn blocks. This example also shows how to add more control to the filters by changing their coefficients for each iteration of the subsystem.

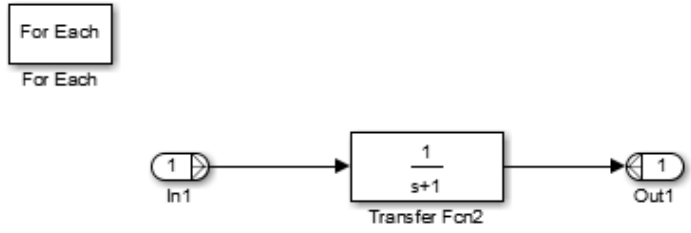
This model uses identical Transfer Fcn blocks to independently process each element of the input sine wave signal. A `Vector Concatenate` block concatenates the resulting output signals. This repetitive process is graphically complex and difficult to maintain. Also, adding another element to the signal requires significant reworking of the model.



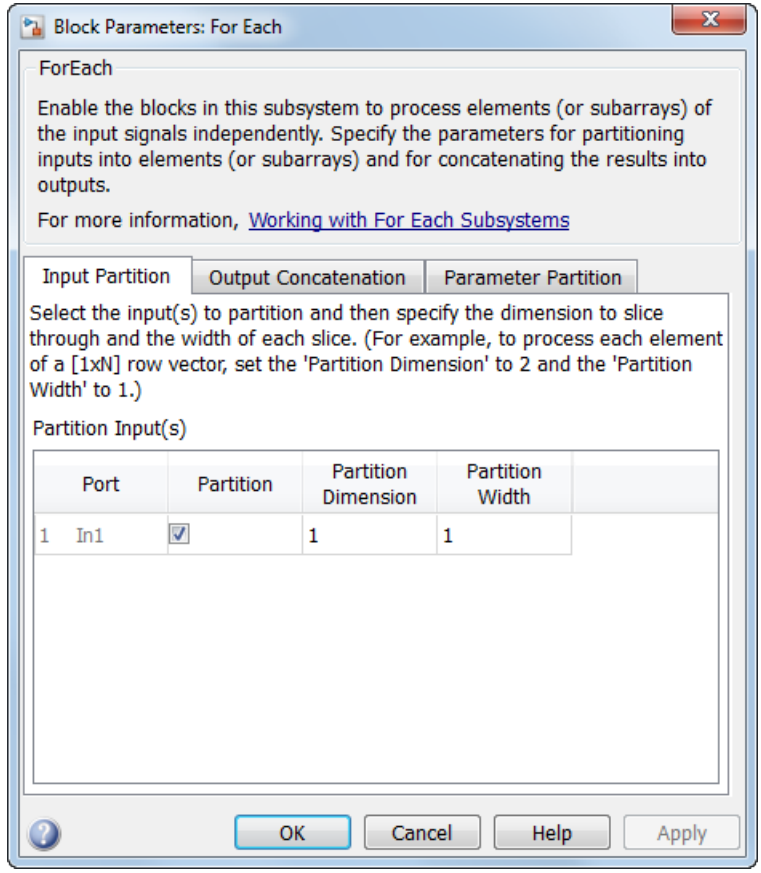
You can simplify this model by replacing the repetitive operations with a single For Each Subsystem block.



The For Each subsystem contains a `For Each` block and a model representing the algorithm of the three blocks it replaces by way of the Transfer Fcn block. The `For Each` block specifies how to partition the input signal vector into individual elements and how to concatenate the processed signals to form the output signal vector. Every block that has a state maintains a separate set of states for each input element processed during a given execution step.



For this example, the input signal is selected for partitioning. The **Partition Dimension** and the **Partition Width** parameters on the For Each block are both set to `1` for the input.

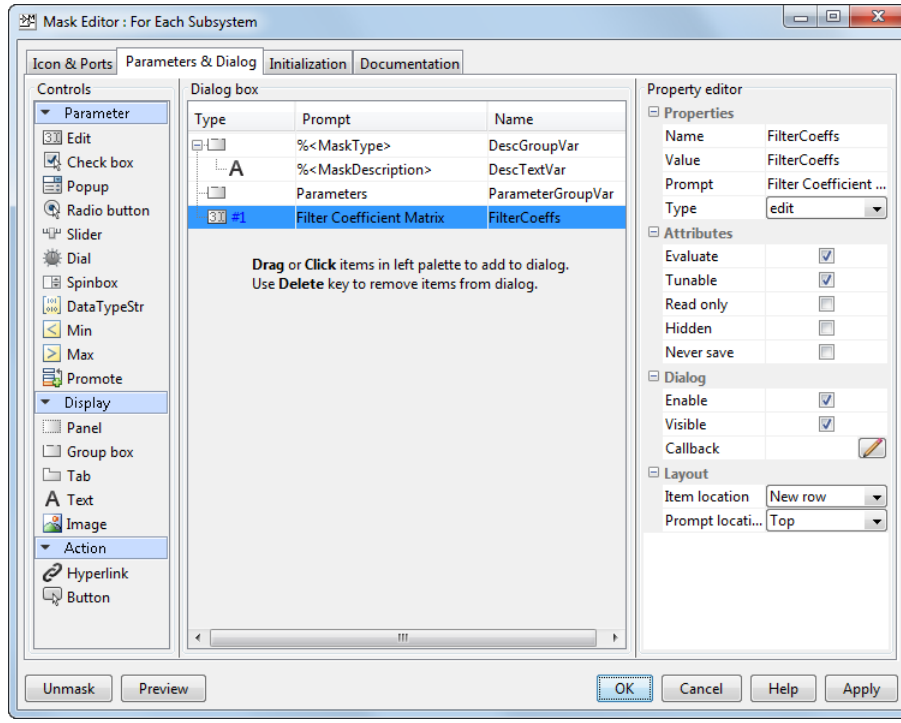


You can scale up this approach to add more signals without having to change the model significantly. This approach is considered easily scalable and graphically simpler.

Model Parameter Variation Without Changing Model Structure

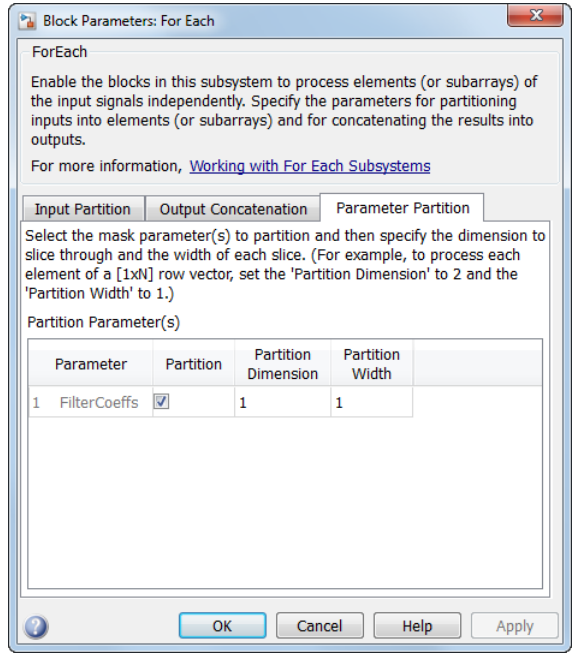
This example shows how to model parameter variation in an algorithm. It uses the For Each Subsystem partitioning model from "Vectorize Algorithms Using For Each Subsystems" on page 1-723 and creates different filters for each input signal while retaining model simplicity. You do this by feeding an array of filter coefficients to the For Each subsystem as a mask parameter marked for partitioning. In each iteration of the For Each subsystem, a partition of the filter coefficients is fed to the Transfer Fcn block.

1 Open the model ex_ForEachSubsystem_Partitioning. Create a mask for the For Each Subsystem block and add an editable mask parameter. Set the name to `FilterCoeffs` and the prompt to `Filter Coefficient Matrix`. For information on how to add a mask parameter, see “Create a Simple Mask”.



2 Open the For Each subsystem. Inside the subsystem, open the For Each block dialog box.

3 In the **Parameter Partition** tab, select the check box next to the **FilterCoeffs** parameter to enable partitioning of this parameter. Keep the **Partition Width** and **Partition Dimension** parameters at their default value of 1.



4. Double-click the For Each Subsystem block and enter your filter coefficient matrix, having one row of filter coefficients for each input signal. For example, enter `[0.0284 0.2370 0.4692 0.2370 0.0284; -0.0651 0 0.8698 0 -0.0651; 0.0284 -0.2370 0.4692 -0.2370 0.0284]` to implement different fourth-order filters for each input signal.

5. In the For Each subsystem, double-click the Transfer Fcn block and enter `FilterCoeffs` for the **Denominator Coefficients** parameter. This setting causes the block to get its coefficients from the mask parameter.

The For Each subsystem slices the input parameter into horizontal partitions of width 1, which is equivalent to one row of coefficients. The parameter of coefficients transforms from a single array:

Partition dimension set to 1

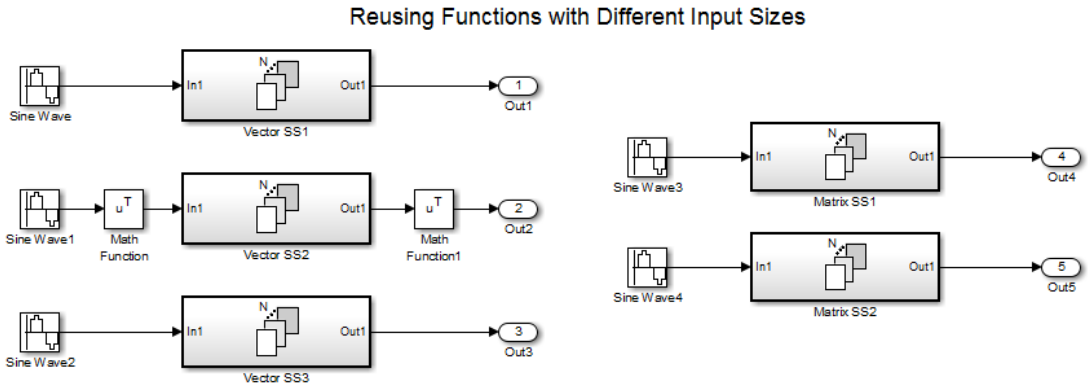
$$\begin{bmatrix} 0.0284 & 0.2370 & 0.4692 & 0.2370 & 0.0284 \\ -0.0651 & 0.0000 & 0.8698 & 0.0000 & -0.0651 \\ 0.0284 & -0.2370 & 0.4692 & -0.2370 & 0.0284 \end{bmatrix}$$

into three rows of parameters:

$$\begin{bmatrix} 0.0284 & 0.2370 & 0.4692 & 0.2370 & 0.0284 \end{bmatrix}$$
$$\begin{bmatrix} -0.0651 & 0.0000 & 0.8698 & 0.0000 & -0.0651 \end{bmatrix}$$
$$\begin{bmatrix} 0.0284 & -0.2370 & 0.4692 & -0.2370 & 0.0284 \end{bmatrix}$$

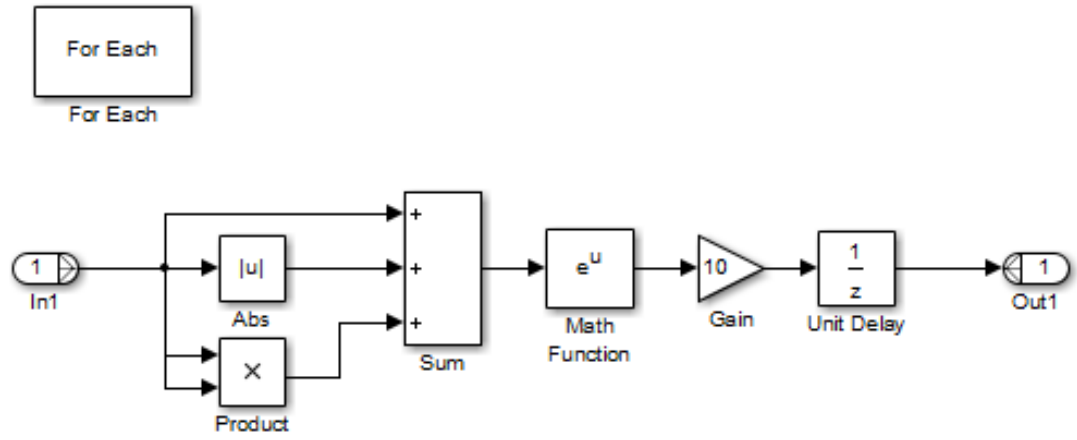
Improved Code Reuse Using For Each Subsystems

This example shows how you can improve code reuse when you have two or more identical For Each subsystems. Consider the following model, rtwdemo_foreachreuse.



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 inputs. 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:



To generate a single shared function for the three subsystems, the configuration of the partitioning they perform on their input signals must be the same. 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 then 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 press **Ctrl+B** 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 function has an input parameter `NumIters` that indicates the number of independent scalars that each For Each Subsystem is to processes. 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 block. Again, pressing **Ctrl+B** to generate the code provides code reuse of a single function.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

See Also

`For Each`

"Repeat an Algorithm Using a For Each Subsystem"

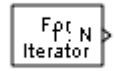
Introduced in R2010a

For Iterator

Repeatedly execute contents of subsystem at current time step until iteration variable exceeds specified iteration limit

Library

Ports & Subsystems



Description

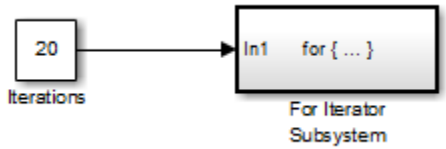
The For Iterator block, when placed in a subsystem, repeatedly executes the contents of the subsystem at the current time step until an iteration variable exceeds a specified iteration limit. You can use this block to implement the block diagram equivalent of a `for` loop in the C programming language.

The output of a For Iterator subsystem can not be a function-call signal. Simulink software will display an error message if the simulation is run or the diagram updated.

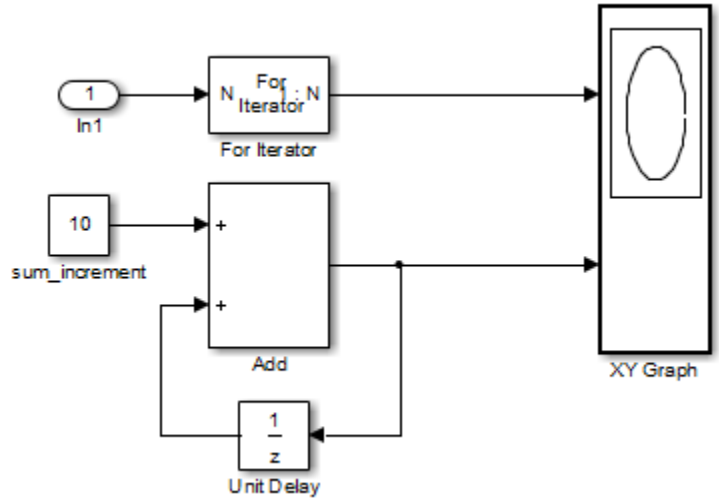
The block's parameter dialog allows you to specify the maximum value of the iteration variable or an external source for the maximum value and an optional external source for the next value of the iteration variable. If you do not specify an external source for the next value of the iteration variable, the next value is determined by incrementing the current value:

$i_{n+1} = i_n + 1$

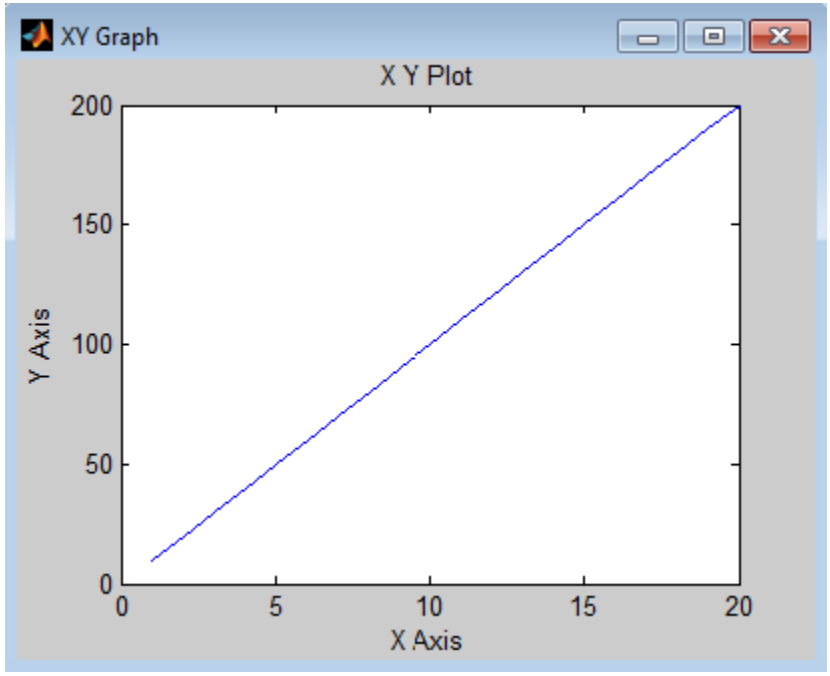
Suppose that you have the following model:



Over 20 iterations, the `For Iterator` block increments a value by 10 at each time step:



The following figure shows the result.



The For Iterator subsystem in this example is equivalent to the following C code.

```
sum = 0;
iterations = 20;
sum_increment = 10;
for (i = 0; i < iterations; i++) {
 sum = sum + sum_increment;
}
```

Note Placing a For Iterator block in a subsystem makes it an atomic subsystem if it is not already an atomic subsystem.

Data Type Support

The following rules apply to the data type of the number of iterations (N) input port:

- The input port accepts data of mixed numeric types.

- If the input port value is noninteger, it is first truncated to an integer.
- Internally, the input value is cast to an integer of the type specified for the iteration variable output port.
- If no output port is specified, the input port value is cast to type `int32`.
- If the input port value exceeds the maximum value of the output port's type, it is truncated to that maximum value.

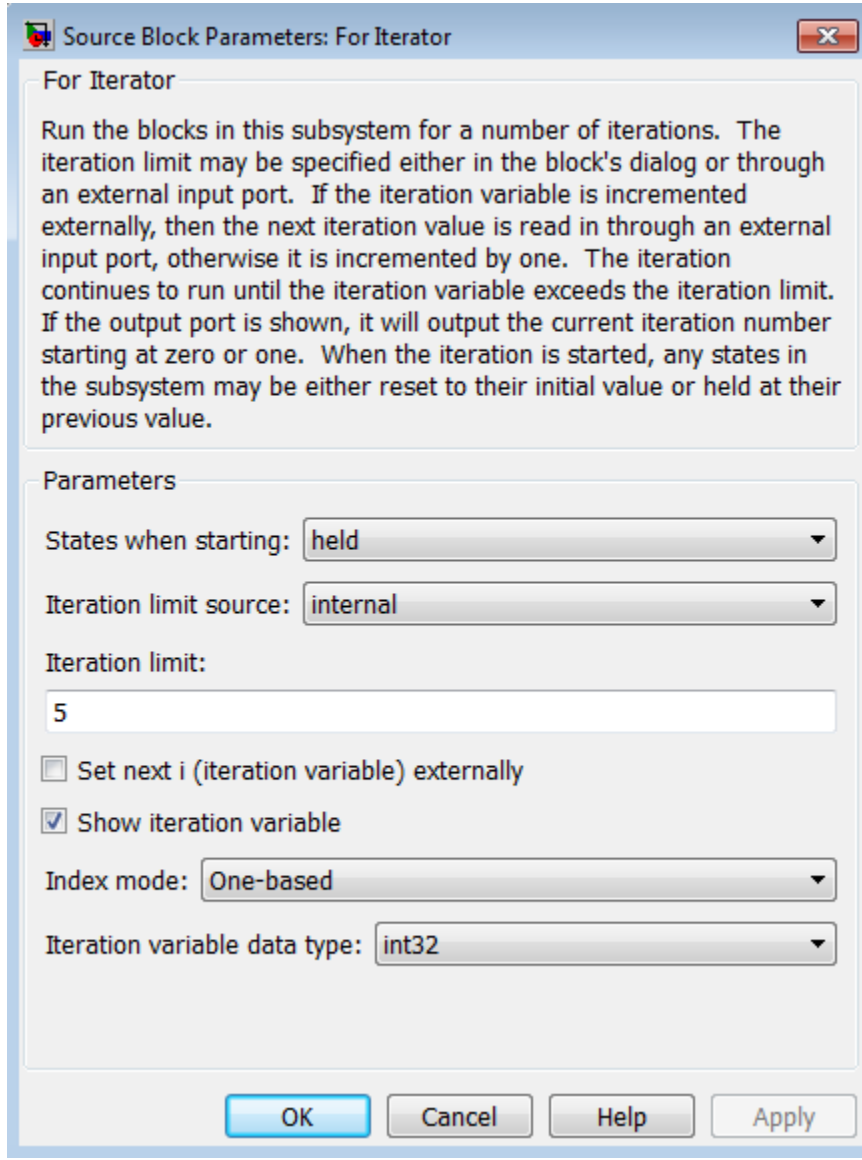
Data output for the iterator value can be selected as `double`, `int32`, `int16`, or `int8` in the block parameters dialog box.

The following rules apply to the iteration variable input port:

- It can appear only if the iteration variable output port is enabled.
- The data type of the iteration variable input port is the same as the data type of the iteration variable output port.

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

Parameters 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 a Custom Storage Class from the Simulink Package Using Data Objects" in the Embedded Coder documentation.

Set next i (iteration variable) externally

This option can be selected only if you select the **Show iteration variable** option. If you select this option, the For Iterator block displays an additional input for connecting an external iteration variable source. The value of the input at the current iteration is used as the value of the iteration variable at the next iteration.

Show iteration variable

If you select this check box, the For Iterator block outputs its iteration value.

Index mode

If you set this field to `Zero-based`, the iteration number starts at zero. If you set this field to `One-based`, the iteration number starts at one.

Iteration variable data type

Set the type for the iteration value output from the iteration number port to `double`, `int32`, `int16`, or `int8`.

Characteristics

Direct Feedthrough	No

Sample Time	Inherited from driving blocks
Scalar Expansion	No
Dimensionalized	No
Zero Crossing	No

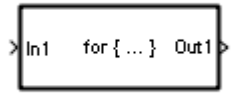
Introduced before R2006a

For Iterator Subsystem

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 “Use 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”.

Characteristics

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

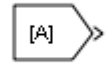
Introduced before R2006a

From

Accept input from Goto block

Library

Signal Routing

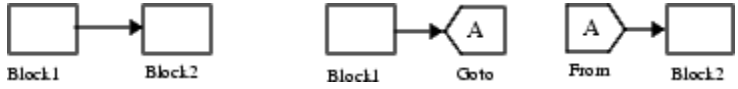


Description

The From block accepts a signal from a corresponding Goto block, then passes it as output. The data type of the output is the same as that of the input from the Goto block. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them. To associate a Goto block with a From block, enter the Goto block's tag in the **Goto Tag** parameter.

A From block can receive its signal from only one Goto block, although a Goto block can pass its signal to more than one From block.

This figure shows that using a Goto block and a From block is equivalent to connecting the blocks to which those blocks are connected. In the model at the left, Block1 passes a signal to Block2. That model is equivalent to the model at the right, which connects Block1 to the Goto block, passes that signal to the From block, then on to Block2.



The visibility of a Goto block tag determines the From blocks that can receive its signal. For more information, see Goto and Goto Tag Visibility. The block indicates the visibility of the Goto block tag:

- A local tag name is enclosed in brackets ([]).
- A scoped tag name is enclosed in braces ({}).
- A global tag name appears without additional characters.

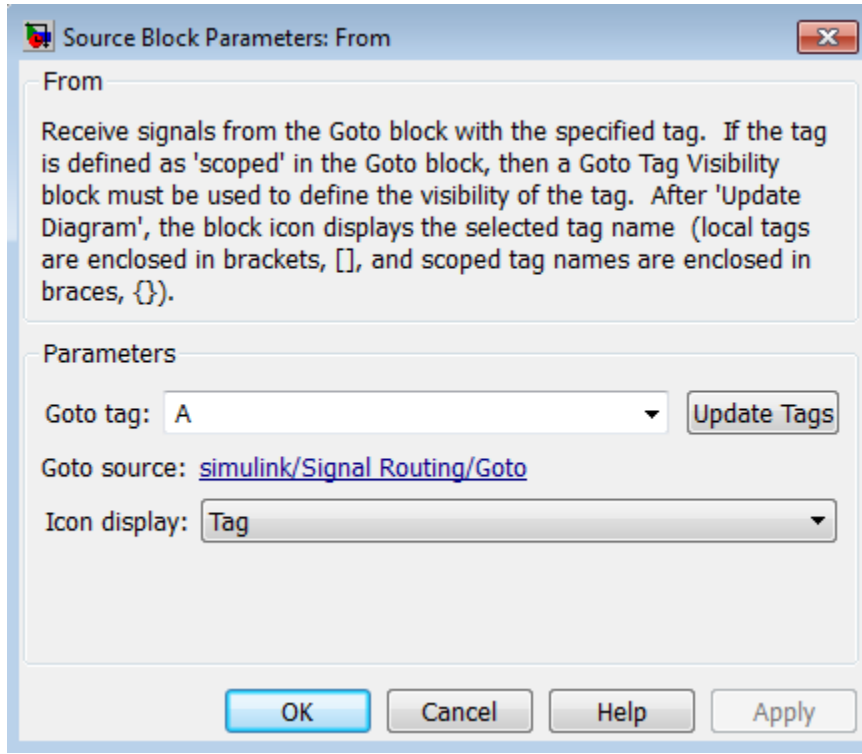
The From block supports signal label propagation.

Data Type Support

The From block outputs real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

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

Parameters and Dialog Box



Goto Tag

The tag of the Goto block that forwards its signal to this From block.

To change the tag, select a new tag from this control's drop-down list. The drop-down list displays the Goto tags that the From block can currently see. An item labeled `<More Tags...>` appears at the end of the list the first time you display the list in a Simulink session. Selecting this item causes the block to update the tags list to include the tags of Goto blocks residing in library subsystems referenced by the model containing this From block. Simulink software displays a progress bar while building the list of library tags. Simulink software saves the updated tags list for the duration of the Simulink session or until the next time you select the adjacent **Update Tags** button. You need to update the tags list again in the current session only if the libraries referenced by the model have changed since the last time you updated the list.

If you use multiple `From` and `Goto Tag Visibility` blocks to refer to the same Goto tag, you can simultaneously rename the tag in all of the blocks. Use the **Rename All** button in the `Goto` block dialog box. To find the `Goto` block, use the **Goto Source** hyperlink.

Update Tags

Updates the list of tags visible to this From block, including tags residing in libraries referenced by the model containing this From block.

Goto Source

Path of the Goto block connected to this From block. Clicking the path displays and highlights the Goto block.

Icon Display

Specifies the text to display on the From block's icon. The options are the block's tag, the name of the signal that the block represents, or both the tag and the signal name.

Examples

The following models show how to use the `From` block:

- `sldemo_auto_climatecontrol`
- `sldemo_hardstop`

Characteristics

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

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

See Also

`Goto`

Introduced before R2006a

From File

Load data from MAT-file

Library

Sources

Description

The From File block loads data from a MAT-file to a model and outputs the data as a signal. The data is a sequence of samples. Each sample consists of a time stamp and an associated data value. The data can be in array format or MATLAB `timeseries` format.

The From File block icon shows the name of the MAT-file that supplies the data to the block.

You can have multiple From File blocks that load from the same MAT-file.

The supported MAT-file versions are Version 7.0 or earlier and Version 7.3. The From File block incrementally loads data from Version 7.3 files.

You can specify how the data is loaded, including:

- Sample time
- How to handle data for missing data points
- Whether to use zero-crossing detection

For more information, see “Load Data Using the From File Block”.

Code Generation Requirements

For a From File block, generating code that builds ERT or GRT targets or uses SIL or PIL simulation modes requires that:

- The MAT-file contains a nonempty, finite, real matrix with at least two rows.
  - Use a data type of `double` for the matrix.
  - Do not include any NaN, Inf, or -Inf elements in the matrix.
- In the From File block parameters dialog box:
  - Set the **Data extrapolation before first data point** and **Data extrapolation after last data point** parameters to `Linear extrapolation`.
  - Set the **Data interpolation within time range** parameter to `Linear interpolation`.
  - Clear the **Enable zero-crossing detection** parameter.

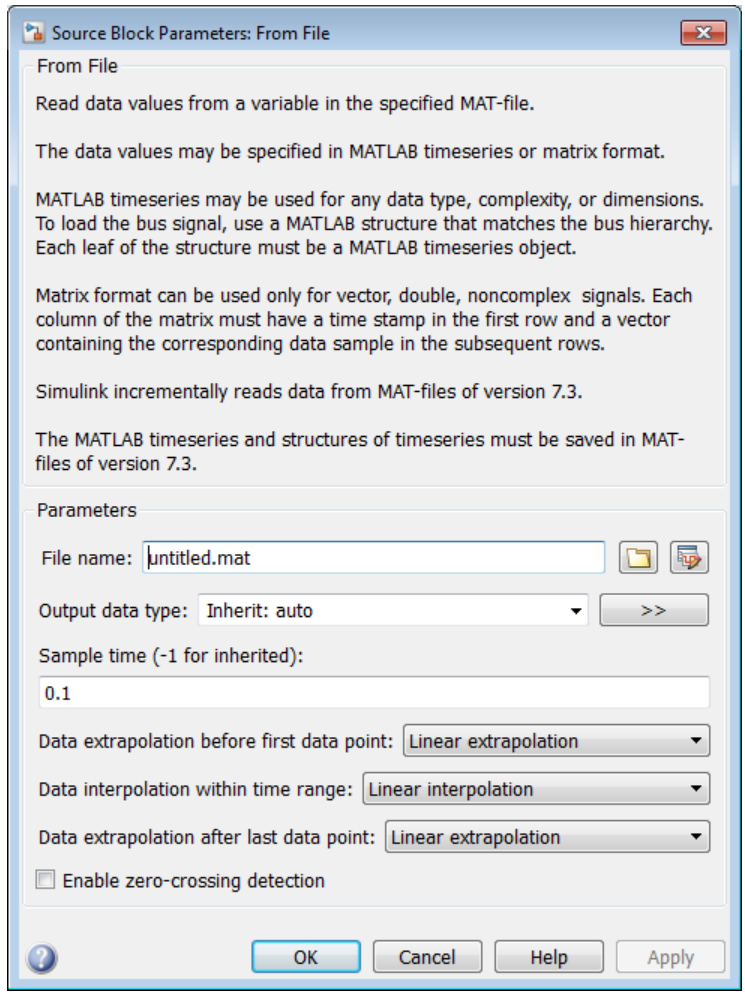
Data Type Support

For MATLAB `timeseries` data, the From File block loads real or complex signal data of any data type that Simulink supports. Fixed-point data cannot have a word length that exceeds 32 bits. The From File block supports loading nonvirtual bus signals in `timeseries` format.

For array data, the From File block loads only double signal data.

For more information, see “Create Data for a From File Block”.

Parameters and Dialog Box



File name

The path or file name of the MAT-file that contains the input data. Specify a path or file name in one of these ways:

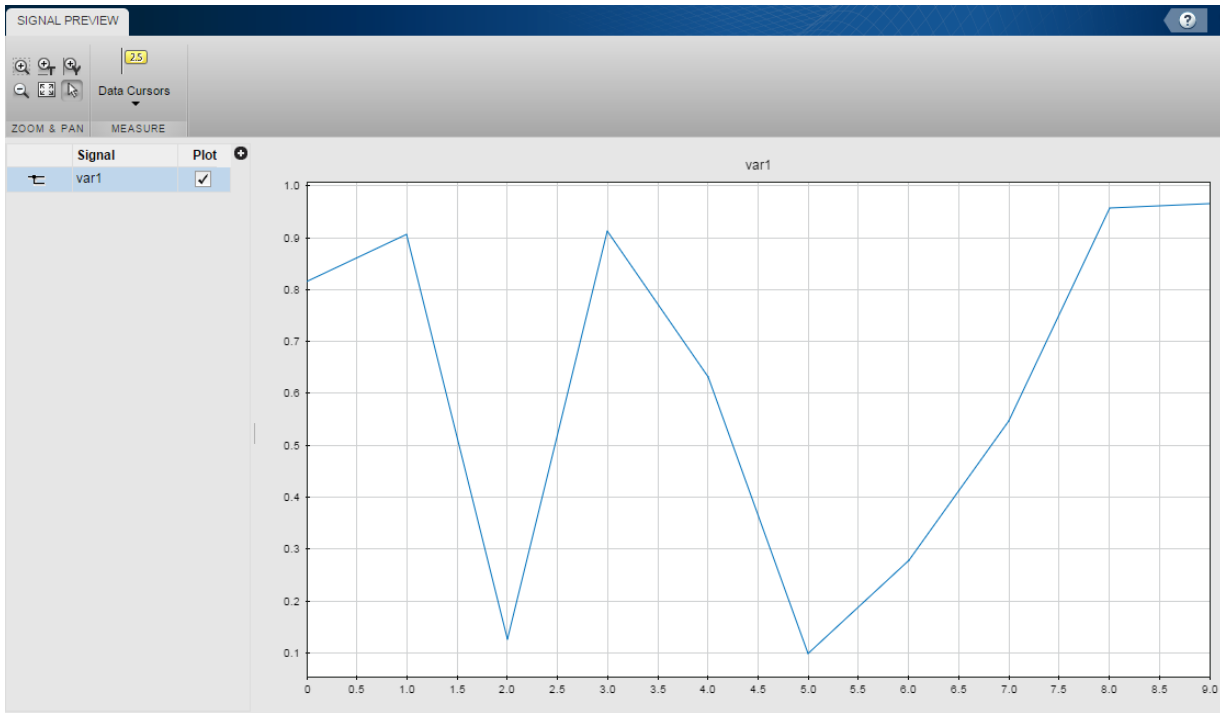
- Browse to a folder that contains a valid MAT-file.

On UNIX® systems, the path name can start with a tilde (~) character, which means your home folder.

- Enter the path for the file in the text box.

The default file name is `untitled.mat`. If you specify a file name without path information, Simulink loads the file in the current folder or on the MATLAB path. (To determine the current folder, at the MATLAB command prompt enter `pwd`.)

To preview the data in the Signal Preview window, click the view button  .



Use the Signal Preview window to plot and inspect data.

1. To plot the signal, select the check box next to the signal. If the format is a bus, click the expander ▸ to see and select the elements of the bus.

2 Explore the plots by using the **Measure** and **Zoom & Pan** sections of the toolstrip.

- In the **Measure** section, use the **Data Cursors** button to display one or two cursors for the plot. These cursors display the T and Y values of a data point in the plot. To view a data point, click a point on the line.
- In the **Zoom & Pan** section, select how you want to zoom and pan the signal plots. Zooming is only for the selected axis.

Zoom Action	Button
Zoom in along the time and data value axes.	
Zoom in along the time axis. Click the button. On the graph, drag the cursor to select an area to enlarge.	
Zoom in along the data value axis. Click the button. On the graph, drag the cusor to select an area to enlarge.	
Zoom out from the graph.	
Fit the plot to the graph. Click the button. Click the graph to enlarge the plot to fill the graph.	
Pan the graph up, down, left, or right. Click the button. On the graph, drag the cursor to the area of the graph that you want to view.	

Command-Line Information

Parameter: `FileName`
Type: string
Default: `'untitled.mat'`

Output data type

The data type for the data that the From File block outputs. For nonbus types, you can use `Inherit: auto` to skip any data type verification. If you specify an output data type, then the From File block verifies that the data in the file matches the specified data type. For more information, see “Control Signal Data Types”.

- `Inherit: auto` — Default
- `double`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `fixdt(1,16,0)` — Data type is fixed point (1,16,0).
- `fixdt(1,16,2^0,0)` — Data type is fixed point (1,16,2^0,0).
- `Enum: <class_name>` — Data type is enumerated.
- `Bus: <bus_object>` — Data type is a bus object. For details, see the "Using Bus Data" section.
- `<data type expression>` — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

If you set **Output data type** as a bus object, the bus object must be available when you compile the model. For each signal in bus data, the From File block verifies that data type, dimension, and complexity are the same for the data and for the bus object.

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see "Control Signal Data Types".

- `Inherit` — Inheritance rule for data types. Selecting `Inherit` enables a second menu and text box. (Default)
- `Built in` — Built-in data types. Selecting `Built in` enables a second menu and text box. Select one of the following choices:
  - `double` — Default

- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `Fixed point` — Fixed-point data types
- `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
- `Bus` — Bus object. Enables a **Bus object** parameter, where you enter bus object name for defining the structure of the bus. If you create or change a bus object, open the Simulink Bus Editor by clicking **Edit**. For details, see "Manage Bus Objects with the Bus Editor".
- `Expression` — Expression that evaluates to a data type. Selecting `Expression` enables a second menu and text box to the right, where you enter the expression. Do not specify a bus object as the expression.

Sample time

The sample period and offset.

The From File block loads data from a MAT-file, using a sample time that either:

- You specify for the From File block.
- The From File block inherits from the blocks into which the From File block feeds data.

The default sample time is `0`, which specifies a continuous sample time. The MAT-file is loaded at the base (fastest) rate of the model. For details, see "Specify Sample Time".

Command-Line Information

Parameter: `SampleTime`
Type: string
Default: `'0'`

Data extrapolation before first data point

Extrapolation method for a simulation time hit that occurs before the initial time stamp in the MAT-file. Choose one of the following extrapolation methods.

Method	Description

`Linear extrapolation`	(Default)
	
	If the MAT-file contains only one sample, then the From File block outputs the corresponding data value.
	
	If the MAT-file contains more than one sample, then the From File block linearly extrapolates using the first two samples:
	
	• For `double` data, linearly extrapolates the value using the first two samples
	• For `Boolean` data, outputs the first data value
	• For a built-in data type other than `double` or `Boolean`, the From File block:
	  • Upcasts the data to `double`
	  • Performs linear extrapolation (as described for `double` data)
	  • Downcasts the extrapolated data value to the original data type
	
	You cannot use the `Linear extrapolation` option with enumerated (`enum`) data. All signals in a bus use the same extrapolation setting. If any signal in a bus uses `enum` data, then you cannot use the `Linear extrapolation` option.

`Hold first value`	Uses the first data value in the file
`Ground value`	Uses a value that depends on the data type of MAT-file sample data values: • Fixed-point data types — uses the ground value • Numeric types other than fixed point — uses `0` • `Boolean` — uses `false`

Method	Description
	• Enumerated data types — uses default value

Command-Line Information
Parameter: `ExtrapolationBeforeFirstDataPoint`
Type: string
Value: `'Linear extrapolation' | 'Hold first value' | 'Ground value'`
Default: `'Linear extrapolation'`

Data interpolation within time range

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the MAT-file. Choose one of the following interpolation methods.

Method	Description

`Linear interpolation`	(Default)
	
	The From File block interpolates using the two corresponding MAT-file samples:
	
	• For `double` data, linearly interpolates the value using the two corresponding samples
	• For `Boolean` data, uses `false` for the first half of the sample and `true` for the second half.
	• For a built-in data type other than `double` or `Boolean`, the From File block:
	  • Upcasts the data to `double`
	  • Performs linear interpolation, as described for `double` data
	  • Downcasts the interpolated value to the original data type
	
	You cannot use the `Linear interpolation` option with enumerated (`enum`) data. All signals in a bus use the same interpolation setting. If any signal in a bus uses `enum` data, then you cannot use the `Linear interpolation` option.

Method	Description
`Zero order hold`	Uses the data from the first of the two samples

Command-Line Information

Parameter: `InterpolationWithinTimeRange`
Type: string
`'Linear interpolation' | 'Zero order hold'`
Default: `'Linear interpolation'`

Data extrapolation after last data point

The extrapolation method for a simulation time hit that occurs after the last time stamp in the MAT-file. Choose one of the following extrapolation methods.

Method	Description

`Linear extrapolation`	(Default)
	
	If the MAT-file contains only one sample, then the From File block outputs the corresponding data value.
	
	If the MAT-file contains more than one sample, then the From File block linearly extrapolates using data values of the last two samples:
	
	• For `double` data, extrapolates the value using the last two samples.
	• For `Boolean` data, outputs the first data value.
	• For built-in data types other than double or `Boolean`:
	
	  • Upcasts the data to `double`
	  • Performs linear extrapolation, as described for `double` data
	  • Downcasts the extrapolated value to the original data type
	
	You cannot use the `Linear extrapolation` option with enumerated (`enum`) data. All signals in a bus use the same

Method	Description
	extrapolation setting. 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: • 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

Command-Line Information

Parameter: `ExtrapolationAfterLastDataPoint`
Type: string
`'Linear extrapolation' | 'Hold last value' | 'Ground value'`
Default: `'Linear extrapolation'`

Enable zero-crossing detection

Enables zero-crossing detection.

The "Zero-Crossing Detection" parameter applies only if the **Sample time** parameter is set to `0` (continuous).

Simulink uses a technique known as zero-crossing detection to locate a discontinuity in time stamps, without resorting to excessively small time steps. "Zero-crossing" represents a discontinuity.

For the From File block, zero-crossing detection occurs only at time stamps in the file. Simulink examines only the time stamps, not the data values.

For bus signals, Simulink detects zero-crossings across all leaf bus elements.

If the input array contains duplicate time stamps (more than one entry with the same time stamp), Simulink detects a zero crossing at those time stamps. For example, suppose that the input array has this data:

```
time:     0 1 2 2 3
```

```
signal:   2 3 4 5 6
```

At time 2, there is a zero crossing from the input signal discontinuity.

For nonduplicate time stamps, zero-crossing detection depends on the settings of the following parameters:

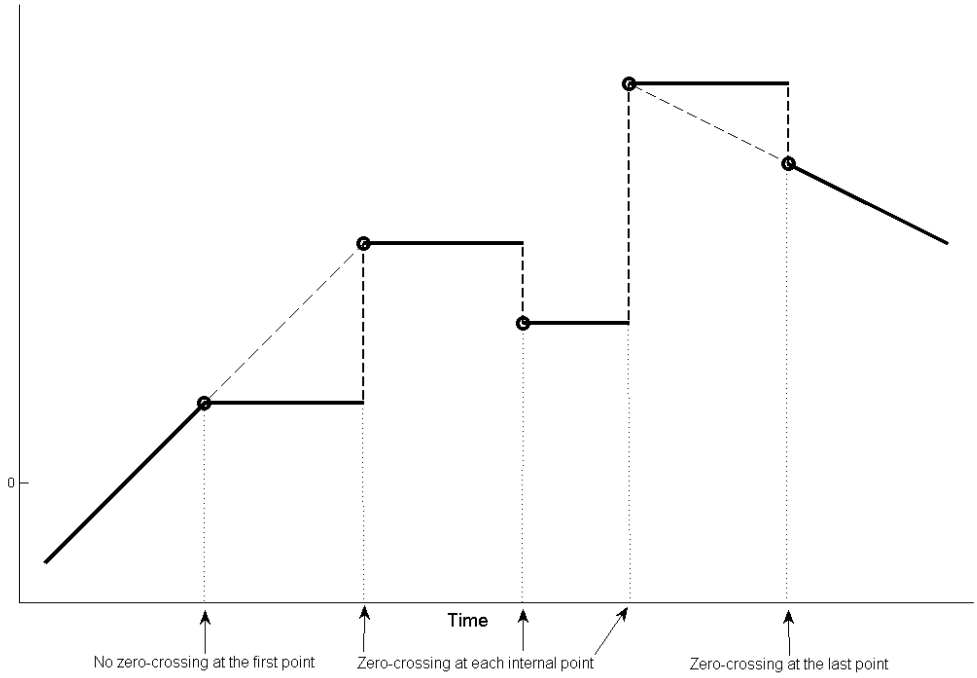
- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

The From File block determination of when zero-crossing occurs depends on the time stamp.

Time Stamp	Setting
First	**Data extrapolation before first data point** is set to `Ground value`.
Between first and last	**Data interpolation within time range** is set to `Zero-order hold`.
Last	One or both of these settings occur: • **Data extrapolation after last data point** is set to `Ground value`. • **Data interpolation within time range** is set to `Zero-order hold`.

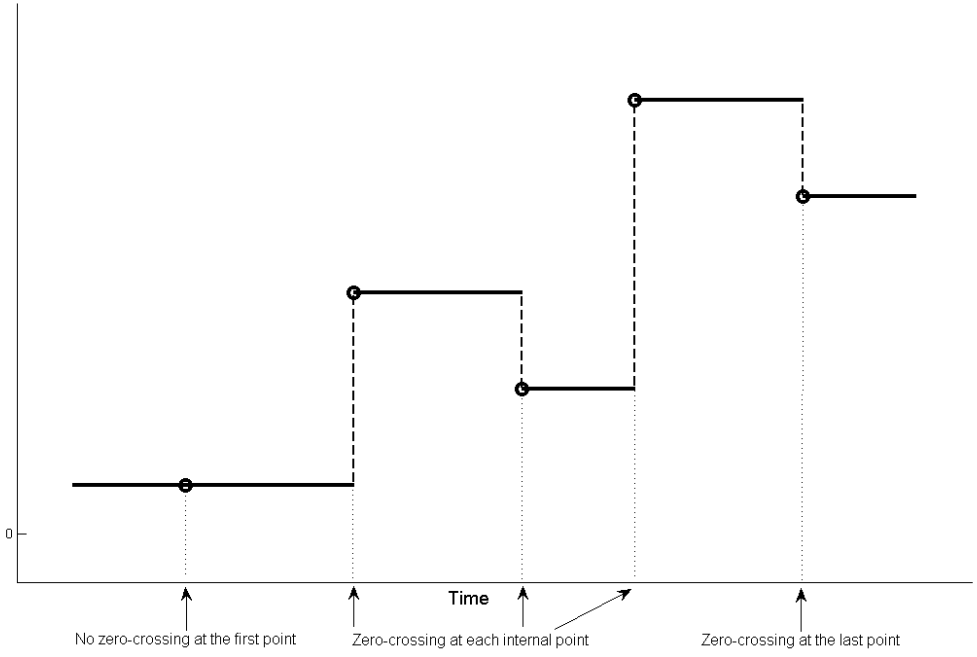
This figure illustrates zero-crossing detection for data accessed by a From File block that has the following settings:

- **Data extrapolation before first data point** — `Linear extrapolation`
- **Data interpolation within time range** (for internal points) — `Zero order hold`
- **Data extrapolation after last data point** — `Linear extrapolation`



This figure is another illustration of zero-crossing detection for data accessed by a From File block. The block has the following settings for the time stamps (points):

- **Data extrapolation before first data point** — `Hold first value`
- **Data interpolation within time range** — `Zero order hold`
- **Data extrapolation after last data point** — `Hold last value`



Examples

From File Block Loading Timeseries Data

Create a MATLAB `timeseries` object with time and signal values. Save the timeseries object to a MAT-file and load into a model using a From File block.

Create an array with the time and signal data with signal data for 10 time steps.

```
t = .1*(1:10);
d = .2*(1:10);
x = [t;d];
```

```
x =

    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000    0.8000    0.9
    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000    1.4000    1.6000    1.8
```

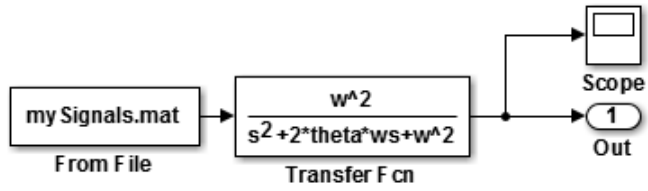
Create a MATLAB `timeseries` object.

```
ts = timeseries(x(2:end,:),x(1,:))
```

Save the `timeseries` object in a Version 7.3 MAT-file.

```
save('mySignals','ts','-v7.3')
```

Add a From File block and set the **File name** parameter of that block to `mySignals.mat`.



Simulate the model. The Scope block reflects the data loaded from the `mySignals.mat` file.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

"Create Data for a From File Block""Load Data Using the From File Block", "Load Signal Data That Uses Units", "Comparison of Signal Loading Techniques", `To FileFrom Workspace`, `To Workspace`, `From Spreadsheet`

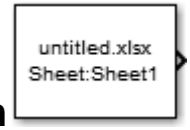
Introduced before R2006a

From Spreadsheet

Read data from spreadsheet

Library

Sources



Description

The `From Spreadsheet` block reads data from Microsoft Excel® (all platforms) or CSV (Microsoft Windows® platform with Microsoft Office installed only) spreadsheets and outputs the data as a signal.

The `From Spreadsheet` icon displays the spreadsheet file name and sheet name specified in the block **File name** and **Sheet name** parameters.

Storage Formats

The data that the `From Spreadsheet` block reads from a spreadsheet must be appropriately formatted.

For Microsoft Excel spreadsheets:

- The `From Spreadsheet` block interprets the first row as a signal name. If you do not specify a signal name, the `From Spreadsheet` block assigns a default one with the format `Signal #`, where # increments with each additional unnamed signal.
- The `From Spreadsheet` block interprets the first column as time. In this column, the time values must monotonically increase.
- The `From Spreadsheet` block interprets the remaining columns as signals.

This example shows an acceptably formatted Microsoft Excel spreadsheet. The first column is Time and the first row contains signal names. Each worksheet contains a signal group.

1	Time	DC In	Trigger	AC In
2	0	1	2	3
3	1	2	3	4
4	2	3	4	5
5	3	4	5	6
6	4	5	6	7
7	5	6	7	8
8	6	7	8	9
9	7	8	9	10
10	8	9	10	11
11	9	10	11	12
12	10	11	12	13
13	11	12	13	14
14	12	13	14	15
15	13	14	15	16
16	14	15	16	17
17	15	16	17	18
18				

Group1 | Group2 | Group3

For CSV text files (Microsoft platform with Microsoft Office installed only):

- The `From Spreadsheet` block interprets the first column as time. In this column, the time values must increase.
- The `From Spreadsheet` block interprets the remaining columns as signals.
- Each column must have the same number of entries.
- The `From Spreadsheet` block interprets each file as one signal group.

This example shows an acceptably formatted CSV file. The contents represent one signal group.

```
0,0,0,5,0
1,0,1,5,0
2,0,1,5,0
3,0,1,5,0
```

```
4,5,1,5,0
5,5,1,5,0
6,5,1,5,0
7,0,1,5,0
8,0,1,5,1
9,0,1,5,1
10,0,1,5,0
```

Block Behavior During Simulation

The `From Spreadsheet` block incrementally reads data from the spreadsheet during simulation.

The **Sample time** parameter specifies the sample time that the `From Spreadsheet` block uses to read data from the spreadsheet. For details, see "Parameters and Dialog Box" on page 1-765. The time stamps in the file must be monotonically nondecreasing.

For each simulation time hit for which the spreadsheet contains no matching time stamp, Simulink software interpolates or extrapolates to obtain the needed data using the selected method. For details, see "Simulation Time Hits That Have No Corresponding Spreadsheet Time Stamps" on page 1-762.

Simulation Time Hits That Have No Corresponding Spreadsheet Time Stamps

If the simulation time hit does not have a corresponding spreadsheet time stamp, the `From Spreadsheet` block output depends on:

- Whether the simulation time hit occurs before the first time stamp, within the range of time stamps, or after the last time stamp
- The interpolation or extrapolation methods that you select
- The data type of the spreadsheet data

For details about interpolation and extrapolation options, see the descriptions of the following parameters in "Parameters and Dialog Box":

- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

Sometimes the spreadsheet includes two or more data values that have the same time stamp. In such cases, the `From Spreadsheet` block action depends on when the simulation time hit occurs, relative to the duplicate time stamps in the spreadsheet.

For example, suppose that the spreadsheet contains this data. Three data values have a time stamp value of 2.

```
time stamps:    0 1 2 2 2 3 4
data values:    2 3 6 4 9 1 5
```

The table describes the `From Spreadsheet` block output.

Simulation Time, Relative to Duplicate Time Stamp Values in Spreadsheet	From Spreadsheet Block Action
Before the duplicate time stamps	Performs the same actions as when the time stamps are distinct, using the first of the duplicate time stamp values as the basis for interpolation. (In this example, the time stamp value is 6.)
At or after the duplicate time stamps	Performs the same actions as when the times stamps are distinct, using the last of the duplicate time stamp values as the basis for interpolation. (In this example, that time stamp value is 9.)

Rounding Mode

The `From Spreadsheet` block rounds positive and negative numbers toward negative infinity. This mode is equivalent to the MATLAB `floor` function.

Saturation on Integer Overflow

For data type conversion, the `From Spreadsheet` block deals with saturation overflow by wrapping to the appropriate value that the data type can represent. For example, the number 130 does not fit in a signed 8-bit integer and wraps to –126.

Code Generation Requirements

Simulating in accelerator, rapid accelerator, or model reference accelerator mode behaves the same way, and has the same requirements, as simulating in normal mode.

The `From Spreadsheet` block does not support generating code that involves building ERT or GRT targets, or using SIL or PIL simulation modes.

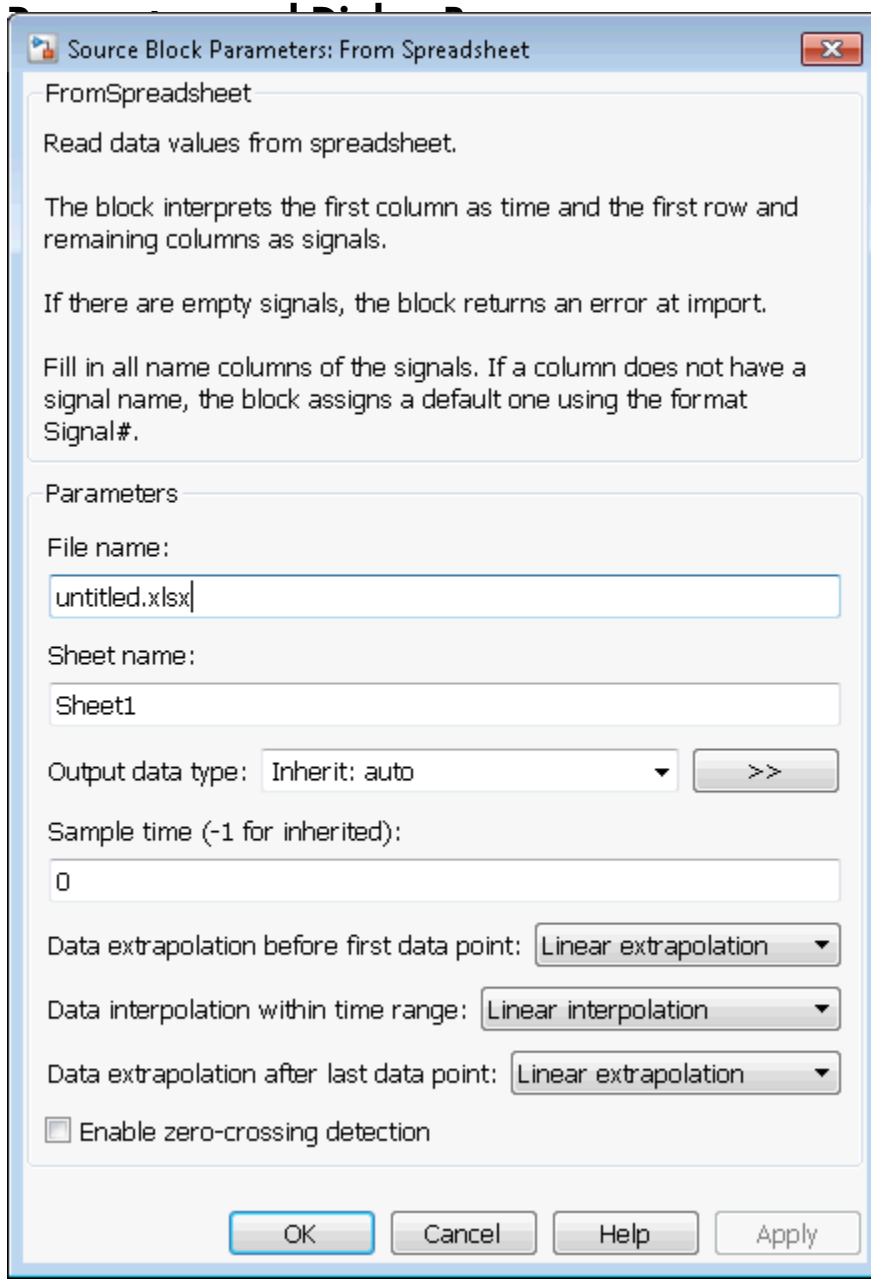
Data Type Support

The `From Spreadsheet` block supports these data types:

- Inherit: auto — Default
- double
- single
- int8
- uint8
- int16
- uint16
- int32
- uint32
- Boolean
- fixdt(1,16,0) — Data type is fixed point (1,16,0).
- fixdt(1,16,2^0,0) — Data type is fixed point (1,16,2^0,0).
- Enum: <class_name> — Data type is enumerated.
- `data type expression` — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

The `From Spreadsheet` block accepts data type specifications at a block level.

If you want to specify different data types for each signal, consider selecting **Output Data Type** > **Inherit: Auto**. This option resolves back signal data types using back propagation. For example, assume that there are two signals in the `From Spreadsheet` block, In1 and In2, which the block sends to ports that have int8 and Boolean data types. With back propagation, the block recasts In1 as int8 and In2 as boolean.


Source Block Parameters: From Spreadsheet
FromSpreadsheet
Read data values from spreadsheet.
The block interprets the first column as time and the first row and remaining columns as signals.
If there are empty signals, the block returns an error at import.
Fill in all name columns of the signals. If a column does not have a signal name, the block assigns a default one using the format Signal#.
Parameters
File name:
untitled.xlsx
Sheet name:
Sheet1
Output data type:
Inherit: auto
>>
Sample time (-1 for inherited):
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
OK
Cancel
Help
Apply

File name

Enter full path and file name of a spreadsheet file.

This block supports non-English full paths and file names only on Microsoft platforms.

Settings

Default: `untitled.xlsx`

Command-Line Information

`FileName`	string

Sheet name

Enter the name of the sheet in the **File name** spreadsheet. You can find this name on the sheet tab.

Settings

Default: `Sheet1`

Command-Line Information

`SheetName`	string

Output data type

The data type for the `From Spreadsheet` block output. The `From Spreadsheet` block accepts spreadsheets that contain many data types. However, the block reads the spreadsheet data type as doubles. It then outputs the data type according to the value of **Output data type**.

To allow the block to cast the output data type to match that of the receiving block, use `Inherit: auto`.

For more information, see "Control Signal Data Types".

Settings

Default: `Inherit: auto`

Cast output data type to match that of the receive block.

`Inherit: auto`

Cast output data type to match that of the receive block.

`double`

Output data type is `double`.

`single`

Output data type is `single`.

`int8`

Output data type is `int8`.

`uint8`

Output data type is `uint8`.

`int16`

Output data type is `int16`.

`uint16`

Output data type is `uint16`.

`int32`

Output data type is `int32`.

`uint32`

Output data type is `uint32`.

`boolean`

Output data type is `boolean`.

`fixdt(1,16,0)`

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

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

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

`Enum: <class name>`

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

`<data type expression>`

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

Command-Line Information

`OutDataTypeStr`	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>'`

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see “Control Signal Data Types”.

Settings

Default: `Inherit`

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

- `Inherit`

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

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

  - `double` — Default
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`
  - `uint32`
  - `boolean`
- `Fixed point` — Fixed-point data types
- `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
- `Expression` — Expression that evaluates to a data type. Selecting `Expression` enables a second menu/text box to the right, where you enter the expression.

Data type override

Specify data type override mode for this signal.

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.

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.

See Also

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

Scaling

Scaling

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

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**

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.

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.

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.

See Also

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

Sample time

The sample period and offset.

The From Spreadsheet block reads data from a spreadsheet using a sample time that either:

- You specify for the From Spreadsheet block
- The From Spreadsheet block inherits from the blocks into which the From Spreadsheet block feeds data

The default is `0`, which specifies a continuous sample time. The spreadsheet is read at the base (fastest) rate of the model. For details, see “Specify Sample Time”.

Command-Line Information

`SampleTime`	string — `{'0'}`

Data extrapolation before first data point

Extrapolation method that Simulink uses for a simulation time hit that is before the first time stamp in the spreadsheet. Choose one of the following extrapolation methods.

Method	Description

`Linear extrapolation`	(Default)
	
	If the spreadsheet contains only one sample, the `From Spreadsheet` block outputs the corresponding data value.
	
	If the spreadsheet contains more than one sample, the `From Spreadsheet` block linearly extrapolates using the first two samples:
	
	• For `double` data, linearly extrapolates the value using the first two samples
	• For `Boolean` data, outputs the first data value
	• For a built-in data type other than `double` or `Boolean`:
	  • Upcasts the data to `double`.
	  • Performs linear extrapolation (as described above for `double` data).
	  • Downcasts the extrapolated data value to the original data type.
	
	You cannot use the `Linear extrapolation` option with enumerated (`enum`) data.

`Hold first value`	Uses the first data value in the file
`Ground value`	Uses a value that depends on the data type of spreadsheet sample data values: • Fixed-point data types — Uses the ground value • Numeric types other than fixed–point — Uses `0` • `Boolean` — Uses `false` • Enumerated data types — Uses default value

Command-Line Information

`ExtrapolationBeforeFirstDataPoint`	string — `{'Linear extrapolation'} | 'Hold first value' | 'Ground value'`

Data interpolation within time range

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the spreadsheet. Choose one of the following interpolation methods.

Method	Description

`Linear interpolation`	(Default)
	
	The `From Spreadsheet` block interpolates using the two corresponding spreadsheet samples:
	
	• For `double` data, linearly interpolates the value using the two corresponding samples
	• For `Boolean` data, uses `false` for the first half of the sample and `true` for the second half.
	• For a built-in data type other than `double` or `Boolean`:
	  • Upcasts the data to `double`.
	  • Performs linear interpolation (as described above for `double` data).
	  • Downcasts the interpolated value to the original data type.
	
	You cannot use the `Linear interpolation` option with enumerated (`enum`) data.

`Zero order hold`	Uses the data from the first of the two samples

Command-Line Information

`InterpolationWithinTimeRange`	string — `{'Linear interpolation'} | 'Zero order hold'`

Data extrapolation after last data point

The extrapolation method that Simulink uses for a simulation time hit that is after the last time stamp in the spreadsheet. Choose one of the following extrapolation methods.

Method	Description

`Linear extrapolation`	(Default)
	
	If the spreadsheet contains only one sample, the `From Spreadsheet` block outputs the corresponding data value.
	
	If the spreadsheet contains more than one sample, the `From Spreadsheet` block linearly extrapolates using data values of the last two samples:
	
	• For `double` data, extrapolates the value using the last two samples
	• For `Boolean` data, outputs the last data value
	• For a built-in data type other than `double` or `Boolean`:
	  • Upcasts the data to `double`.
	  • Performs linear extrapolation (as described above for `double` data).
	  • Downcasts the extrapolated value to the original data type.
	
	You cannot use the `Linear extrapolation` option with enumerated (`enum`) data.

`Hold last value`	Uses the last data value in the file
`Ground value`	Uses a value that depends on the data type of spreadsheet sample data values: • Fixed-point data types — uses the ground value • Numeric types other than fixed–point — uses `0` • `Boolean` — uses `false` • Enumerated data types — uses default value

Command-Line Information

`ExtrapolationAfterLastDataPoint`	string — `{'Linear extrapolation'}` | `'Hold last value'` | `'Ground value'`

Enable zero-crossing detection

Select to enable zero-crossing detection.

The "Zero-Crossing Detection" parameter applies only if the **Sample time** parameter is set to `0` (continuous).

Simulink uses a technique known as zero-crossing detection to locate accurately a discontinuity, without resorting to excessively small time steps. This section uses "zero-crossing" to represent discontinuities.

For the `From Spreadsheet` block, zero-crossing detection can only occur at time stamps in the file. Simulink examines only the time stamps, not the data values.

If the input array contains duplicate time stamps (more than one entry with the same time stamp), Simulink detects a zero crossing at that time stamp. For example, suppose that the input array has this data:

```
time:     0 1 2 2 3
signal:   2 3 4 5 6
```

At time 2, there is a zero crossing from the input signal discontinuity.

For data with nonduplicate time stamps, zero-crossing detection depends on the settings of the following parameters:

- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

The block applies the following rules when determining when:

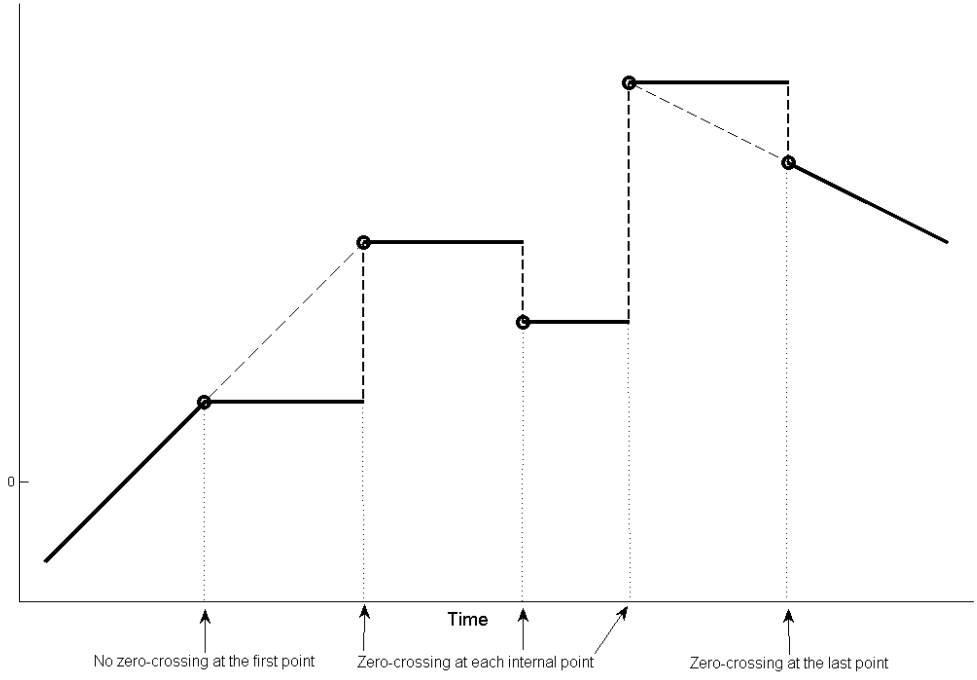
- Zero-crossing occurs for the first time stamp.
- For time stamps between the first and last time stamp.
- For the last time stamp.

Time Stamp	When Zero-Crossing Detection Occurs
First	**Data extrapolation before first data point** is set to `Ground value`.
Between first and last	**Data interpolation within time range** is set to `Zero-order hold`.

Time Stamp	When Zero-Crossing Detection Occurs
Last	One or both of these settings occur: • **Data extrapolation after last data point** is set to `Ground value`. • **Data interpolation within time range** is set to `Zero-order hold`.

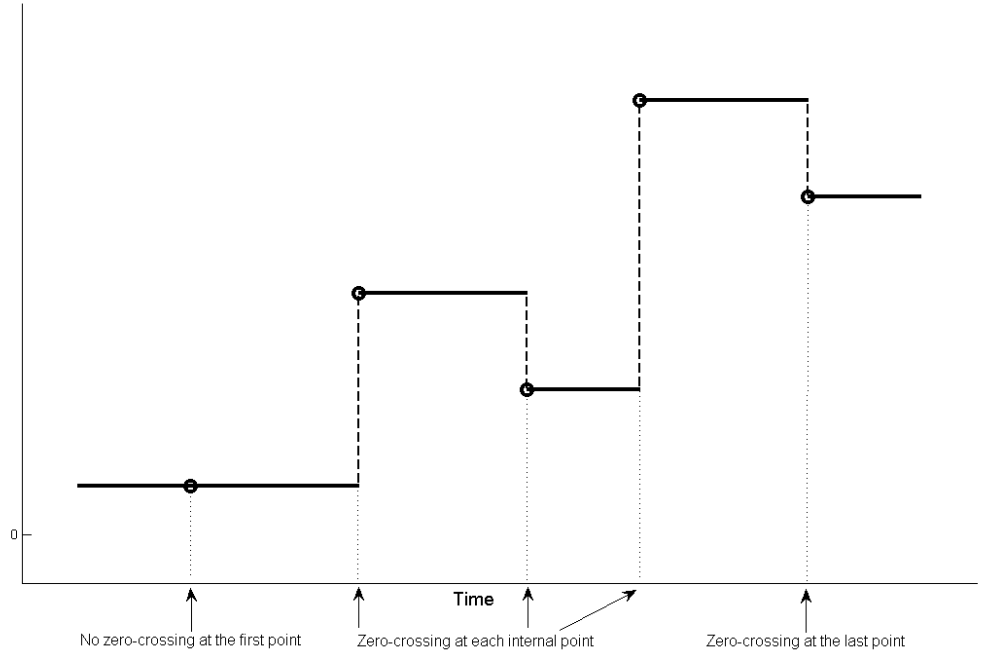
The following figure illustrates zero-crossing detection for data accessed by a `From Spreadsheet` block that has the following settings:

- **Data extrapolation before first data point** — `Linear extrapolation`
- **Data interpolation within time range** (for internal points) — `Zero order hold`
- **Data extrapolation after last data point** — `Linear extrapolation`



The following figure is another illustration of zero-crossing detection for data accessed by a `From Spreadsheet` block. The block has the following settings for the time stamps (points):

- **Data extrapolation before first data point** — `Hold first value`
- **Data interpolation within time range** — `Zero order hold`
- **Data extrapolation after last data point** — `Hold last value`



Command-Line Information

`ZeroCross`	string — `'off' | {'on'}`

Characteristics

Data Types	Specified in “Data Type Support” on page 1-764

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

See Also

`From File`

Related Examples

- "Power Window"

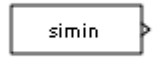
Introduced in R2015b

From Workspace

Load signal data from workspace

Library

Sources



Description

The From Workspace block reads signal data from a workspace and outputs the data as a signal.

The From Workspace icon displays the expression specified in the **Data** parameter. For details about how Simulink software evaluates this expression, see "Symbol Resolution".

You can specify how the data is loaded, including sample time, how to handle data for missing data points, and whether to use zero-crossing detection. For more information, see "Load Data Using the From Workspace Block".

Specifying the Workspace Data

In the Block Parameters dialog box of the From Workspace block, in the **Data** parameter, specify the workspace data to load. Specify a MATLAB expression (for example, the name of a variable in the MATLAB workspace) that evaluates to one of the following:

- A MATLAB timeseries object
- A structure of MATLAB `timeseries` objects
- A structure, with or without time
- A two-dimensional matrix

For additional information, see "Create Data for a From Workspace Block".

Data Type Support

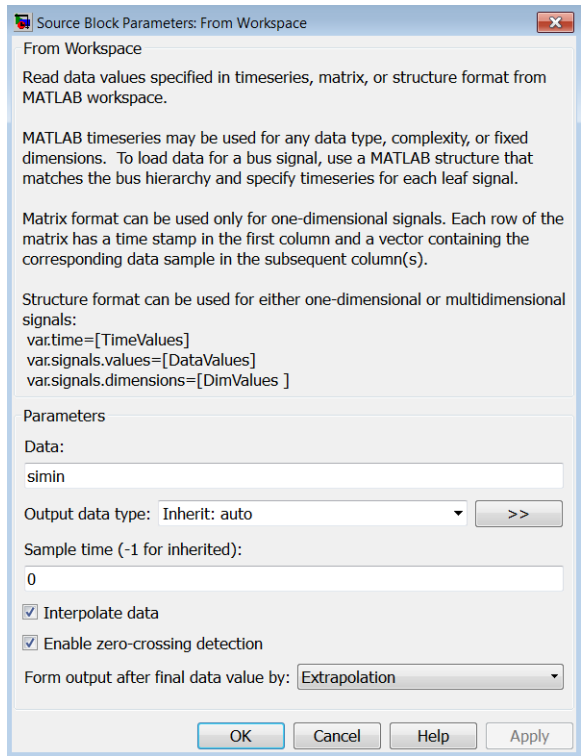
The From Workspace block accepts data from the workspace. The block outputs real or complex signals of any type that Simulink supports, including fixed-point and enumerated data types.

The From Workspace block also accepts a bus object as a data type. To load bus data, use a structure of MATLAB `timeseries` objects. For details, see "Import Bus Data to Top-Level Input Ports".

Real signals of type `double` can be in any data format that the From Workspace block supports. For complex signals and real signals of a data type other than `double`, use any format except `Array`.

For additional information, see "Create Data for a From Workspace Block".

Parameters and Dialog Box



Data

In the **Data** parameter, specify the workspace data to load. Specify a MATLAB expression (for example, the name of a variable in the MATLAB workspace) that evaluates to one of the following:

- A MATLAB timeseries object
- A structure of MATLAB `timeseries` objects
- A structure, with or without time
- A two-dimensional matrix

For additional information, see “Specify the Workspace Data”.

Command-Line Information

Parameter: `VariableName`
Type: string
Default: `'simin'`

Output data type

The required data type for the data for the workspace variable that the From Workspace block loads. For non-bus types, to skip any data type verification, you can use `Inherit: auto`. For more information, see "Control Signal Data Types".

To load bus data, use a structure of MATLAB timeseries objects. For details, see "Import Bus Data to Top-Level Input Ports".

- `Inherit: auto` — Default.
- `double`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `fixdt(1,16,0)` — Data type is fixed point (1,16,0).
- `fixdt(1,16,2^0,0)` — Data type is fixed point (1,16,2^0,0).
- `Enum: <class_name>` — Data type is enumerated, for example, `Enum: Basic Colors`.
- `Bus: <bus_object>` — Data type is a bus object.
- `<data type expression>` — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

Command-Line Information

Parameter: `OutDataTypeStr`
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>'

Default: 'Inherit: auto'

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see "Control Signal Data Types".

- `Inherit` — Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right. (Default)
- `Built in` — Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:
  - `double` — Default
  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`
  - `uint32`
  - `boolean`
  - `Fixed point` — Fixed-point data types
  - `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
  - `Bus` — Bus object. Selecting `Bus` enables a **Bus object** parameter, where you enter the name of a bus object that you want to use to define the structure of the bus. To create or change a bus object, click **Edit** (to the right of the **Bus object** field) to open the Simulink Bus Editor. For details, see "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 loaded data. For details, see "Specify Sample Time".

Command-Line Information

Parameter: `SampleTime`
Type: string
Default: `'0'`

Interpolate data

To have the block linearly interpolate at time hits for which no corresponding workspace data exist, select this option. Otherwise, the current output equals the output at the most recent time for which data exists.

The From Workspace block interpolates by using the two corresponding workspace samples:

- For `double` data, linearly interpolates the value by using the two corresponding samples
- For `Boolean` data, uses `false` for the first half of the time between two time values and `true` for the second half
- For a built-in data type other than `double` or `Boolean`:
  - Upcasts the data to `double`
  - Performs linear interpolation (as described for `double` data)
  - Downcasts the interpolated value to the original data type

You cannot use linear interpolation with enumerated (`enum`) data.

The block uses the value of the last known data point as the value of time hits that occur after the last known data point.

To determine the block output after the last time hit for which workspace data is available, combine the settings of these parameters:

- **Interpolate data**
- **Form output after final data value by**

For details, see the **Form output after final data value by** parameter.

Command-Line Information

Parameter: `Interpolate`
Type: string
`'off' | 'on'`
Default: `'on'`

Enable zero-crossing detection

If you select the **Enable zero-crossing detection** parameter, then when the input array contains multiple entries for the same time hit, Simulink detects a zero crossing. For example, suppose that the input array has this data:

```
time:     0 1 2 2 3
signal:   2 3 4 5 6
```

At time 2, there is a zero crossing from input signal discontinuity. For more information, see "Zero-Crossing Detection".

For bus signals, Simulink detects zero crossings across all leaf bus elements.

Command-Line Information

Parameter: `ZeroCross`
Type: string
`'off' | 'on'`
Default: `'on'`

Form output after final data value by

To determine the block output after the last time hit for which workspace data is available, combine the settings of these parameters:

- **Interpolate data**
- **Form output after final data value by**

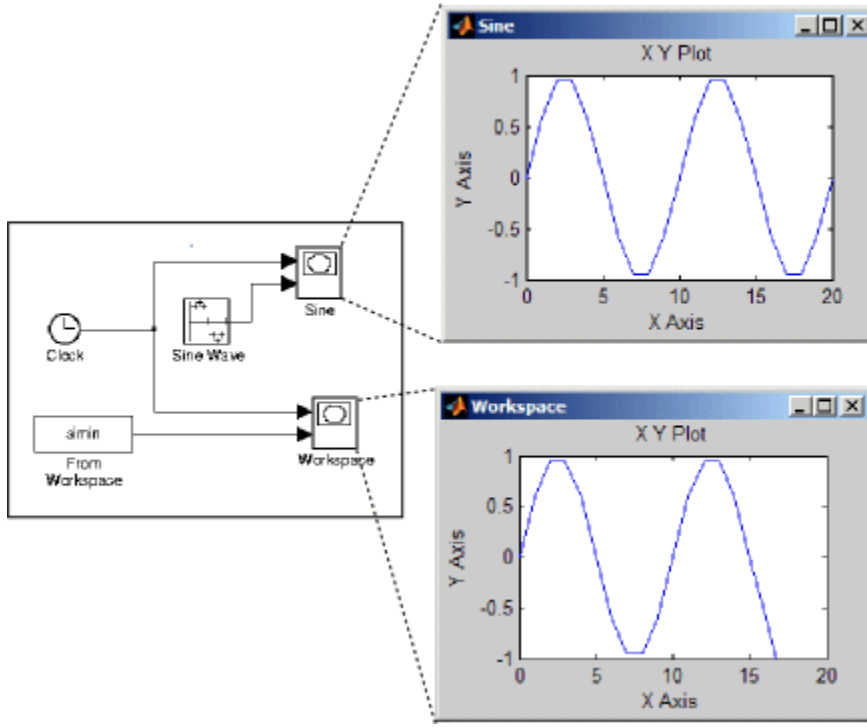
This table lists the block output, based on the values of the two options.

Setting for Form Output After Final Data Value By	Setting for Interpolate Data	Block Output After Final Data
Extrapolation	On	Extrapolated from final data value
	Off	Error
Setting to zero	On	Zero
	Off	Zero
Holding final value	On	Final value from workspace
	Off	Final value from workspace
Cyclic repetition	On	Error
	Off	Repeated from workspace if the workspace data is in structure-without-time format. Error otherwise.

For example, the block uses the last two known data points to extrapolate data points that occur after the last known point if you:

- Set **Interpolate data** to `Extrapolation`.
- Select **Form output after final data value by**.

Consider this model.



The From Workspace block reads data from the workspace. The data consists of the output of the Simulink Sine block sampled at one-second intervals. The workspace contains the first 16 samples of the output. The top and bottom X-Y plots display the output of the Sine Wave and From Workspace blocks, respectively, from 0 to 20 seconds. The straight line in the output of the From Workspace block reflects the linear extrapolation of missing data points at the end of the simulation.

Command-Line Information

Parameter: `OutputAfterFinalValue`
Type: string
`'Extrapolation' | 'Setting to zero' | 'Holding final value' | 'Cyclic repetition'`
Default: `'Extrapolation'`

Examples

With the From Workspace block, you can read 1-D and 2-D signals into Simulink. The From Workspace block does not read n-D signals into Simulink.

Read 1-D Signals in Array Format

Create two signals `x` and `y` with a time vector `t`. Import the values into Simulink with an array.

1. In the MATLAB Command Window, enter:

   ```
   t = 0.2 * [0:49]';
   x = sin(t);
   y = 10*sin(t);
   ```

   The time vector must be a column vector.
2. Add a From Workspace block to your model.
3. For the From Workspace block, in **Data** parameter, enter the array `[t,x,y]`.

Read 1-D Signals in Structure Format

Create two signals `x` and `y` with a time vector `t`. Import the values into Simulink with a structure.

1. In the MATLAB Command Window, enter:

   ```
   t = 0.2 * [0:49]';
   x = sin(t);
   y = 10*sin(t);
   wave.time = t;
   wave.signals.values = [x,y];
   wave.signals.dimensions =2;
   ```

   The time vector must be a column vector. The `signals.dimensions` field for the signal is a scalar corresponding to the number of columns in the `signals.values` field.
2. Add a From Workspace block to your model.
3. In the From Workspace block parameters dialog box, in the **Data** field, enter the structure name.

Use Sample Time from Model

If you do not have a time vector, you can define the sample time in your model.

1. In the MATLAB Command Window, enter

   ```
   wave.time = [];
   ```

2. In the From Workspace block parameters dialog box, in the **Sample time** field, enter a time interval. For example, enter `0.2`. Clear the **Interpolate data** check box.
3. In the **Form output after final data value by**, select `Setting to zero`, `Holding final value`, or `Cyclic repetition`. Do not select `Extrapolation`.

Read 2-D Signals in Structure Format

To load 2-D signals from the MATLAB workspace into Simulink, you must have the signals in a structure format. This example creates a 10–by–10 matrix (2-D signal) by using the `magic` function, and then creates a 3-D matrix by adding a time vector.

1. In the MATLAB Command Window, enter:

   ```
   t1 = 0.2 * [0:49]';
   m = magic(10);
   M = repmat(m,[1 1 length(t1)]);
   data.time=t1;
   data.signals.values = M;
   data.signals.dimensions=[10 10];
   ```

   The time vector must be a column vector. The `signals.values field` is a 3-D matrix where the third dimension corresponds to time. The `signals.dimensions` field is a two-element vector. The first element is the number of rows and the second element is the number of columns in the `signals.values` field.
2. In the From Workspace block parameters dialog box, in the **Data** field, enter the name of the structure.

Characteristics

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

Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	No

See Also

"Create Data for a From Workspace Block", "Load Data Using the From Workspace Block", "Use From Workspace Block for Test Case", "Load Signal Data That Uses Units", "Load Signal Data for Simulation", `To Workspace`, `From File`, `To File`

Introduced before R2006a

Function-Call Feedback Latch

Break feedback loop involving data signals between function-call blocks

Library

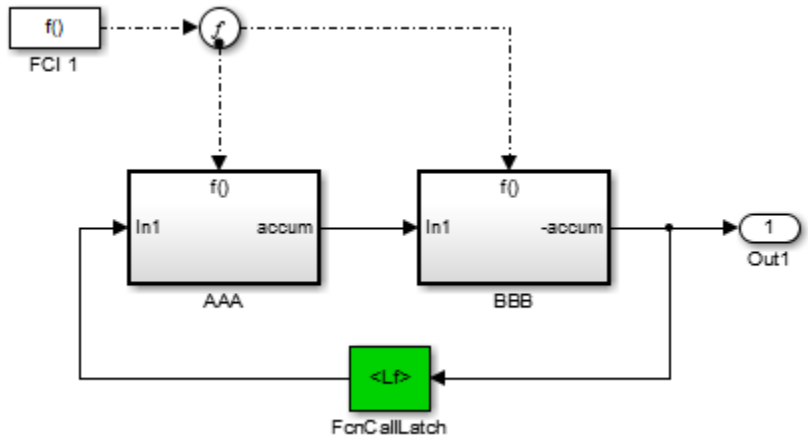
Ports & Subsystems

Description

Use the `Function-Call Feedback Latch` block to break a feedback loop of data signals between one or more function-call blocks. Specifically, break a loop formed in one of the following ways.

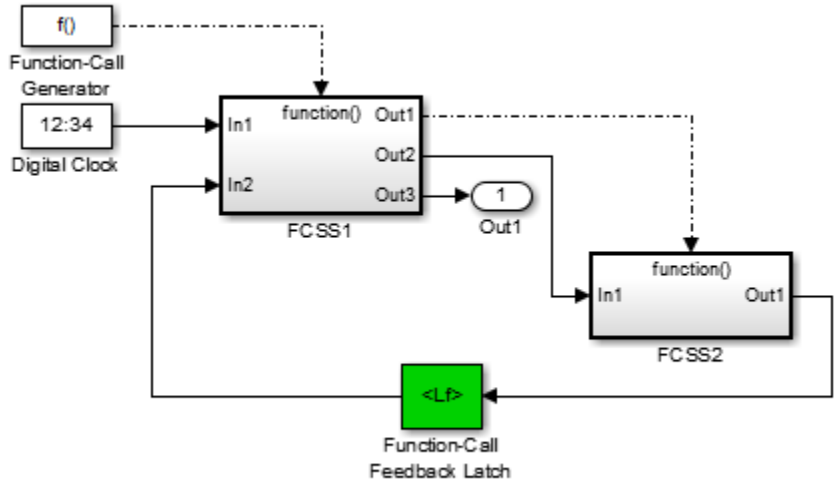
- **When function-call blocks connect to branches of the same function-call signal**

  Place the Function-Call Feedback Latch block on the feedback signal between the branched blocks. As a result, the latch block delays the signal at the input of the destination function-call block, and the destination function-call block executes prior to the source function-call block of the latch block.



- **When the loop involves parent and child function-call blocks, where the child initiator is inside the parent**

  Place the Function-Call Feedback Latch block on the feedback signal between the child and the parent. This arrangement prevents the signal value, read by the parent (FCSS1), from changing during execution of the child. In other words, the parent reads the value from the previous execution of the child (FCSS2).



Using the latch block is equivalent to selecting the **Latch input for function-call feedback signals** check box on the `Inport` block in the destination function-call subsystem. 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.

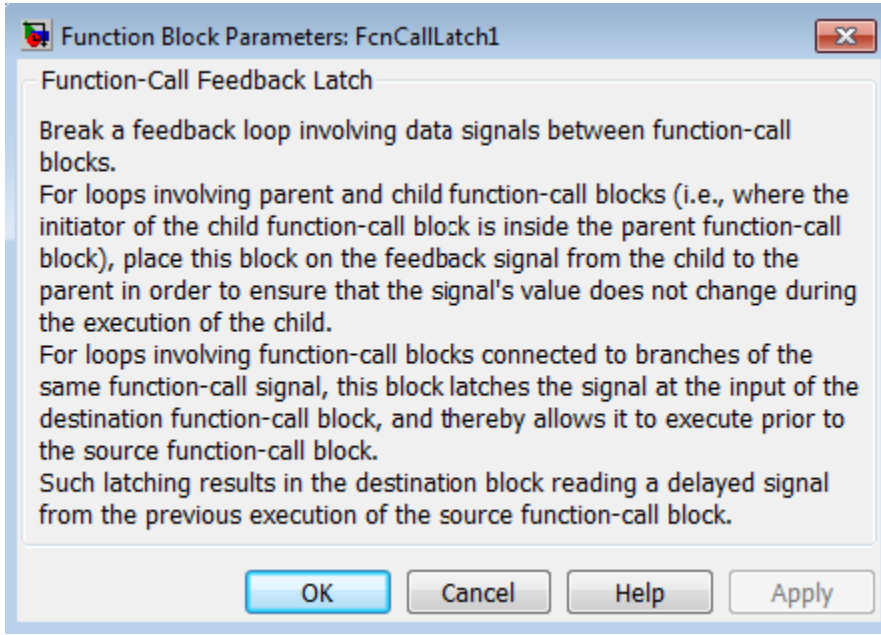
This block does not accept:

- Function-call signals
- Action signals
- Variable-sized signals

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

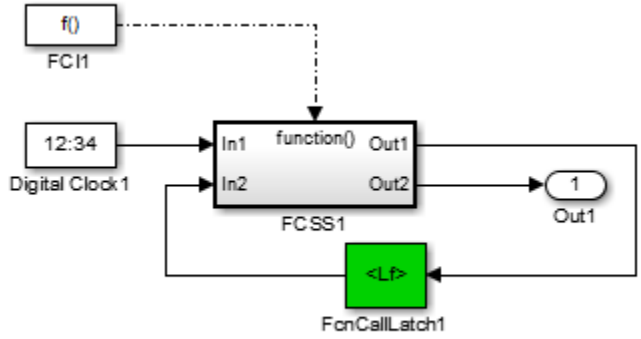
Parameters and Dialog Box

The Function-Call Feedback Latch block dialog box appears as follows:

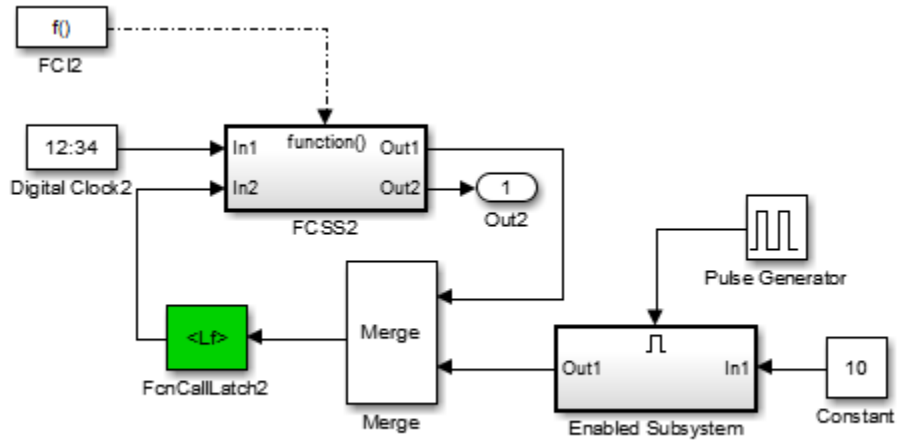


Examples

In the following model, a single function-call subsystem output serves as its own input.



A more complex case occurs when a merged signal serves as the input to a function-call subsystem. Latching is necessary if one of the signals entering the `Merge` block forms a feedback loop with the function-call subsystem. In this example, one of the output signals from FCSS2 combines with the output of an `Enabled Subsystem` block and then feeds back into an inport of FCSS2.



Characteristics

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

See Also

`Function-Call Subsystem` block

Introduced in R2011a

Function-Call Generator

Execute function-call subsystem specified number of times at specified rate

Library

Ports & Subsystems

Description

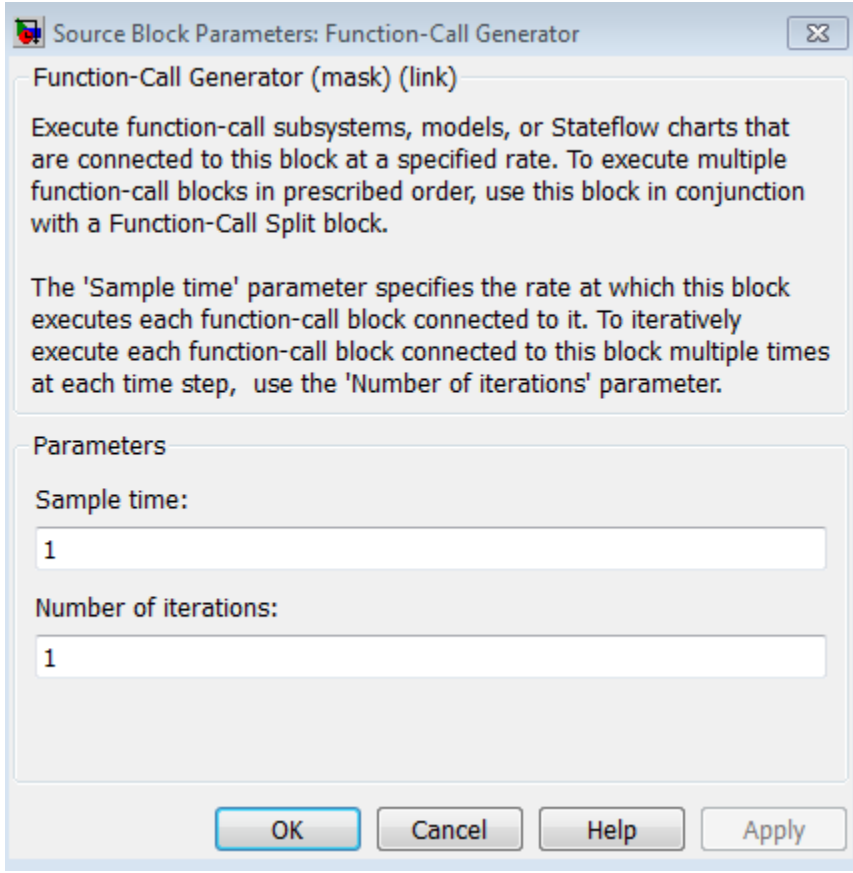
The Function-Call Generator block executes a function-call subsystem (for example, a Stateflow chart acting as a function-call subsystem) at the rate that you specify with the **Sample time** parameter. To iteratively execute each function-call block connected to this block multiple times at each time 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`.

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

Data Types	Not applicable
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Function-Call Split

Provide junction for splitting function-call signal

Library

Ports & Subsystems

Description 

The `Function-Call Split` block allows a function-call signal to branch and connect to several function-call subsystems and models.

A `Function-Call Split` outputs multiple function-call outputs to create multiple branches for a function-call signal. In some cases, when you use this block, you do not need the initiator block to create multiple function-call signals to invoke a set of function-call subsystems or models.

Function-call subsystems or models connected to the output port of the `Function-Call Split` block that are marked with a dot execute before the subsystems or models connected to other output ports. If any data dependencies between these subsystems (or between models) do not support the specified execution order, the block returns an error. To eliminate this error, consider turning on **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 a data-dependency loop. Turning on this option delays the corresponding input signal, thereby eliminating the data-dependency loop.

For a model to contain `Function-Call Split` blocks, you must set the following diagnostic to `error`: **Model Configuration Parameters > Diagnostics > Connectivity > Invalid function-call connection**.

If you 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 appears 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

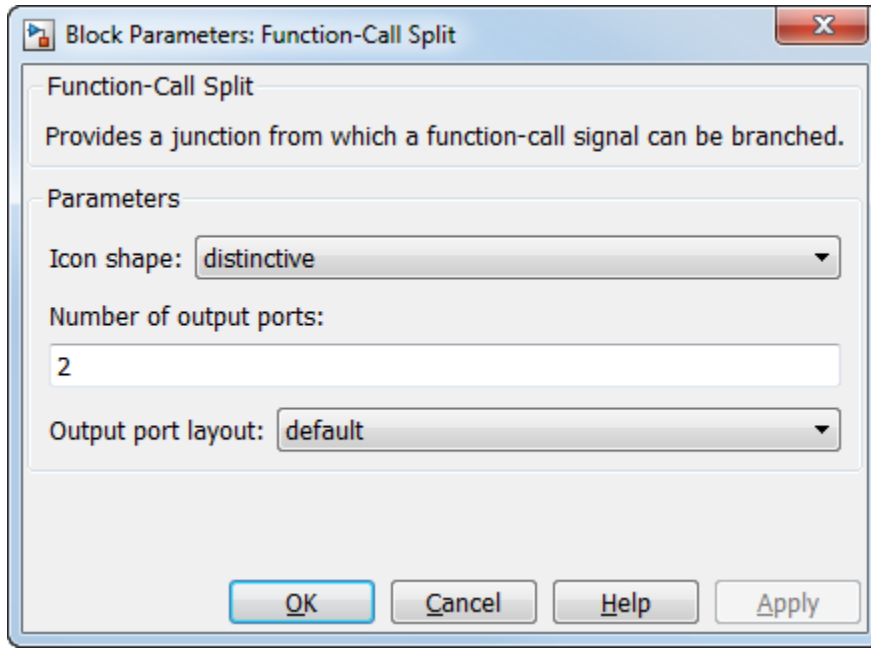
The `Function-Call Split` block has these limitations.

- All function-call subsystems and models connected to a given function-call signal must reside within the same nonvirtual layer of the model hierarchy.
- You cannot connect branched function-call subsystems (or models) and their children directly back to the initiator.
- Function-call subsystems and models connected to branches of a function-call signal cannot have multiple (muxed) initiators.
- A Function-Call Split block cannot have its input from a signal with multiple function-call elements.

Data Type Support

The `Function-Call Split` block accepts periodic and asynchronous function-call signals only.

Parameters and Dialog Box



Icon shape

Choose the shape of the icon. Set the icon shape to `round` for a circular block icon, or to `distinctive`.

Number of output ports

Select the number of output function-call signals.

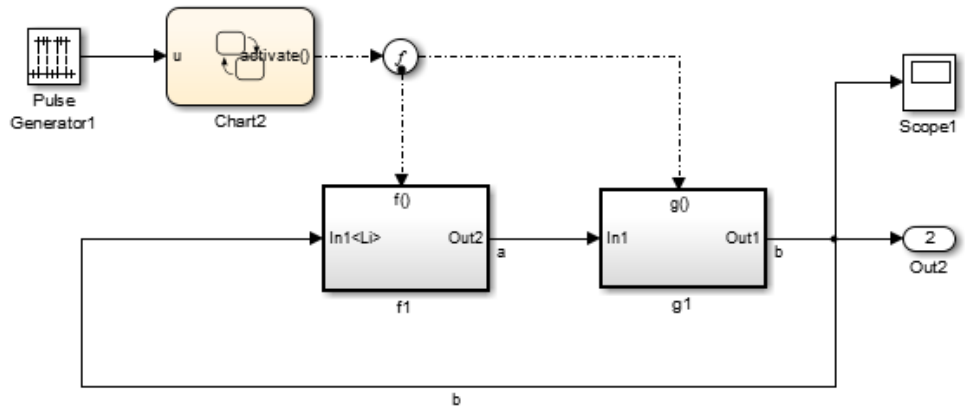
Output port layout

Choose the layout of the output ports.

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. The block breaks the loop of data dependencies between subsystems f1 and g1. The model achieves the behavior of consistently calling f1 to execute before g1. For a given execution step, subsystem f1 uses the g1 output computed at the previous execution step.



Characteristics

Data Types	Double
Sample Time	Inherited from the block driving the function-call split
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

`Function-Call Subsystem` | `Function-Call Generator`

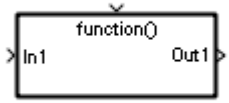
Introduced in R2010a

Function-Call Subsystem

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 "Create a Function-Call Subsystem" in the "Creating a Model" chapter of the Simulink documentation.

Characteristics

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

Introduced before R2006a

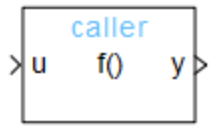
Function Caller

Call Simulink or Stateflow function

Library

User-Defined Functions

Description



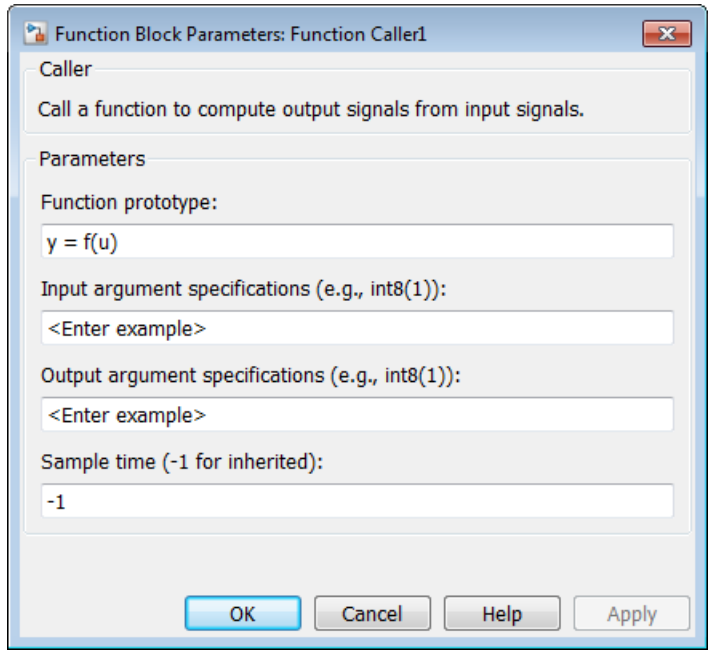
Function Caller

A Function Caller block calls and executes a function defined in a Simulink Function block or by calling an exported Stateflow function.

Data Type Support

A Function Caller block accepts a real or complex scalar, vector, or matrix of any numeric data type that Simulink supports. It also supports fixed-point, bus, and enumerated data types. For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box



Function prototype

Specify the function name, input arguments, and output arguments for the function implementation. The arguments must also match the argument definition for the function the block calls.

The function prototype uses MATLAB syntax to identify input and output arguments and the name of the called function. Each of these must be a valid MATLAB identifier.

The function prototype determines the number and name of input ports and output ports that appear on the Function Caller block. Connect signal lines to the ports to pass data to and from a function through the function arguments. For example, `y = myfunction(u)` creates one input port (`u`) and one output port (`y`) on the Function Caller block.

Input argument specifications

Specify a comma-separated list of MATLAB expressions that combine data type, dimensions, and complexity (real or imaginary) of each input argument. This specification must match the Simulink Function block data types specified with the **Data type** parameter in the Source Block Parameters blocks.

Output argument specifications

Specify a comma-separated list of MATLAB expressions that combine data type, dimensions, and complexity (real or imaginary) of each output argument. This specification must match the Simulink Function block data types specified with the **Data type** parameter in the Sink Block Parameters blocks.

When you can optionally specify arguments — When a Simulink Function block is within the scope of a Function Caller block, the Function Caller block can determine the input and output argument specifications without you specifying the parameters.

When you must specify arguments — When a Simulink Function block is outside the scope of a Function Caller block, you must specify the **Input argument specifications** and **Output argument specifications** parameters. This can happen when a Function Caller block and a Simulink Function block are in separate models that are referenced by a common parent model.

The following table includes a list of possible input and output argument specifications.

Simulink Function Block Data Type	Function Caller Block Expression	Description
`double`	`double(1.0)`	Double-precision scalar.
`double`	`double(ones(12,1))`	Double-precision column vector of length 12.
`single`	`single(1.0)`	Single-precision scalar.
`int8`, `int16`, `int32`	`int8(1)`, `int16(1)`, `int32(1)`	Integer scalars.
	`int32([1 1 1])`	Integer row vector of length 3.
	`int32(1+1i)`	Complex scalar whose real and imaginary parts are 32-bit integers.
`uint8`, `int16`, `int32`	`uint8(1), uint16(1), uint32(1)`	Unsigned integer scalers.

Simulink Function Block Data Type	Function Caller Block Expression	Description
`boolean`	`boolean(true)boolean(`	Booleans, initialized to true (1) or false (0).

`fixdt(1,16)`
 	
`fixdt (signed, word_length)`
	`fi(0,1,16)`
	
	`fi (value, signed, word_length)`
		16-bit fixed-point signed scalar with binary point set to zero.
		
		Fixed-point numbers can have a word size up to 128 bits.

`fixdt(1,16,4)`	`fi(0,1,16,4)`	16-bit fixed-point signed scalar with binary point set to 4.
`fixdt(1,16,2^0,0)`	`fi(0,1,16,2^0,0)`	16-bit fixed-point signed scalar with slope set to 2^0 and bias set to 0.
`Bus: <object name>`	`parameter object name`	`Simulink.Parameter` object with the `Value` parameter set to a MATLAB structure for the bus.
`Enum: <class name>`	`parameter object name`	`Simulink.Parameter` object with the `Value` parameter set to an enumerated value.
`<alias name>`	`parameter object name`	`Simulink.Parameter` object with the `DataType` parameter set to a `Simulink.AliasType` object and the `Value` parameter set to a value.

Sample time (-1 for inherited)

Specify the time interval between function calls. To inherit sample time, set this parameter to `-1`. If the Function Caller block has any inputs, it is a nonsource block. In this case, you must set **Sample time** to `-1`. See “Specify Sample Time” for more information.

Examples

Specify Input Specification for a Bus Data Type

Create a bus with two signals, and then specify the **Input argument specification** parameter for a Function Caller block. The Function Caller block calls a Simulink Function block that accepts the bus as input.

1. Create a Simulink bus object `myBus`.

   ```
   myBus = Simulink.Bus;
   ```

2. Add elements `A` and `B`.

   ```
   myBus.Elements(1).Name = 'A';
   myBus.Elements(2).Name = 'B';
   ```

3. Create a MATLAB structure `myBus_MATLABstruct` with fields `A` and `B`.

   ```
   myBus_MATLABStruct.A = 0;
   myBus_MATALBStruct.B = 0;
   ```

4. Create a Simulink parameter object `myBus_parameter` and assign the MATLAB structure to the `Value` parameter.

   ```
   myBus_parameter = Simulink.Parameter;
   myBus_parameter.Value = myBus_MATLABStruct;
   ```

5. In the Function Caller block dialog box, set the **Input argument specification** parameter to `myBus_parameter`.

6. In the Argument In block dialog box within a Simulink Function, set the **Data type** parameter to `Bus: myBus`.

Specify Input Specification for an Enumerated Data Type

Create an enumerated data type for the three primary colors, and then specify the **Input argument specification** parameter for a Function Caller block. The Function Caller block calls a Simulink Function block that accepts a signal with the enumerated type as input.

1. Create a MATLAB file for saving the data type definition. On the MATLAB toolstrip, select **New** > **Class**.

2. In the MATLAB editor, define the elements of an enumerated data type. The class `BasicColors` is a subclass of the class `Simulink.IntEnumType`.

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
end
```

3. Save the class definition in a file named `BasicColors.m`.
4. Create a Simulink parameter object `myEnum_parameter` and assign one of the enumerated values to the `Value` parameter.

```
myEnum_parameter = Simulink.Parameter;
myEnum_parameter.Value = BasicColors.Red;
```

5. For the Function Caller block dialog box, set the **Input argument specification**. to `myEnum_parameter`.
6. For the Argument In block dialog box within a Simulink Function block, set the **Data type** parameter to `Enum: BasicColors`.

Specify Input Specification for an Alias Data Type

Create an alias name for the data type single, and then specify the **Input argument specification** parameter for a Function Caller block. The Simulink Function block called by the Function Caller block also uses the alias name to define the input data type.

1. Create a Simulink alias data type object `myAlias`.

```
myAlias = Simulink.AliasType;
```

2. Assign a data type.

```
myAlias.BaseType = 'single';
```

3. Create a Simulink parameter object `myAlias_parameter` and assign the alias name to the `DataType` parameter.

```
myAlias_parameter = Simulink.Parameter;
myAlias_parameter.DataType = myAlias;
myAlias_parameter.Value = 1;
```

4. In the `Function Caller` block dialog box, set the **Input argument specification** parameter to `myAlias_parameter`.

5 In the Argument In block dialog box within a Simulink Function block, set the **Data type** parameter to `myAlias`.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Function-Call Subsystem` | `Simulink Function`

Related Examples

- "Simulink Functions in Simulink Models"

Introduced in R2014b

Gain

Multiply input by constant

Library

Math Operations

Description

The Gain block multiplies the input by a constant value (gain). The input and the gain can each be a scalar, vector, or matrix.

You specify the value of the gain in the **Gain** parameter. The **Multiplication** parameter lets you specify element-wise or matrix multiplication. For matrix multiplication, this parameter also lets you indicate the order of the multiplicands.

The gain is converted from doubles to the data specified in the block mask offline using round-to-nearest and saturation. The input and gain are then multiplied, and the result is converted to the output data type using the specified rounding and overflow modes.

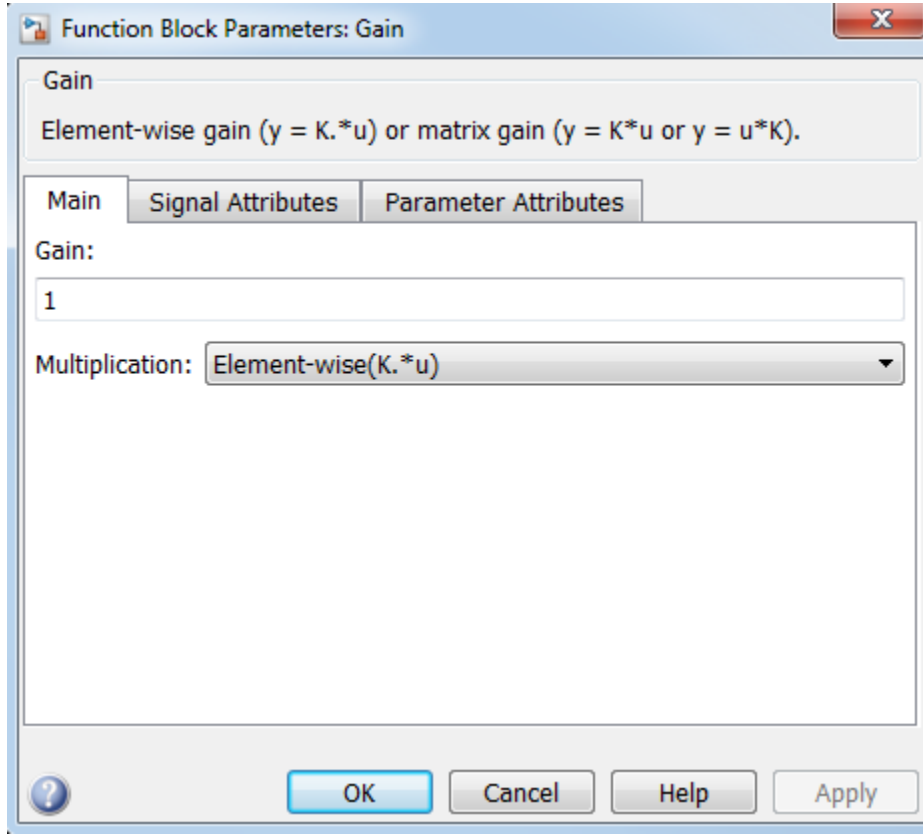
Data Type Support

The Gain block accepts a real or complex scalar, vector, or matrix of any numeric data type that Simulink supports. The Gain block supports fixed-point data types. If the input of the Gain block is real and the gain is complex, the output is complex.

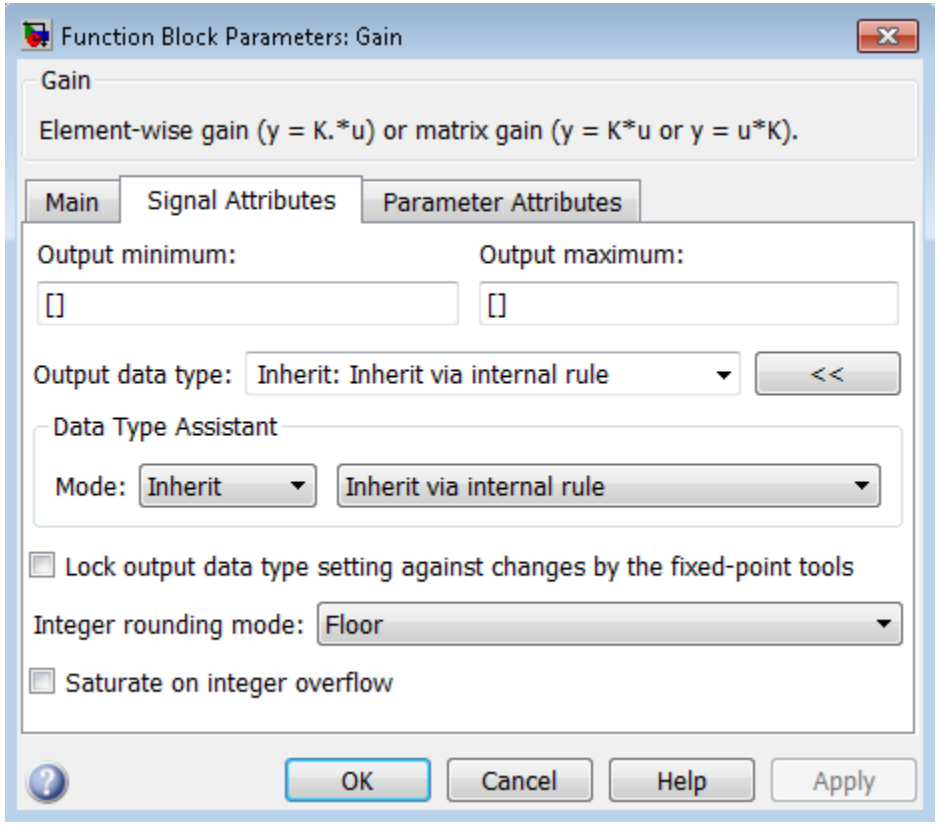
For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

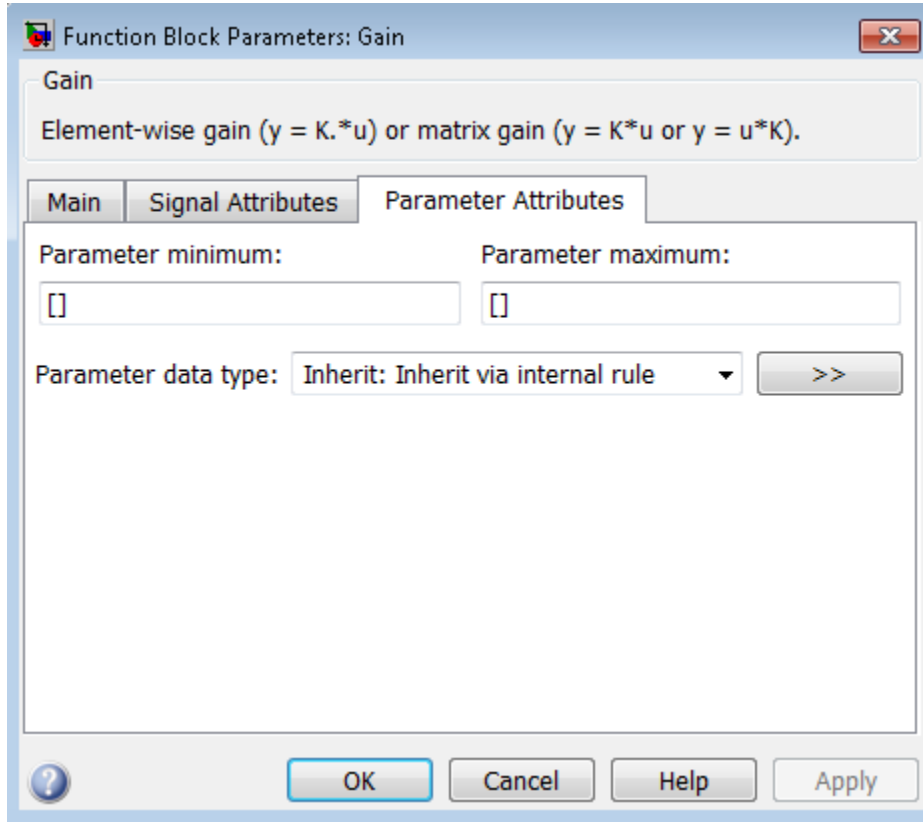
The **Main** pane of the Gain block dialog box appears as follows:



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



The **Parameter Attributes** pane of the Gain block dialog box appears as follows:


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
Parameter minimum:
Parameter maximum:
[]
[]
Parameter data type:
Inherit: Inherit via internal rule
>>
OK
Cancel
Help
Apply

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 6-101 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 6-101 for the command-line information.

Sample time

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

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the minimum to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges") 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a `Data Type Propagation` block. Examples of how to use this block are available in the Signal Attributes library `Data Type Propagation Examples` block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as input`

Use data type of input signal.

`double`

Output data type is `double`.

`single`

Output data type is `single`.

`int8`

Output data type is `int8`.

`uint8`

Output data type is `uint8`.

`int16`

Output data type is `int16`.

`uint16`

Output data type is `uint16`.

`int32`

Output data type is `int32`.

`uint32`

Output data type is `uint32`.

`fixdt(1,16,0)`

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

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

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

`<data type expression>`

Use a data type object, for example, `Simulink.NumericType`.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

See Also

See "Control Signal 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 6-101 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.

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.

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 6-101 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.

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.

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.

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 “Control Block Parameter Value Ranges”)
- Automatic scaling of fixed-point data types

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 "Control Block Parameter Value Ranges")
- Automatic scaling of fixed-point data types

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 6-101 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 6-101 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**

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

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

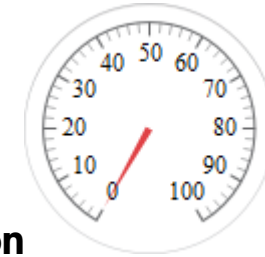
Introduced before R2006a

Gauge

Display input value on circular scale

Library

Dashboard



Description

The `Gauge` block displays connected signals during simulation on a circular gauge.

To view data from a signal on the `Gauge` block, double-click the `Gauge` block to open the dialog box. Select a signal in the model. The signal appears in the dialog box **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal to the block.

You can modify the tick range by modifying the **Minimum**, **Maximum**, and **Tick Interval** values.

You can also add scale colors that appear on the outside of the `Gauge` block scale using the **Scale Colors** table.

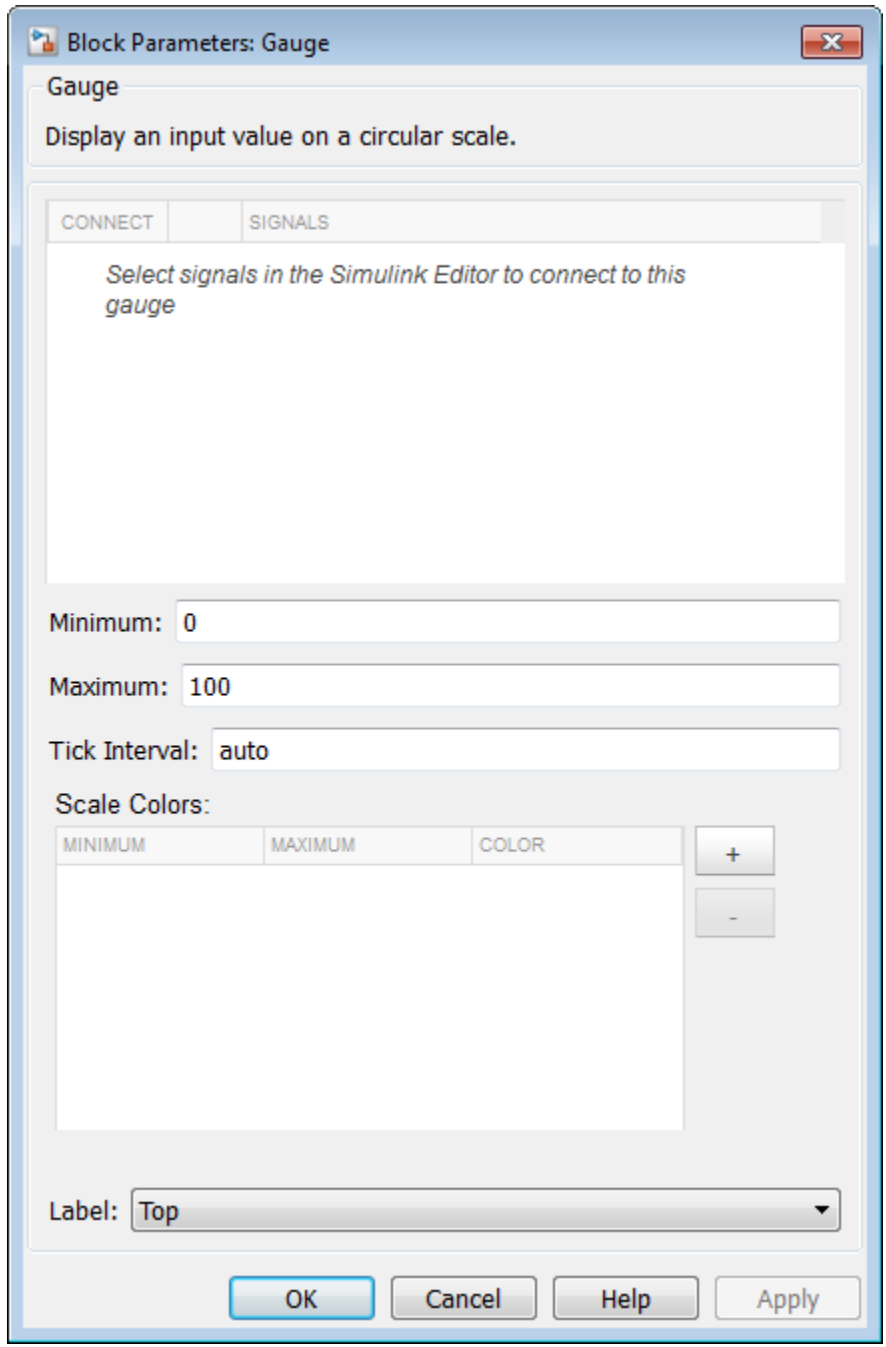
Limitations

The `Gauge` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.

If you turn off streaming for a signal connected to any dashboard gauge, then the connection shows as broken. Signal data does not stream to the block. To view signal data again, double-click the gauge and reconnect the signal.

The External simulation mode is not supported for the `Gauge` block.


Block Parameters: Gauge
Gauge
Display an input value on a circular scale.
CONNECT
SIGNALS
Select signals in the Simulink Editor to connect to this gauge
Minimum:
0
Maximum:
100
Tick Interval:
auto
Scale Colors:
MINIMUM
MAXIMUM
COLOR
+
-
Label:
Top
OK
Cancel
Help
Apply

Connection

Select a signal to connect and display.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

Settings

The table has a row for the signal connected to the block. If there are no signals selected in the model or the block is not connected to any signals, then the table is empty.

Minimum

Minimum tick mark value.

Settings

Default: 0

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

Dependencies

The **Minimum** tick value must be less than the **Maximum** tick value.

Maximum

Maximum tick mark value.

Settings

Default: 100

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

Dependencies

The **Maximum** tick value must be greater than the **Minimum** tick value.

Tick Interval

Interval between major tick marks.

Settings

Default: `auto`

Specify this number as a finite, real, positive, integer, scalar value. Specify as `auto` for the block to adjust the tick interval automatically.

Scale Colors

Specify ranges of color bands on the outside of the scale. Specify the minimum and maximum color range to display on the gauge.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

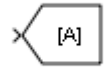
Introduced in R2015a

Goto

Pass block input to From blocks

Library

Signal Routing



Description

The Goto block passes its input to its corresponding From blocks. The input can be a real- or complex-valued signal or vector of any data type. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them.

A Goto block can pass its input signal to more than one From block, although a From block can receive a signal from only one Goto block. The input to that Goto block is passed to the From blocks associated with it as though the blocks were physically connected. Goto blocks and From blocks are matched by the use of Goto tags.

The **Tag Visibility** parameter determines whether the location of From blocks that access the signal is limited:

- `local`, the default, means that From and Goto blocks using the same tag must be in the same subsystem. A local tag name is enclosed in brackets ([]).
- `scoped` means that From and Goto blocks using the same tag must be in the same subsystem or at any level in the model hierarchy below the Goto Tag Visibility block that does not entail crossing a nonvirtual subsystem boundary, i.e., the boundary of an atomic, conditionally executed, or function-call subsystem or a model reference. A scoped tag name is enclosed in braces ({}).
- `global` means that From and Goto blocks using the same tag can be anywhere in the model except in locations that span nonvirtual subsystem boundaries.

The rule that From-Goto block connections cannot cross nonvirtual subsystem boundaries has the following exception. A Goto block connected to a state port in one

conditionally executed subsystem is visible to a From block inside another conditionally executed subsystem. 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.

Parameters and Dialog Box

Goto Tag

The Goto block identifier. This parameter identifies the Goto block whose scope is defined in this block.

Rename All

Rename the `Goto` tag. The new name propagates to the `From` and `Goto Tag Visibility` blocks that are listed in the **Corresponding blocks** box.

Tag Visibility

The scope of the Goto block tag: `local`, `scoped`, or `global`. The default is `local`.

Corresponding blocks

List of the `From` blocks and `Goto Tag Visibility` blocks connected to this `Goto` block. Click an entry in the list to display and highlight the corresponding `From` or `Goto Tag Visibility` block.

Icon Display

Specifies the text to display on the block's icon. The options are the block's tag, the name of the signal that the block represents, or both the tag and the signal name.

Examples

The following models show how to use the `Goto` block:

- `sldemo_auto_climatecontrol`
- `sldemo_hardstop`

Characteristics

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

See Also

`From` `Goto Tag Visibility`

Introduced before R2006a

Goto Tag Visibility

Define scope of Goto block tag

Library

Signal Routing

Description

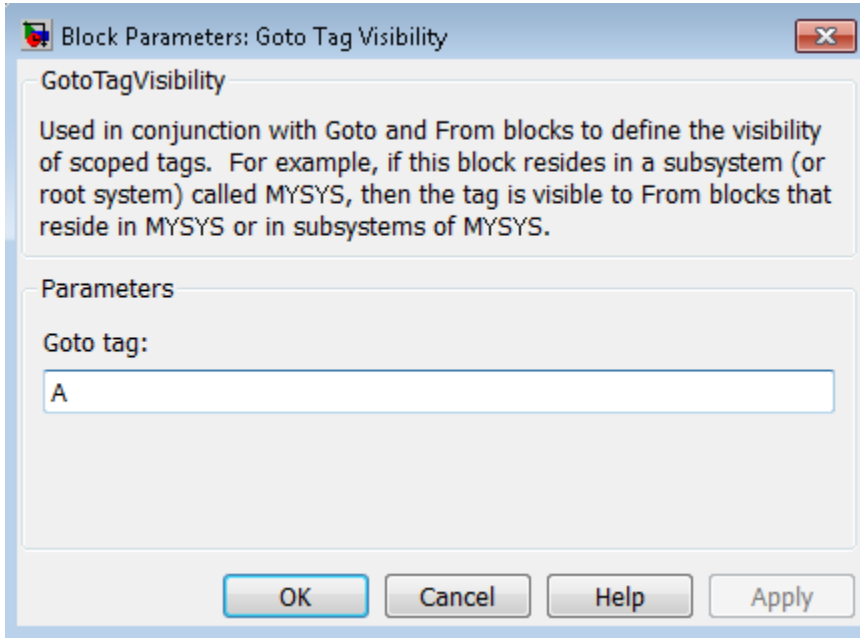
The Goto Tag Visibility block defines the accessibility of Goto block tags that have `scoped` visibility. The tag specified as the **Goto tag** parameter is accessible by From blocks in the same subsystem that contains the Goto Tag Visibility block and in subsystems below it in the model hierarchy.

A Goto Tag Visibility block is required for Goto blocks whose **Tag Visibility** parameter value is `scoped`. No Goto Tag Visibility block is needed if the tag visibility is either `local` or `global`. The block shows the tag name enclosed in braces ({}).

Data Type Support

Not applicable.

Parameters and Dialog Box



Goto tag

The Goto block tag whose visibility is defined by the location of this block.

If you use multiple `From` and `Goto Tag Visibility` blocks to refer to the same Goto tag, you can simultaneously rename the tag in all of the blocks. Use the **Rename All** button in the `Goto` block dialog box.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No

Code Generation	Yes

Introduced before R2006a

Ground

Ground unconnected input port

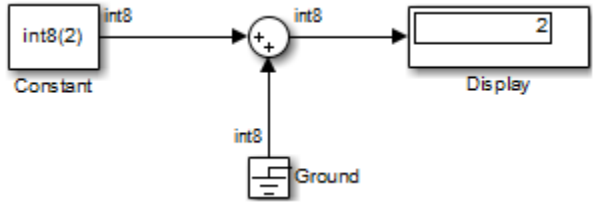
Library

Sources

Description

The Ground block connects to blocks whose input ports do not connect to other blocks. If you run a simulation with blocks having unconnected input ports, Simulink issues warnings. Using a Ground block to ground those unconnected blocks can prevent these warnings.

The Ground block outputs a signal of the same data type as the port to which it connects. For example, consider the following model:



In this example, the output of the Constant block determines the data type (`int8`) of the port to which the Ground block is connected. That port determines the output data type of the Ground block.

The Ground block outputs a signal with zero value. When the output data type cannot represent zero exactly, the Ground block outputs a nonzero value that is the closest possible value to zero. This behavior applies only to fixed-point data types with nonzero bias. The following expressions are examples of fixed-point data types that cannot represent zero:

- `fixdt(0, 8, 1, 1)` — an unsigned 8-bit type with slope of 1 and bias of 1
- `fixdt(1, 8, 6, 3)` — a signed 8-bit type with slope of 6 and bias of 3

If the output is an enumerated data type, the Ground block outputs the default value of the enumeration. This behavior applies whether or not:

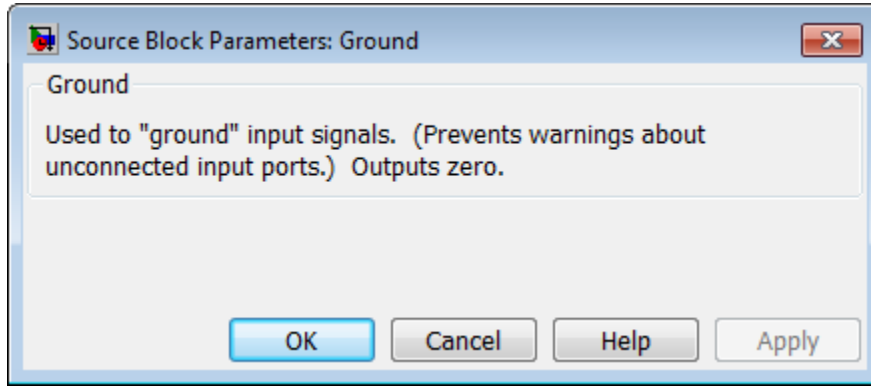
- The enumeration can represent zero.
- The default value of the enumeration is zero.

If the enumerated type does not have a default value, the Ground block outputs the first enumeration value in the type definition.

Data Type Support

The Ground block supports all data types that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Examples

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

- `sldemo_doublebounce`

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Constant (`inf`)
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

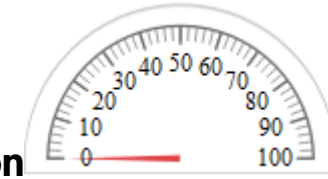
Introduced before R2006a

Half Gauge

Display input value on semicircular scale

Library

Dashboard



Description

The `Half Gauge` block displays connected signals during simulation on a semicircular gauge.

To view data from a signal on the `Half Gauge` block, double-click the `Half Gauge` block to open the dialog box. Select a signal in the model canvas. The signal appears in the dialog box **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal to the block.

You can modify the tick range by modifying the **Minimum**, **Maximum**, and **Tick Interval** values.

You can also add scale colors that appear on the outside of the `Half Gauge` block scale using the **Scale Colors** table.

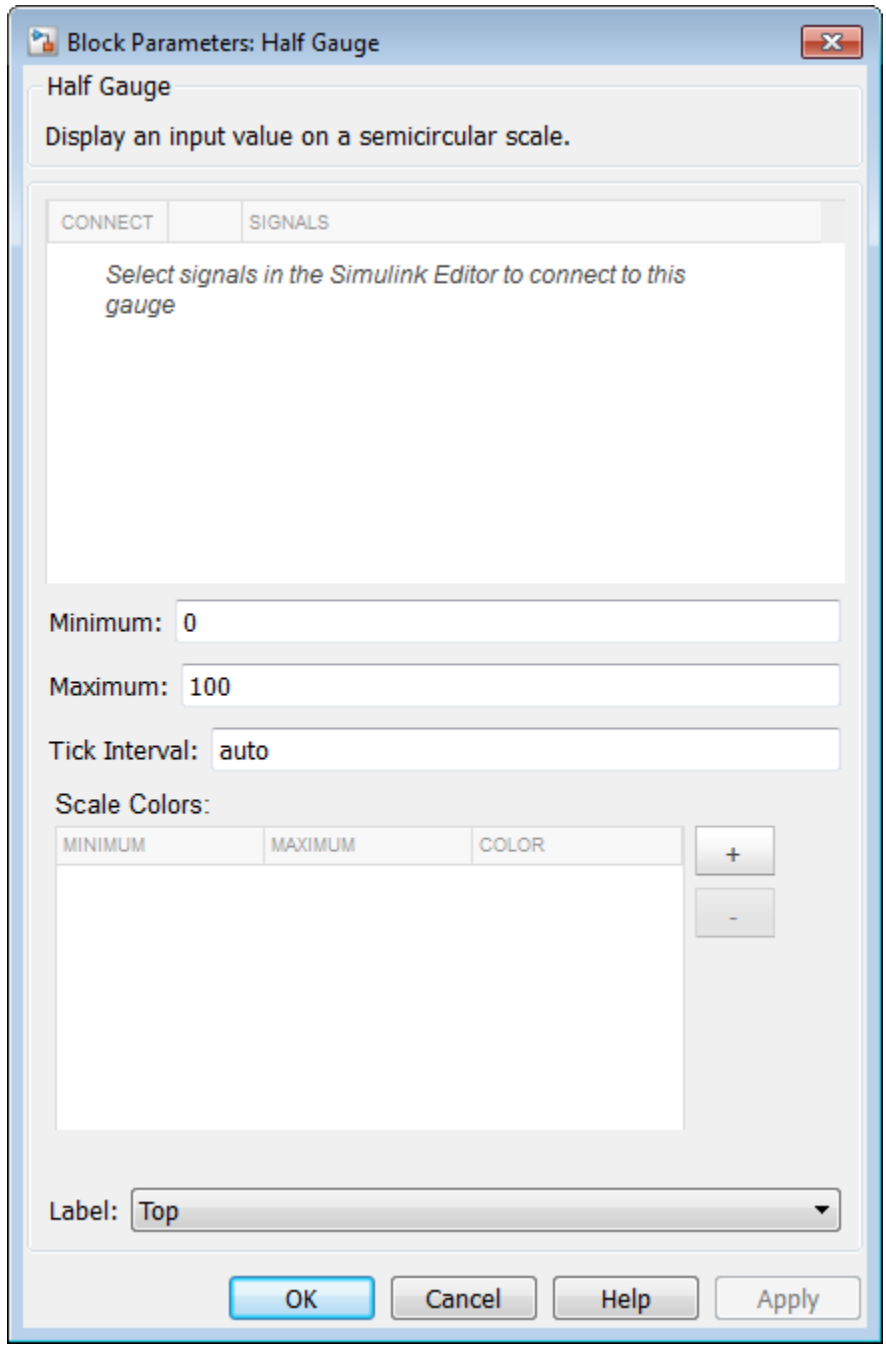
Limitations

The `Half Gauge` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.

If you turn off streaming for a signal connected to any dashboard gauge, then the connection shows as broken. Signal data does not stream to the block. To view signal data again, double-click the gauge and reconnect the signal.

The External simulation mode is not supported for the `Half Gauge` block.


Block Parameters: Half Gauge
Half Gauge
Display an input value on a semicircular scale.
CONNECT
SIGNALS
Select signals in the Simulink Editor to connect to this gauge
Minimum:
0
Maximum:
100
Tick Interval:
auto
Scale Colors:
MINIMUM
MAXIMUM
COLOR
+
-
Label:
Top
OK
Cancel
Help
Apply

Connection

Select a signal to connect and display.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

Settings

The table has a row for the signal connected to the block. If there are no signals selected in the model or the block is not connected to any signals, then the table is empty.

Minimum

Minimum tick mark value.

Settings

Default: 0

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

Dependencies

The **Minimum** tick value must be less than the **Maximum** tick value.

Maximum

Maximum tick mark value.

Settings

Default: 100

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

Dependencies

The **Maximum** tick value must be greater than the **Minimum** tick value.

Tick Interval

Interval between major tick marks.

Settings

Default: `auto`

Specify this number as a finite, real, positive, integer, scalar value. Specify as `auto` for the block to adjust the tick interval automatically.

Scale Colors

Specify ranges of color bands on the outside of the scale. Specify the minimum and maximum color range to display on the gauge.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see "Tune and Visualize Your Model with Dashboard Blocks".

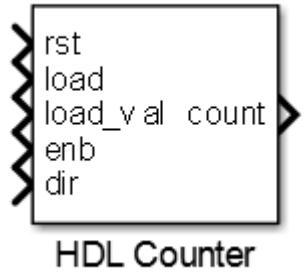
Introduced in R2015a

HDL Counter

Free-running or count-limited hardware counter

Library

HDL Coder / HDL Operations



Description

The `HDL Counter` block models a free-running or count-limited hardware counter that supports signed and unsigned integer and fixed-point data types.

The counter emits its value for the current sample time.

Control Ports

By default, the counter does not have input ports. Optionally, you can add control ports that enable, disable, load, reset or set the direction of the counter.

The table shows the priority of the control signals and how the counter value is updated in relation to the control signals.

Local reset, `rst`	Load trigger, `load`	Count enable, `enb`	Count direction, `dir`	Next Counter Value
1	–	–	–	initial value

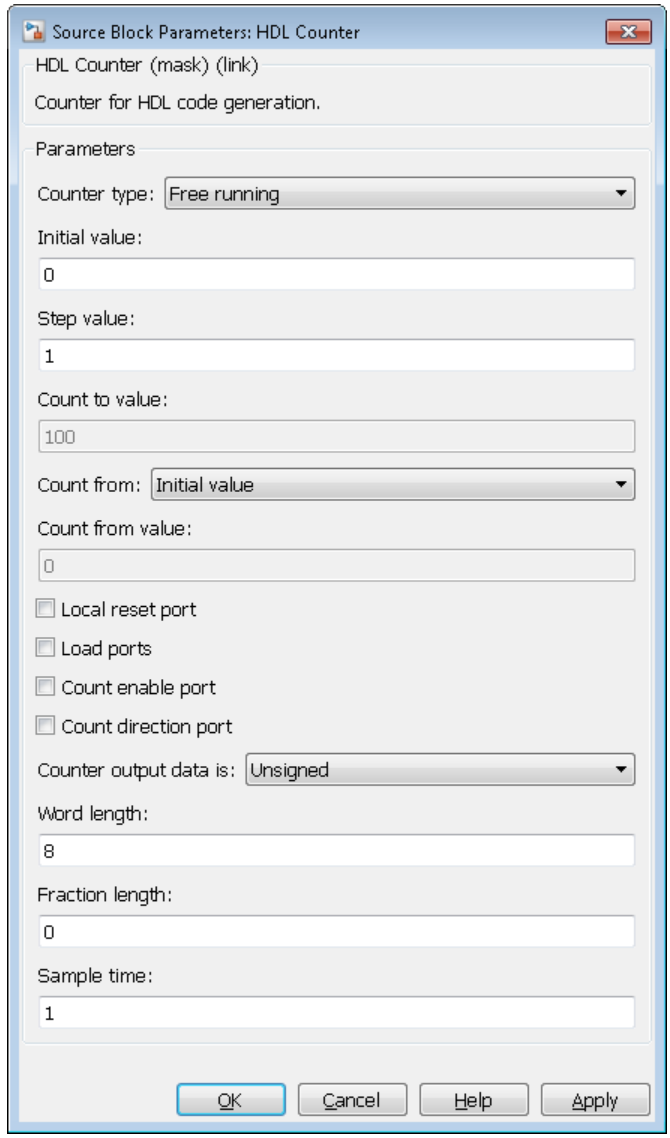
Local reset, `rst`	Load trigger, `load`	Count enable, `enb`	Count direction, `dir`	Next Counter Value
0	1	–	–	`load_val` value
0	0	0	–	current value
0	0	1	1	current value + step value
0	0	1	0	current value - step value

Count direction

The **Step value** parameter and optional count direction port, `dir`, interact to determine the actual count direction.

`dir` Signal Value	Step Value Sign	Actual Count Direction
1	+ (positive)	Up
1	- (negative)	Down
0	+ (positive)	Down
0	- (negative)	Up

Dialog Box and Parameters



Counter type

Counter behavior.

- `Free running` (default): The counter continues to increment or decrement by the **Step value** until reset.
- `Count limited`: The counter increments or decrements by the **Step value** until it is exactly equal to the **Count to value**.

Initial value

Counter value after reset. The default is 0.

Step value

Value added to counter at each sample time. The default is 1.

Count to value

When the count is exactly equal to **Count to value**, the count restarts at the **Initial value**. This option is available when **Counter type** is set to `Count limited`. The default is 100.

Count from

Specifies the parameter that sets the start value after rollover. When set to `Specify`, the **Count from value** parameter is the start value after rollover. The default is `Initial value`.

Count from value

Counter value after rollover when **Count from** is set to `Specify`. The default is 0.

Local reset port

When selected, creates a local reset port, `rst`.

Load ports

When selected, creates a load data port, `load_val`, and load trigger port, `load`.

Count enable port

When selected, creates a count enable port, `enb`.

Count direction port

When selected, creates a count direction port, `dir`.

Counter output data is

Output data type signedness. The default is `Unsigned`.

Word length

Bit width, including sign bit, for an integer counter; word length for a fixed-point data type counter. The minimum value if Output data type is `Unsigned` is 1, 2 if `Signed`. The maximum value is 125. The default is 8.

Fraction length

Fixed-point data type fraction length. The default is 0.

Sample time

Sample time. The default is 1.

This parameter is not available, and the block inherits its sample time from the input ports when any of these parameters is selected:

- **Local reset port**
- **Load ports**
- **Count enable port**
- **Count direction port**

Ports

The block has the following ports:

`rst`

Resets the counter value. Active-high.

This port is available when you select **Local reset port**.

Data type: Boolean

`load`

Sets the counter to the load value, `load_val`. Active-high.

This port is available when you select **Load ports**.

Data type: Boolean

`load_val`

Data value to load.

This port is available when you select **Load ports**.

Data type: Same as `count`.

`enb`

Enables counter operation. Active-high.

This port is available when you select **Count enable port**.

Data type: Boolean

`dir`

Count direction. This port interacts with **Step value** to determine count direction.

- `1`: **Step value** is added to the current counter value to compute the next value.
- `0`: **Step value** is subtracted from the current counter value to compute the next value.

This port is available when you select **Count direction port**.

Data type: Boolean

`count`

Counter value.

Data type: Determined automatically based on **Counter output data is**, **Word length**, and **Fraction length**.

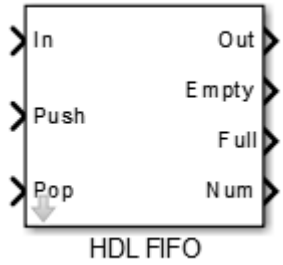
Introduced in R2014a

HDL FIFO

Stores sequence of input samples in first in, first out (FIFO) register

Library

HDL Coder / HDL Operations



Description

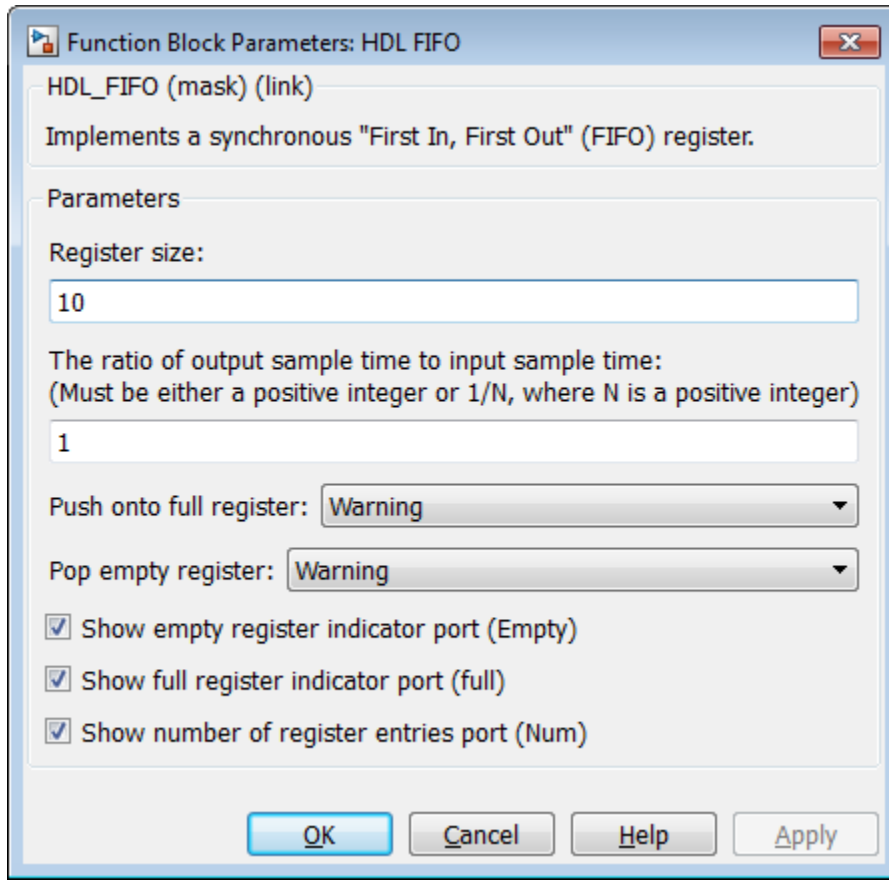
The `HDL FIFO` block stores a sequence of input samples in a first in, first out (FIFO) register.

HDL Code Generation

For simulation results that match the generated HDL code, in the Configuration Parameters dialog box, in the Solver pane, **Tasking mode for periodic sample times** must be `SingleTasking`.

If you simulate this block using `MultiTasking` mode, the output data can update in the same cycle, but in the generated HDL code, the output data is updated one cycle later.

Dialog Box and Parameters



Register size

Specify the number of entries that the FIFO register can hold. The minimum is 4. The default is 10.

The ratio of output sample time to input sample time

Inputs (`In`, `Push`) and outputs (`Out`, `Pop`) can run at different sample times. Enter the ratio of output sample time to input sample time. Use a positive integer or `1/N`, where `N` is a positive integer. The default is 1.

For example:

- If you enter `2`, the output sample time is twice the input sample time, meaning the outputs run slower.
- If you enter `1/2`, the output sample time is half the input sample time, meaning the outputs run faster.

The `Full`, `Empty`, and `Num` signals run at the faster rate.

Push onto full register

Response (`Ignore`, `Error`, or `Warning`) to a trigger received at the Push port when the register is full. The default is `Warning`.

Pop empty register

Response (`Ignore`, `Error`, or `Warning`) to a trigger received at the Pop port when the register is empty. The default is `Warning`.

Show empty register indicator port (Empty)

Enable the Empty output port, which is high (1) when the FIFO register is empty and low (0) otherwise.

Show full register indicator port (Full)

Enable the Full output port, which is high (1) when the FIFO register is full and low (0) otherwise.

Show number of register entries port (Num)

Enable the Num output port, which tracks the number of entries currently in the queue.

Ports

The block has the following ports:

`In`

Data input signal.

`Push`

Control signal. When this port receives a value of `1`, the block pushes the input at the `In` port onto the end of the FIFO register.

`Pop`

Control signal. When this port receives a value of `1`, the block pops the first element off the FIFO register and holds the `Out` port at that value.

`Out`

Data output signal.

`Empty`

The block asserts this signal when the FIFO register is empty. This port is optional.

`Full`

The block asserts this signal when the FIFO register is full. This port is optional.

`Num`

Current number of data values in the FIFO register. This port is optional.

If two or more of the control input ports are triggered in the same time step, the operations execute in the following order:

1. Pop
2. Push

See Also

`Dual Rate Dual Port RAM`

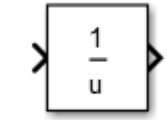
Introduced in R2014a

HDL Reciprocal

Calculate reciprocal with Newton-Raphson approximation method

Library

HDL Coder / HDL Operations



Description

The `HDL Reciprocal` block uses the Newton-Raphson iterative method to compute the reciprocal of the block input. The Newton-Raphson method uses linear approximation to successively find better approximations to the roots of a real-valued function.

The reciprocal of a real number a is defined as a zero of the function:

$$f(x) = \frac{1}{x} - a$$

HDL Coder™ chooses an initial estimate in the range $0 < x_0 < \frac{2}{a}$ as this is the domain of convergence for the function.

To successively compute the roots of the function, specify the **Number of iterations** parameter in the Block Parameters dialog box. The process is repeated as:

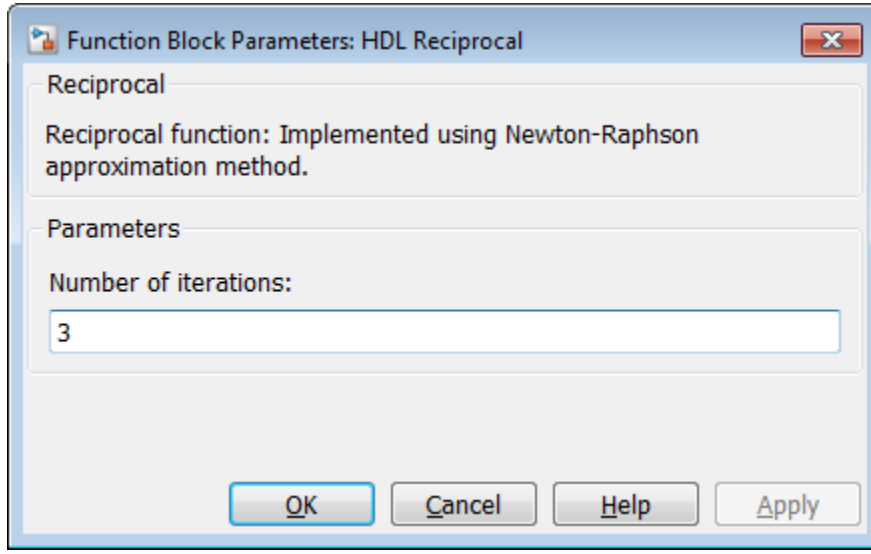
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + (x_i - a{x_i}^2) = x_i.(2 - a x_i)$$

$f'(x)$ is the derivative of the function $f(x)$.

Following table shows comparison of simulation behavior of `HDL Reciprocal` with `Math Reciprocal` block:

`Math Reciprocal`	`HDL Reciprocal`
Computes the reciprocal as 1/N by using the HDL divide operator (/) to implement the division.	Uses the Newton-Rapshon iterative method. The block computes an approximate value of reciprocal of the block input and can yield different simulation results compared to the Math Reciprocal block. To match the simulation results with the `Math Reciprocal` block, increase the number of iterations for the `HDL Reciprocal` block.

Dialog Box and Parameters



Number of iterations

Number of Newton-Raphson iterations. The default is 3.

Ports

The block has the following ports:

Input

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

Output

Input data type	Output data type
double	double
single	single
built-in integer	built-in integer
built-in fixed-point	built-in fixed-point
`fi` (*value*, 0, *word_length*, *fraction_length*)	`fi` (*value*, 0, *word_length*, *word_length–fraction_length*–1)
`fi` (*value*, 1, *word_length*, *fraction_length*)	`fi` (*value*, 1, *word_length*, *word_length–fraction_length*–2)

See Also

`Divide` | `Math Function`

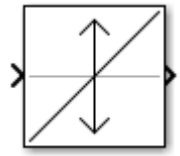
Introduced in R2014b

Hit Crossing

Detect crossing point

Library

Discontinuities



Description

The Hit Crossing block detects when the input reaches the **Hit crossing offset** parameter value in the direction specified by the **Hit crossing direction** property.

The block accepts one input of type `double`. If you select the **Show output port** check box, the block output indicates when the crossing occurs. If the input signal is exactly the value of the offset value after the hit crossing is detected, the block continues to output a value of 1. If the input signals at two adjacent points bracket the offset value (but neither value is exactly equal to the offset), the block outputs a value of 1 at the second time step. If the **Show output port** check box is *not* selected, the block ensures that the simulation finds the crossing point but does not generate output. If the input signal is constant and equal to the offset value, the block outputs 1 only if the **Hit crossing direction** property is set to `either`.

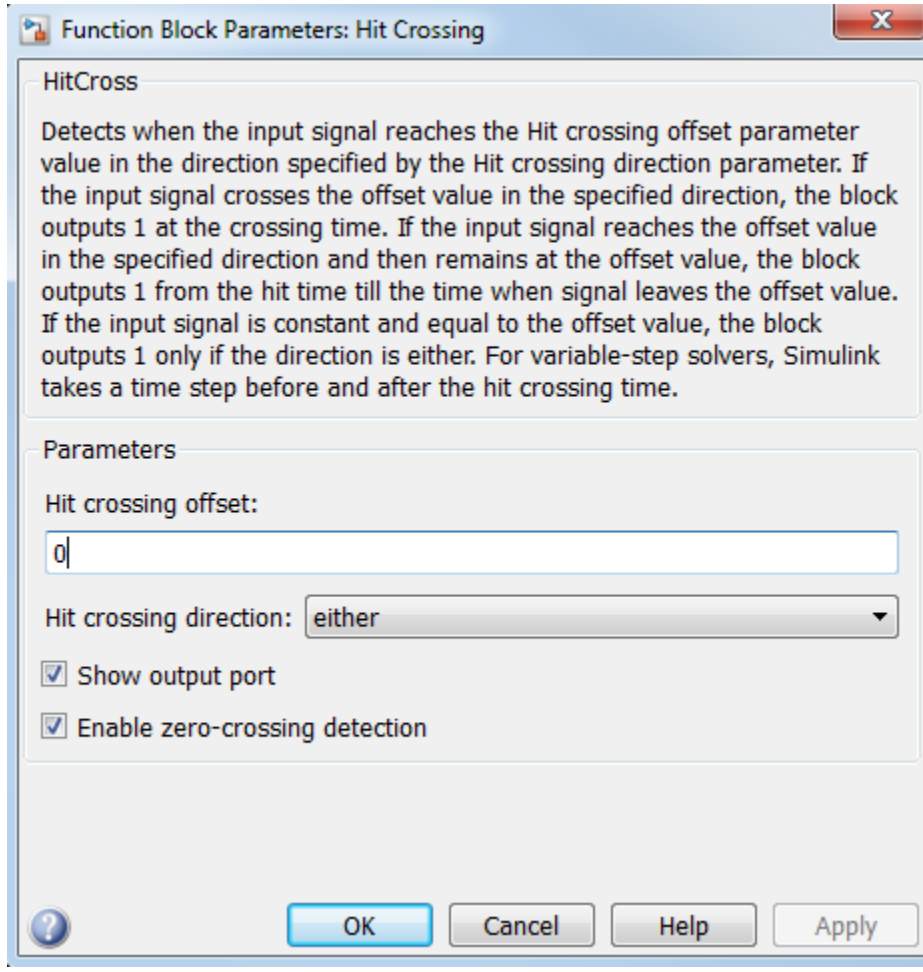
When the block's **Hit crossing direction** property is set to `either`, the block serves as an "Almost Equal" block, useful in working around limitations in finite mathematics and computer precision. Used for these reasons, this block might be more convenient than adding logic to your model to detect this condition.

When the block's **Hit crossing direction** property is set to `either` and the model uses a fixed-step solver, the block has the following behavior. If the output signal is 1, the block sets the output signal to 0 at the next time step, unless the input signal equals the offset value.

Data Type Support

The Hit Crossing block outputs a signal of type `Boolean` if Boolean logic signals are enabled (see "Implement logic signals as Boolean data (vs. double)"). Otherwise, the block outputs a signal of type `double`.

Parameters 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.

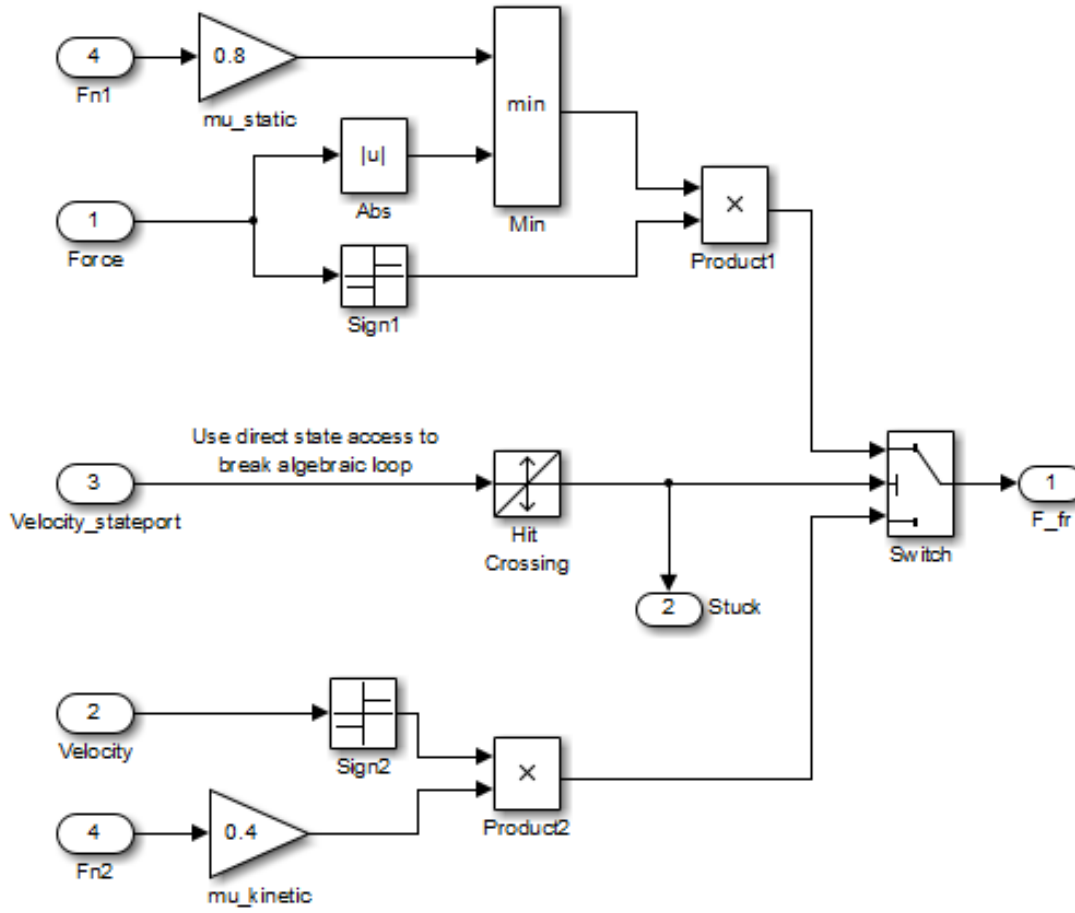
Sample time

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

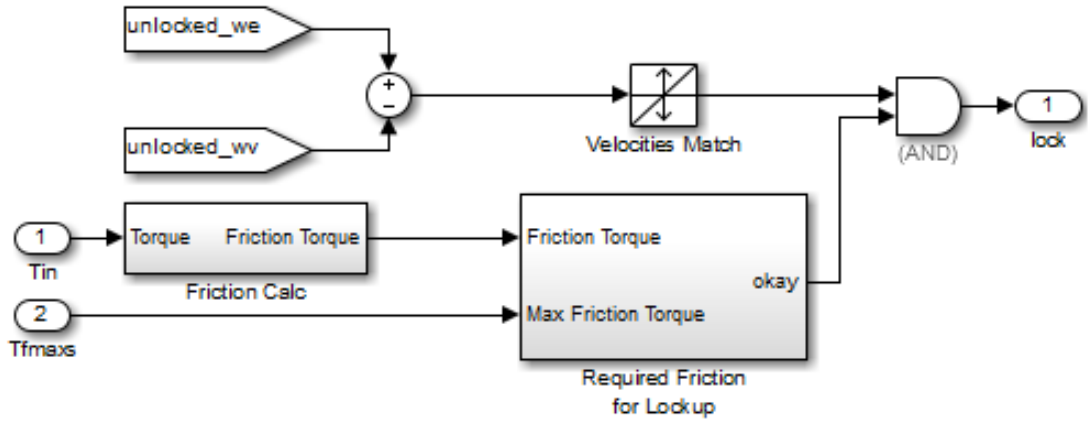
Examples

The `sldemo_hardstop` and `sldemo_clutch` models show how you can use the Hit Crossing block.

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



In the `sldemo_clutch` model, the Hit Crossing block is in the Friction Mode Logic/ Lockup Detection subsystem.



Characteristics

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

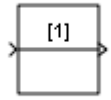
Introduced before R2006a

IC

Set initial value of signal

Library

Signal Attributes



Description

The IC block sets the initial condition of the signal at its input port, for example, the value of the signal at the simulation start time (t_{start}). The block does this by outputting the specified initial condition when you start the simulation, regardless of the actual value of the input signal. Thereafter, the block outputs the actual value of the input signal.

Note: If an IC block has a nonzero sample time offset (t_{offset}), the IC block outputs its initial value at time `t`,

$$t = n * t_{period} + t_{offset}$$

where `n` is the smallest integer such that $t \geq t_{start}$.

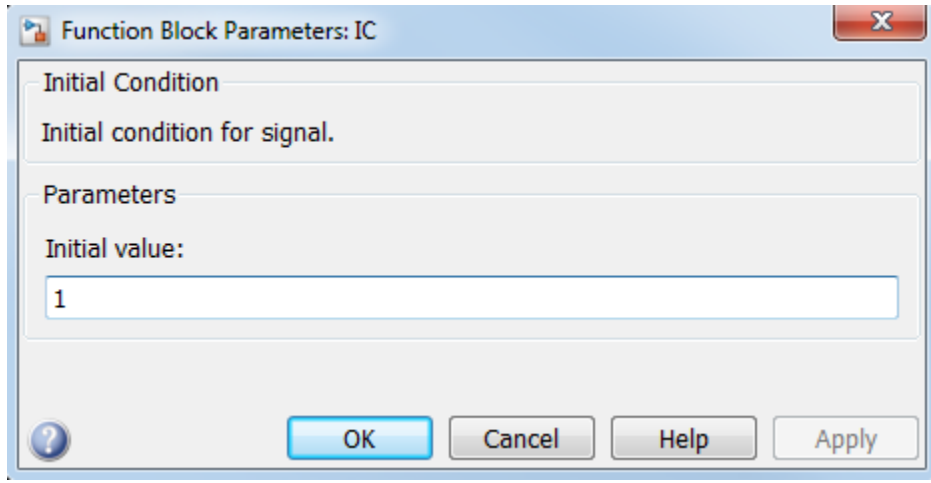
That is, the IC block outputs its initial value the first time blocks with sample time $[t_{period}, t_{offset}]$ execute, which can be after t_{start}.

The IC block is useful for providing an initial guess for the algebraic state variables in a loop. For more information, see “Algebraic Loops”.

Data Type Support

The IC block accepts and outputs signals of any Simulink built-in and fixed-point data type. The **Initial value** parameter accepts any built-in data type that Simulink supports. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



Initial value

Specify the initial value for the input signal.

Sample time

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

Examples

The following examples show how to use the `IC` block:

- `sldemo_bounce`
- `sldemo_hardstop`
- `sldemo_enginewc`

Characteristics

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

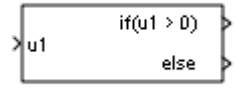
Introduced before R2006a

If

Model `if-else` control flow

Library

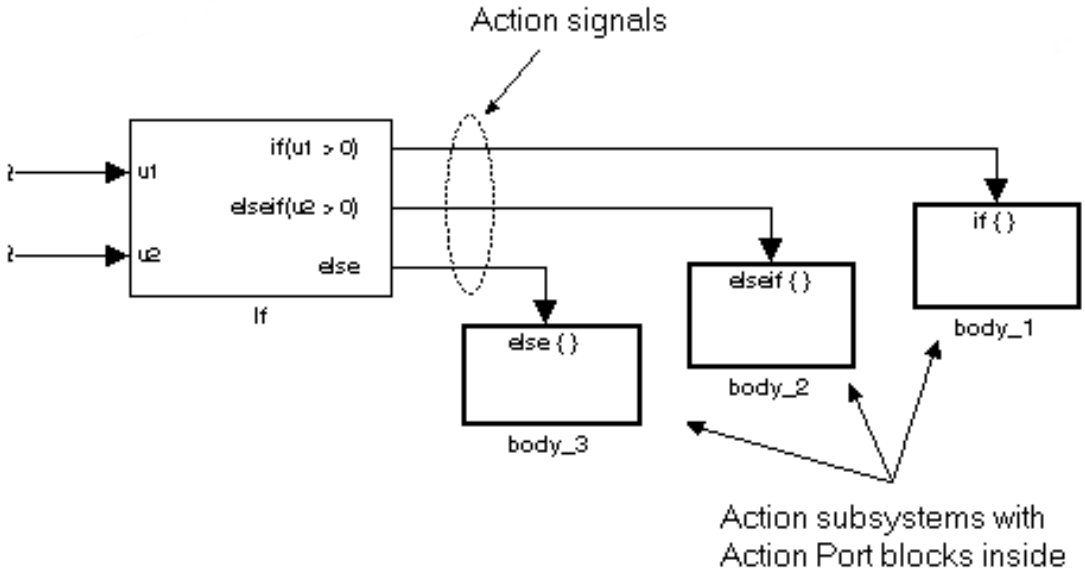
Ports & Subsystems



Description

The If block, along with the `If Action Subsystem` block containing an `Action Port`, implements standard C-like `if-else` logic.

The following shows a completed `if-else` control flow statement.



In this example, the inputs to the If block determine the values of conditions represented as output ports. Each output port is attached to an `If Action Subsystem`, 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.

Limitations

The If block has the following limitations:

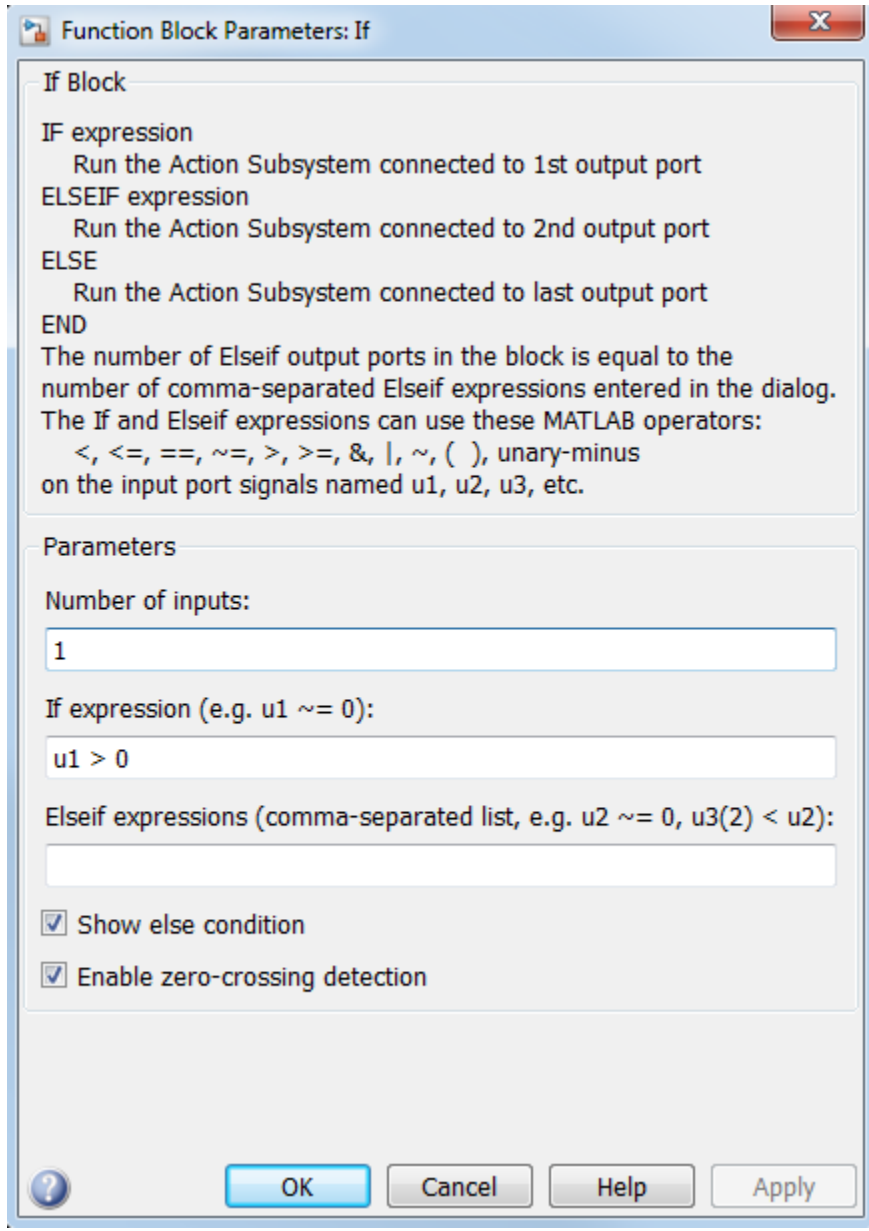
- You cannot tune an if or elseif expression during simulation in Normal or Accelerator mode (see "How Acceleration Modes Work"), or in generated code. The If block does not support tunable parameters. To implement tunable if/else expressions, tune the expression outside the If block. For example, use the Relational operator block to evaluate the expression outside or add the tunable parameter as an input to the If block.
- The If block does not support custom storage classes. See "Custom Storage Classes" in the Embedded Coder documentation.
- The **If expression** and **Elseif expressions** cannot accept certain operators, such as `+`, `-`, `*`, and `/`. See If Expression and Elseif Expressions in "Parameters and Dialog Box" on page 1-901

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.

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 include only the operators `<, <=, ==, ~=, >, >=, &, |, ~, (), unary-minus`, and cannot include operators such as `+, -, *, /`, and `^`. The `If Action Subsystem` attached to the `if` port executes if its condition is true. The expression must not contain data type expressions, for example, `int8(6)`, and must not reference workspace variables whose data type is other than `double` or `single`.

Note: There are limitations to the tunability of the **If expression**. See "Limitations" on page 1-899 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 include only the operators `<, <=, ==, ~=, >, >=, &, |, ~, (), unary-minus`, and cannot include operators such as `+, -, *, /`, and `^`. The `If Action Subsystem` attached to an `elseif` port executes if its condition is true and all of the `if` and `elseif` conditions are false. The expression must not contain data type expressions, for example, `int8(6)`, and must not reference workspace variables whose data type is other than `double` or `single`.

Note: There are limitations to the tunability of **Elseif expressions**. See "Limitations" on page 1-899 for more information.

Show else condition

If you select this check box, an `else` port is created. The If Action subsystem attached to the `else` port executes if the `if` port and all the `elseif` ports are false.

Enable zero-crossing detection

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

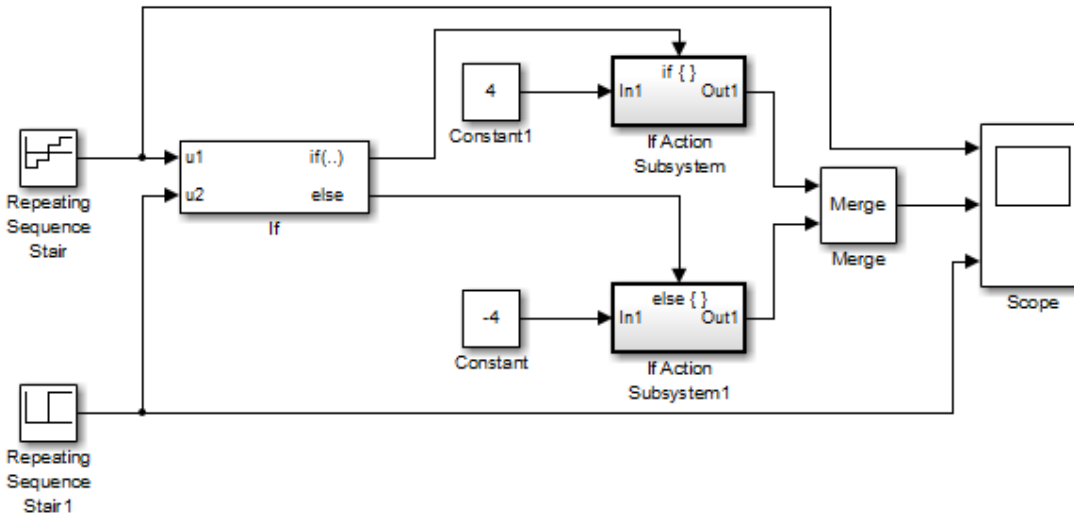
Sample time

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

Examples

The If block does not directly support fixed-point data types. However, you can use the `Compare To Constant` block to work around this limitation.

For example, consider the following floating-point model:

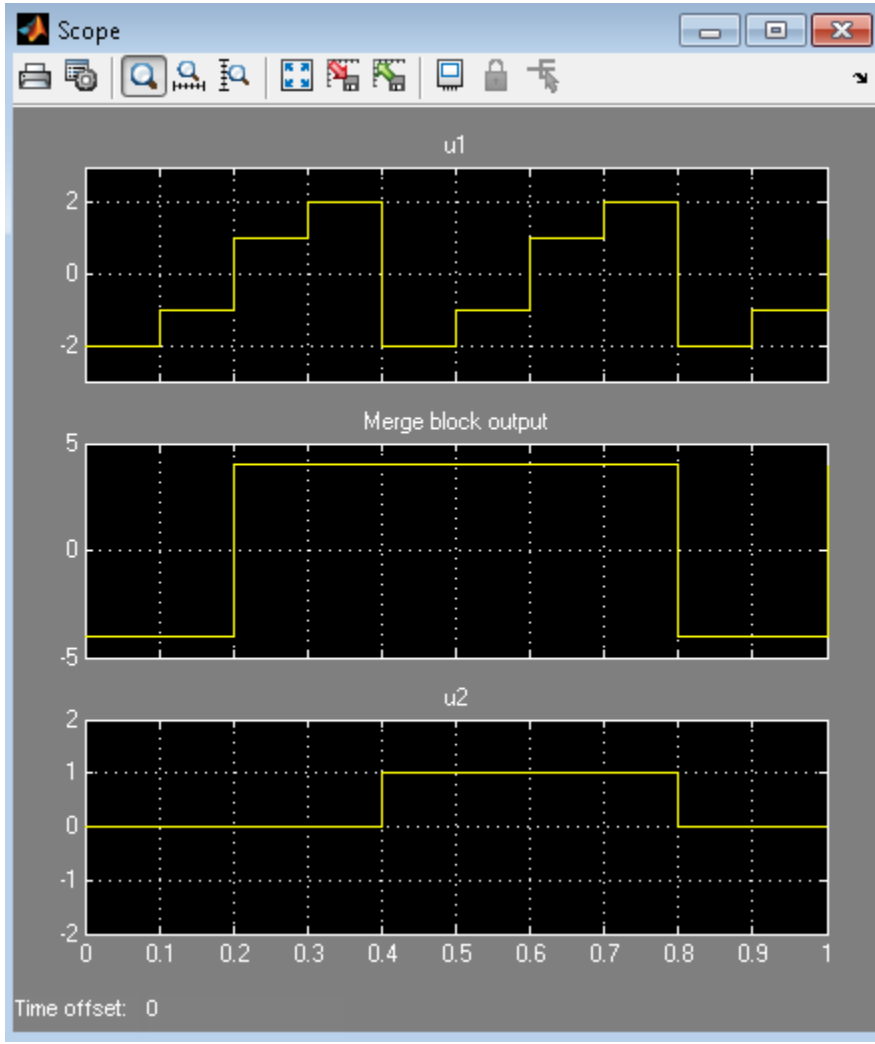


In this model, the If Action subsystems use their default configurations. The block and simulation parameters for the model are set to their default values except as follows:

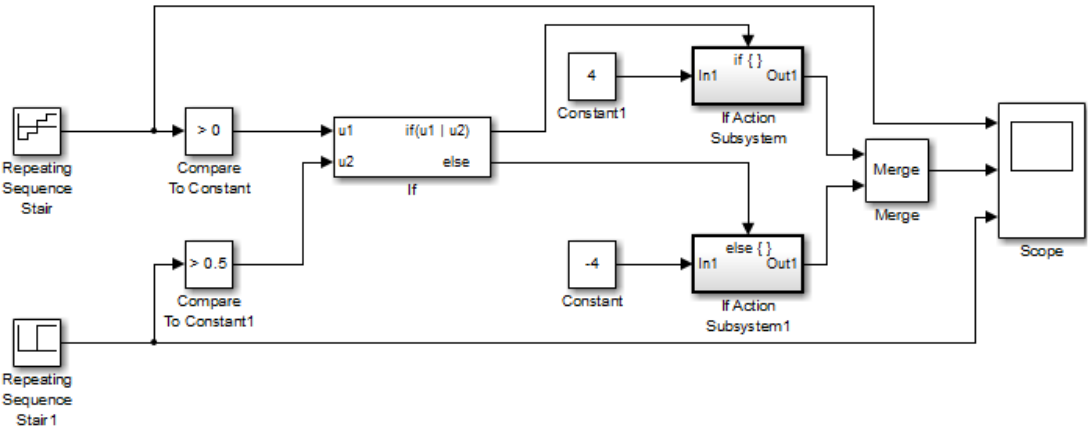
Block or Dialog Box	Parameter	Setting
Configuration Parameters — **Solver** pane	**Start time**	`0.0`

Block or Dialog Box	Parameter	Setting
	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`:



You can implement this block diagram as a model with fixed-point data types:



The Repeating Sequence Stair blocks now output fixed-point data types.

The Compare To Constant blocks implement two parts of the **If expression** that is used in the If block in the floating-point version of the model, `(u1 > 0)` and `(u2 > 0.5)`. The `OR` operation, `(u1|u2)`, can still be implemented inside the If block. For a fixed-point model, the expression must be partially implemented outside of the If block as it is here.

The block and simulation parameters for the fixed-point model are the same as for the floating-point model with the following exceptions and additions:

Block	Parameter	Setting
Compare To Constant	**Operator**	`>`
	Constant value	`0`
	Output data type mode	`Boolean`
	Enable zero-crossing detection	off
Compare To Constant1	**Operator**	`>`
	Constant value	`0.5`
	Output data type mode	`Boolean`
	Enable zero-crossing detection	off
If	**Number of inputs**	`2`

Block	Parameter	Setting
	If expression	u1 | u2

Characteristics

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

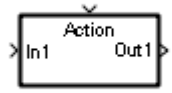
Introduced before R2006a

If Action Subsystem

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 “Create an Action Subsystem”, `If` block and Modeling with Control Flow Blocks in the “Creating a Model” chapter of the Simulink documentation.

Characteristics

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

Introduced before R2006a

Increment Real World

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.

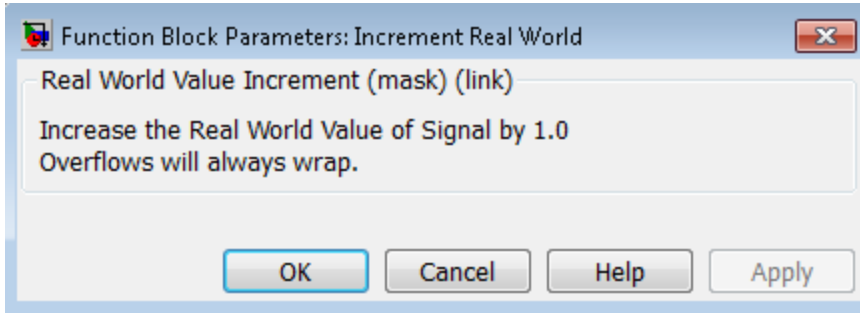
Data Type Support

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



Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Decrement Real World`, `Increment Stored Integer`

Introduced before R2006a

Increment Stored Integer

Increase stored integer value of signal by one

Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Increment Stored Integer block increases the stored integer value of a signal by one.

Floating-point signals also increase by one, and overflows always wrap.

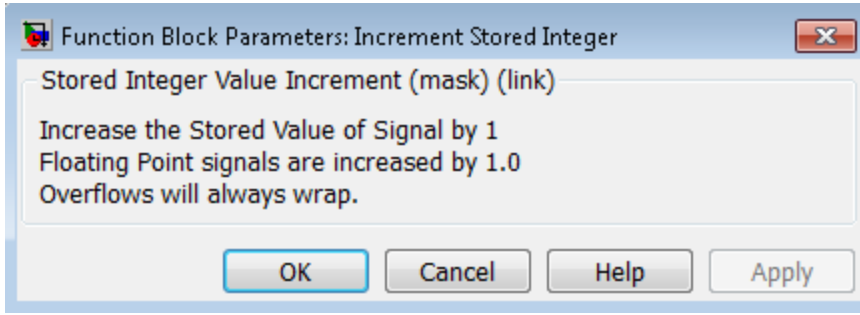
Data Type Support

The Increment Stored Integer block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

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

Parameters and Dialog Box



Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Decrement Stored Integer`, `Increment Real World`

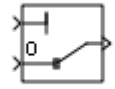
Introduced before R2006a

Index Vector

Switch output between different inputs based on value of first input

Library

Signal Routing



Description

The Index Vector block is an implementation of the Multiport Switch block. See Multiport Switch for more information.

Characteristics

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

Introduced before R2006a

Inport

Create input port for subsystem or external input

Library

Ports & Subsystems, Sources

Description

Inport blocks are the links from outside a system into the system.

Simulink software assigns Inport block port numbers according to these rules:

- It automatically numbers the Inport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Inport block, the label is the next available number.
- If you delete an Inport block, other port numbers are automatically renumbered to ensure that the Inport blocks are in sequence and that no numbers are omitted.
- If you copy an Inport block into a system, its port number is *not* renumbered unless its current number conflicts with an Inport block already in the system. If the copied Inport block port number is not in sequence, renumber the block. Otherwise, you get an error message when you run the simulation or update the block diagram.

You can specify the dimensions of the input to the Inport block using the **Port dimensions** parameter. Entering a value of `-1` lets Simulink determine the port dimension.

The **Sample time** parameter is the rate at which the signal is coming into the system. A value of `-1` causes the block to inherit its sample time from the block driving it. You might need to set this parameter for:

- Inport blocks in a top-level system.
- Models with blocks where Simulink cannot determine the sample time, but these blocks drive Inport blocks.

For more information, see “Specify Sample Time”.

Inport Blocks in a Top-Level System

Inport blocks in a top-level system have two uses:

- To supply external inputs from the workspace, use the Configuration Parameters dialog box (see “Comparison of Signal Loading Techniques”) 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.

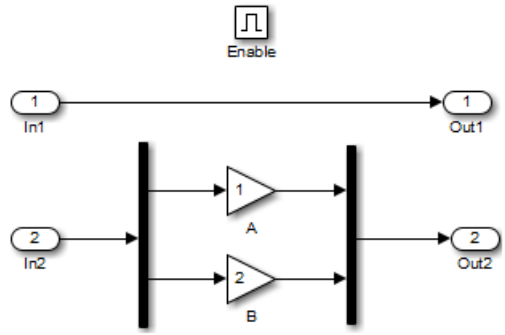
Note: If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite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

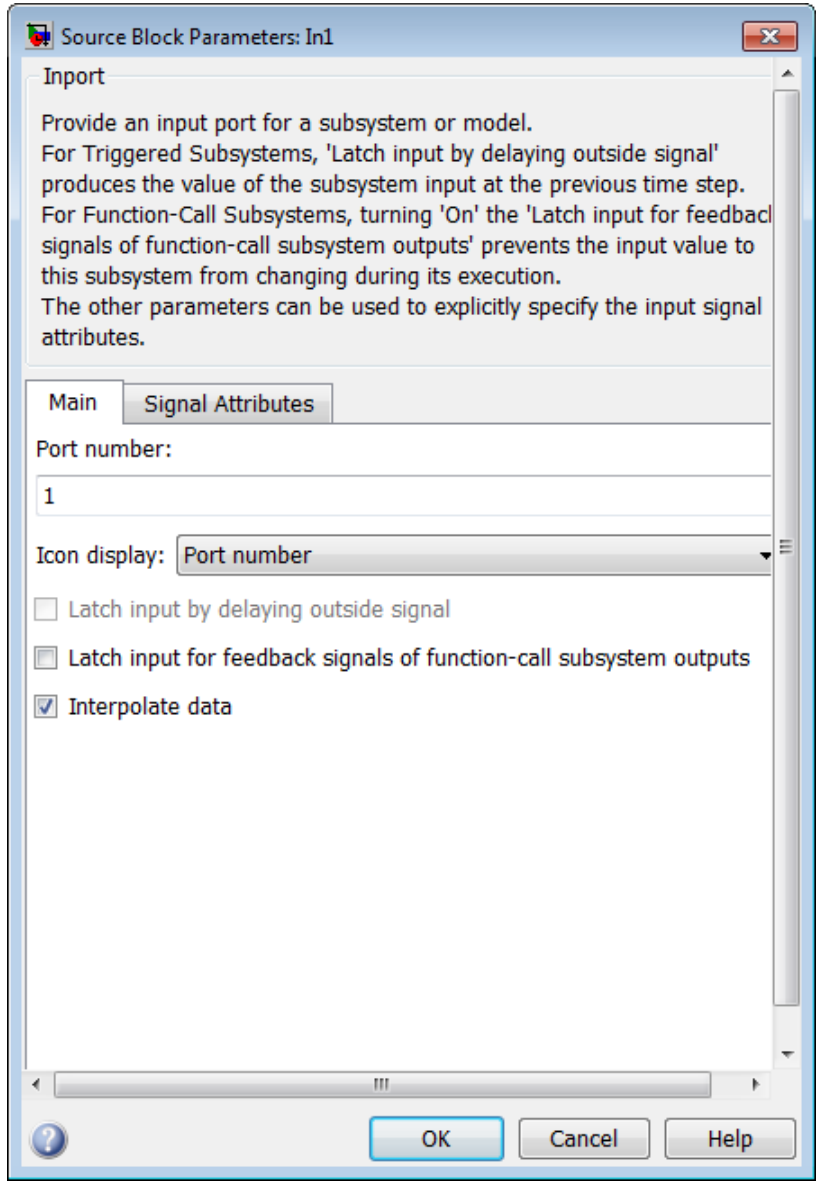
The elements of a signal array connected to a root-level Inport block must be of the same numeric and data types. Signal elements connected to a subsystem input port can be of differing numeric and data types, except in the following circumstance: If the subsystem contains an Enable, Trigger, or Atomic 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:



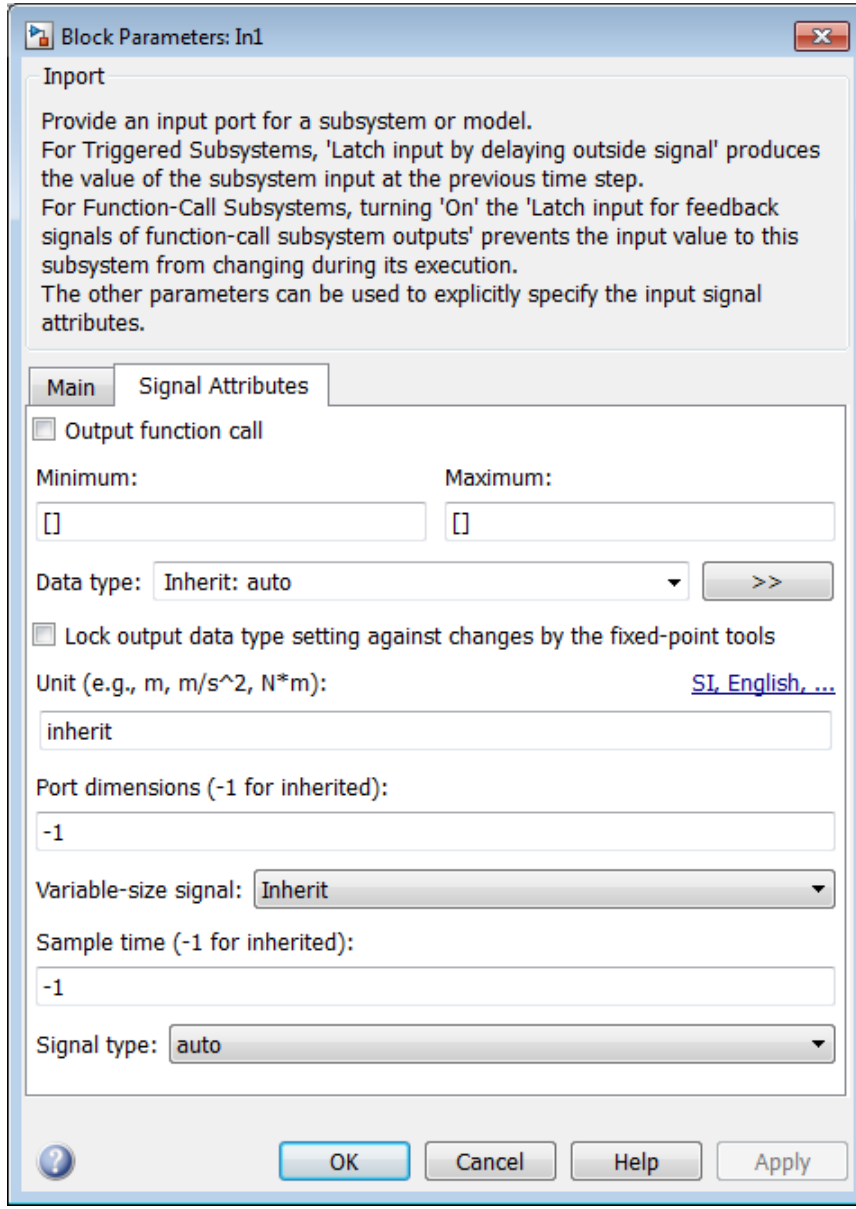
In this example, the elements of a signal vector connected to `In1` must be of the same type. The elements connected to `In2`, however, can be of differing types.

Parameters and Dialog Box

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



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



- “Port number” on page 1-539

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 6-101 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 6-101 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

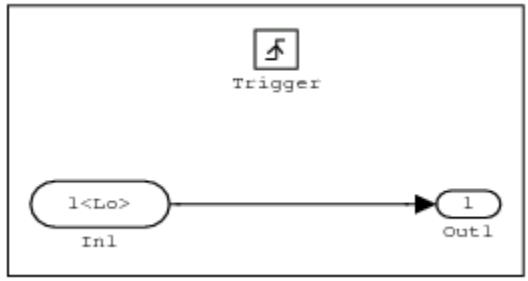
Output the value of the input signal at the previous time step.

 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 6-101 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

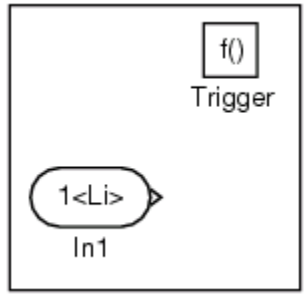
Latch the input value.

☐ 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>`.



Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Interpolate data

When loading data from the workspace, cause the block to linearly interpolate and extrapolate output at time steps for which no corresponding data exists.

To load discrete signal data from the workspace, in the Inport block dialog box:

1. Set the **Sample time** parameter to a discrete value, such as 2.
2. Clear the **Interpolate data** parameter.

Specifying the discrete sample time causes the simulation to have hit times exactly at those instances when the discrete data is sampled. You only need to specify the data values, not time values.

Turning interpolation off avoids unexpected data values at other simulation time points as a result of double precision arithmetic processing. For more information, see "Import Data to Test a Discrete Algorithm".

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 6-101 for the command-line information.

Connect Input

To import, visualize, and map signal and bus data to root-level inports, click this button. The Root Inport Mapping tool displays.

Dependency

This button appears only if this block is a root inport block.

Output function call

Specify that the input signal is outputting a function-call trigger signal.

Settings

Default: Off

 On

Input signal is a function-call trigger signal.

 Off

Input signal is not a function-call trigger signal.

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 6-101 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 6-101 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 6-101 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 6-101 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 “Control Signal Data Types”.

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 6-101 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.

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.

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**

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.

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.

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.

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

- All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. For referenced models, buses are single-rate. For details, see "Connect Multi-Rate Buses to Referenced Models".
- For the top model in a model reference hierarchy, code generation creates a C structure to represent the bus signal output by this block.
- For referenced models, select this option to create a C structure. Otherwise, code generation creates an argument for each leaf element of the bus used in the referenced model.

Dependencies

Selecting **Data type** > `Bus: <object name>` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

See Also

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

Unit (e.g., m, m/s^2, N*m)

Specify physical unit of the input signal to the block.

Settings

Default: `inherit`

To specify a unit, begin typing in the text box. As you type, the parameter displays potential unit string matches. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

- If a `Unit System Configuration` block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.

- If a `Unit System Configuration` block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 6-101 for the command-line information.

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 6-101 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 6-101 for the command-line information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Outport`

`Asynchronous Task Specification`

Introduced before R2006a

Integrator, Integrator Limited

Integrate signal

Library

Continuous


Description

The `Integrator` block outputs the value of the integral of its input signal with respect to time.

The `Integrator Limited` block is identical to the Integrator block with the exception that the output of the block is limited based on the upper and lower saturation limits. See "Limiting the Integral" on page 1-953 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:

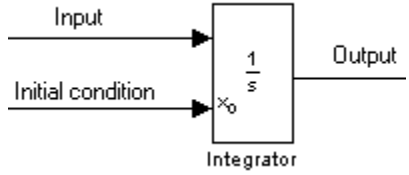
- Define upper and lower limits on the integral
- Create an input that resets the block's output (state) to its initial value, depending on how the input changes
- Create an optional state output so that the value of the block's output can trigger a block reset

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.



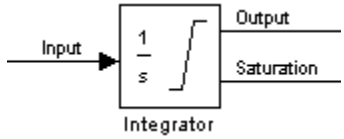
Note If the integrator limits its output (see “Limiting the Integral” on page 1-953), the initial condition must fall inside the integrator's saturation limits. If the initial condition is outside the block saturation limits, the block displays an error message.

Limiting the Integral

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. This action causes the block to function as a limited integrator. When the output reaches the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The block determines output as follows:

- When the integral is less than or equal to the **Lower saturation limit**, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than or equal to the **Upper saturation limit**, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port.



The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- –1 indicates that the lower limit is being applied.

When you select this check box, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

Note: For the `Integrator Limited` block, by default, **Limit output** is selected, **Upper saturation limit** is set to `1`, and **Lower saturation limit** is set to `0`.

Wrapping Cyclic States

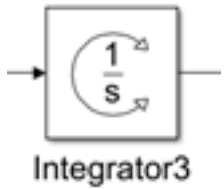
Several physical phenomena are cyclic, periodic, or rotary in nature. Objects or machinery that exhibit rotational movement and oscillators are examples of such phenomena.

Modeling these phenomena in Simulink involves integrating the rate of change of the periodic or cyclic signals to obtain the state of the movement.

The drawback with this approach, however, is that over long simulation time spans, the states representing periodic or cyclic signals integrate to large values. Further, computing the sine or cosine of these signals takes an increasingly large amount of time because of angle reduction. The large signals values also negatively impact solver performance and accuracy.

One approach for overcoming this drawback is to reset the angular state to `0` when it reaches 2π (or to $-\pi$ when it reaches π, for numerical symmetry). This approach improves the accuracy of sine and cosine computations and reduces angle reduction time. But it also requires zero-crossing detection and introduces solver resets, which slow down the simulation for variable step solvers, particularly in large models.

To eliminate solver resets at wrap points, the Integrator block supports wrapped states that you can enable by checking **Wrap state** on the block parameter dialog box. When you enable **Wrap state**, the block icon changes to indicate that the block has wrapping states.



Simulink allows wrapping states that are bounded by upper and lower values parameters of the wrapped state. The algorithm for determining wrapping states is given by:

$$y = \begin{cases} x \\ x - (x_u - x_l)\left\lfloor \dfrac{x - x_l}{x_u - x_l} \right\rfloor \end{cases} \begin{matrix} x \in [x_l, x_u) \\ \text{otherwise} \end{matrix}$$

where:

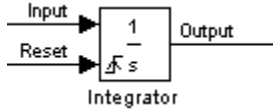
- x_l is the lower value of the wrapped state.
- x_u is the upper value of the wrapped state.
- y is the output.

The support for wrapping states provides these advantages.

- It eliminates simulation instability when your model approaches large angles and large state values.
- It reduces the number of solver resets during simulation and eliminates the need for zero-crossing detection, improving simulation time.
- It eliminates large angle values, speeding up computation of trigonometric functions on angular states.
- It improves solver accuracy and performance and enables unlimited simulation time.

Resetting the State

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the **External reset** choices. A trigger port appears below the block's input port and indicates the trigger type.



- Select `rising` to reset the state when the reset signal rises from a 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.

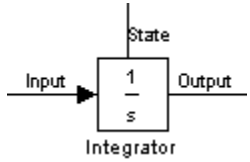
- Select `level` to reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.
- Select `level hold` to reset the state when the reset signal is nonzero at the current time step.

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.



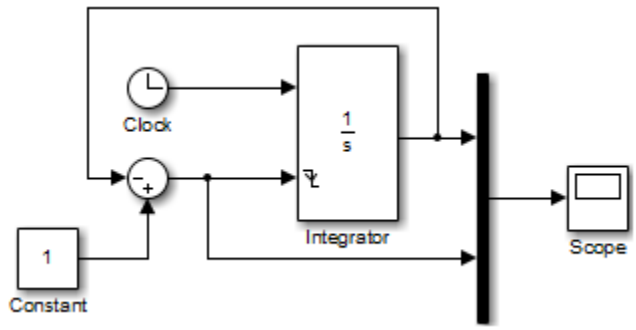
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. 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 1-957)
- Handing off a state from one enabled subsystem to another (see "Handing Off States Between Enabled Subsystems" on page 1-958)

Note When updating a model, Simulink checks that the state port applies to one of these two scenarios. If not, an error message appears. Also, you cannot log the output of this port in a referenced model that executes in Accelerator mode. If logging is enabled for the port, Simulink generates a "signal not found" warning during execution of the referenced model.

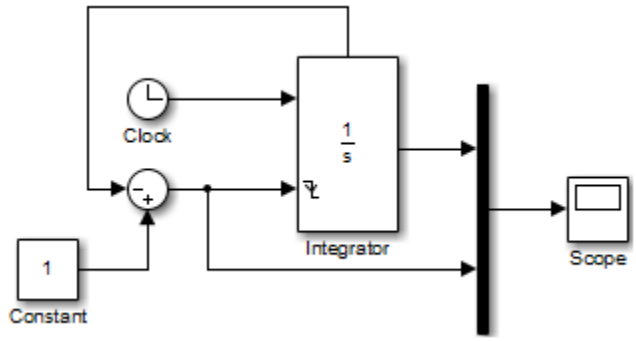
Creating Self-Resetting Integrators

The Integrator block's state port helps you avoid an algebraic loop when creating an integrator that resets itself based on the value of its output. Consider, for example, the following model.



This model tries to create a self-resetting integrator by feeding the integrator's output, subtracted from 1, back into the integrator's reset port. However, the model creates an algebraic loop. To compute the integrator block's output, Simulink software needs to know the value of the block's reset signal, and vice versa. Because the two values are mutually dependent, Simulink software cannot determine either. Therefore, an error message appears if you try to simulate or update this model.

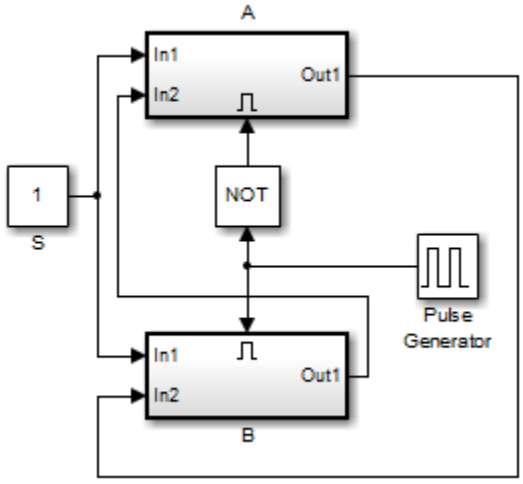
The following model uses the integrator's state port to avoid the algebraic loop.



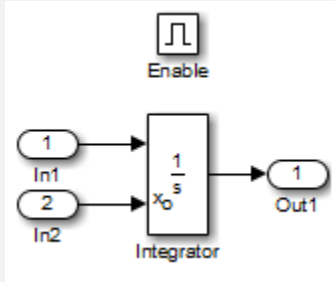
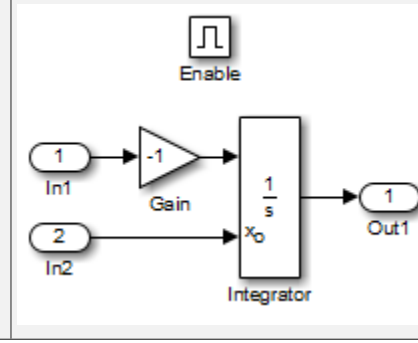
In this version, the value of the reset signal depends on the value of the state port. The value of the state port is available earlier in the current time step than the value of the integrator block's output port. Therefore, Simulink can determine whether the block needs to be reset before computing the block's output, thereby avoiding the algebraic loop.

Handing Off States Between Enabled Subsystems

The state port helps you avoid an algebraic loop when passing a state between two enabled subsystems. Consider, for example, the following model.



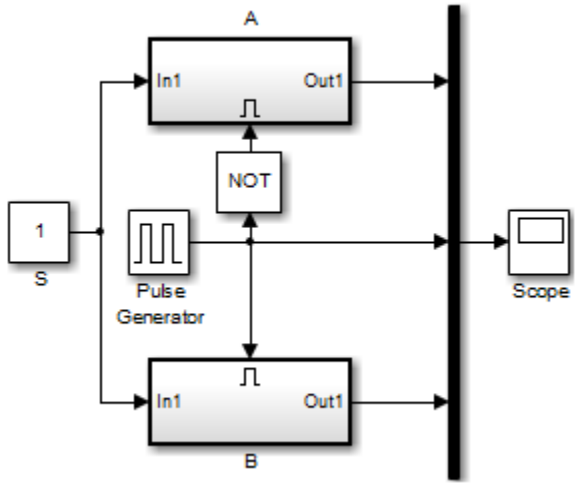
The enabled subsystems, A and B, contain the following blocks:

Subsystem A	Subsystem B
	

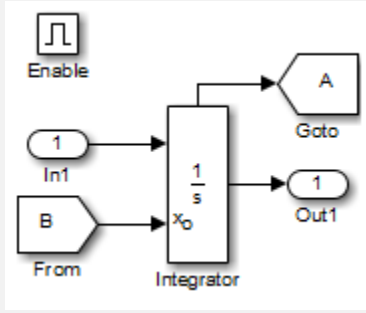
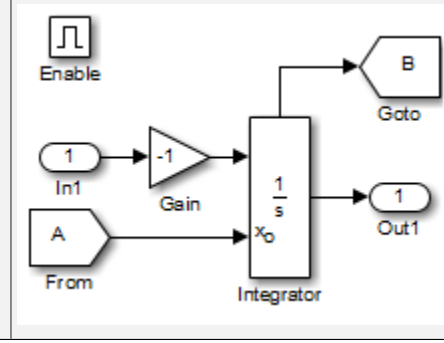
In this model, a constant input signal drives two enabled subsystems that integrate the signal. A pulse generator generates an enabling signal that causes execution to alternate between the two subsystems. The enable port of each subsystem is set to reset, which causes the subsystem to reset its integrator when it becomes active. Resetting the integrator causes the integrator to read the value of its initial condition port. The initial condition port of the integrator in each subsystem is connected to the output port of the integrator in the other subsystem.

This connection is intended to enable continuous integration of the input signal as execution alternates between two subsystems. However, the connection creates an algebraic loop. To compute the output of A, Simulink needs to know the output of B, and vice versa. Because the outputs are mutually dependent, Simulink cannot compute the output values. Therefore, an error message appears if you try to simulate or update this model.

The following version of the same model uses the integrator state port to avoid creating an algebraic loop when handing off the state.



The enabled subsystems, A and B, contain the following blocks:

Subsystem A	Subsystem B
	

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.

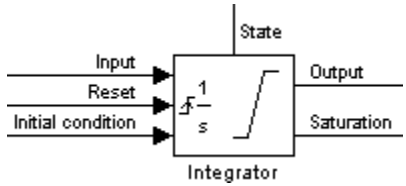
Note Simulink does not permit three or more enabled subsystems to hand off a model state. If Simulink detects that a model is handing off a state among more than two enabled subsystems, it generates an error.

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 "Error Tolerances for Variable-Step Solvers") to compute the output of the Integrator block. If this value does not provide sufficient error control, specify a more appropriate value in the **Absolute tolerance** field of the Integrator block dialog box. The value that you specify is used to compute all the block outputs.

Selecting All Options

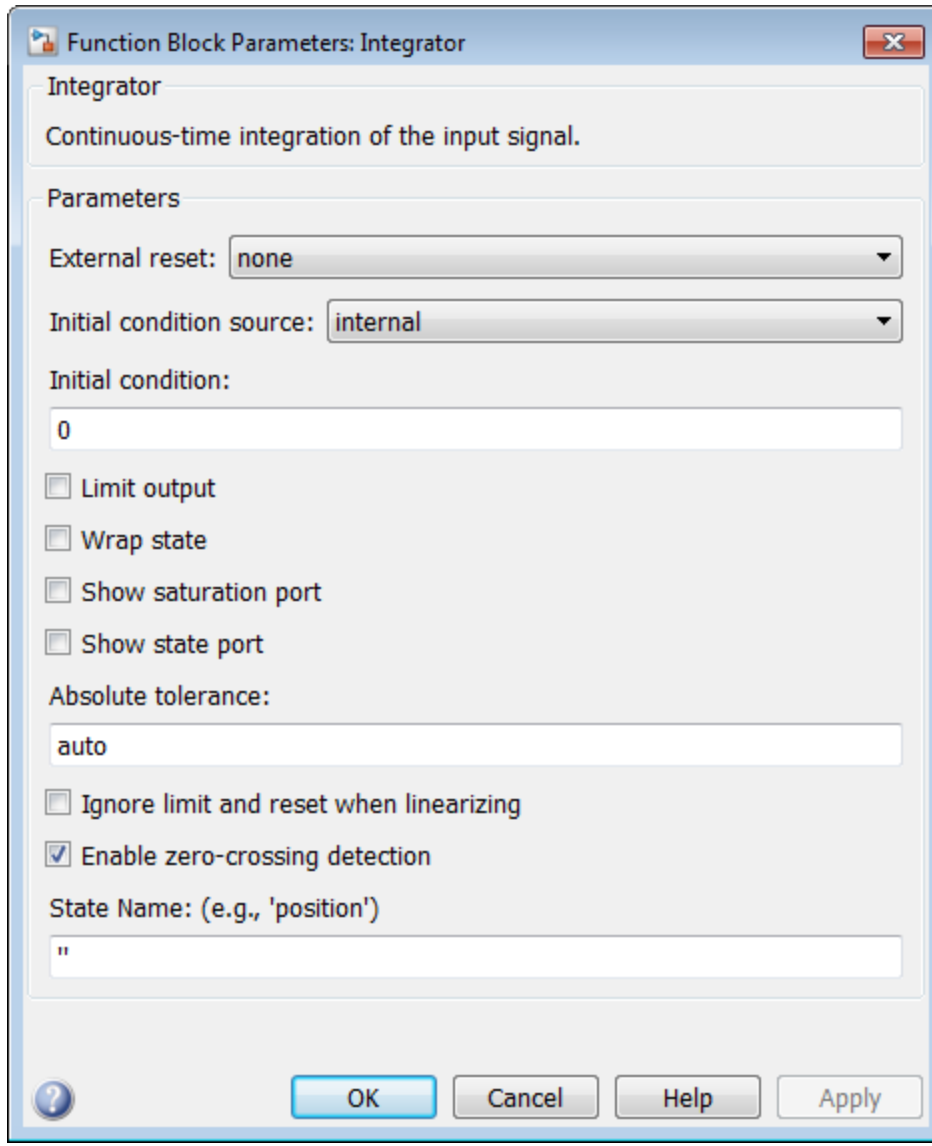
When you select all options, the block icon looks like this.



Data Type Support

The Integrator block accepts and outputs signals of type `double` on its data ports. The external reset port accepts signals of type `double` or `Boolean`.

Parameters 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

Parameter: `ExternalReset`
Type: string
Value: `'none' | 'rising' | 'falling' | 'either' | 'level' | 'level hold'`
Default: `'none'`

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

Parameter: `InitialConditionSource`
Type: string
Value: `'internal' | 'external'`
Default: `'internal'`

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

Parameter: `InitialCondition`
Type: scalar or vector
Value: `'0'`
Default: `'0'`

Limit output

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

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

Parameter: `LimitOutput`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `UpperSaturationLimit`
Type: scalar or vector
Value: `'inf'`
Default: `'inf'`

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

Parameter: `LowerSaturationLimit`
Type: scalar or vector
Value: `'-inf'`
Default: `'-inf'`

Wrap state

Enable wrapping of states between the **Wrapped state upper value** and **Wrapped state lower value** parameters. Enabling wrap states eliminates the need for zero-crossing detection, reduces solver resets, improves solver performance and accuracy, and increases simulation time span when modeling rotary and cyclic state trajectories.

Settings

Default: `off`

 On

Enable wrap states between the **Wrapped state upper value** and **Wrapped state lower value** parameters.

If you specify **Wrapped state upper value** as `inf` and **Wrapped state lower value** as `-inf`, wrapping will never occur.

 Off

Do not enable wrap states.

Dependencies

This parameter enables **Wrapped state upper value**.

This parameter enables **Wrapped state lower value**.

Command-Line Information

Parameter: `WrapState`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Wrapped state upper value

Specify the upper value for the wrap state.

Settings

Default: `'pi'`

Dependencies

Wrap state enables this parameter.

Command-Line Information

Parameter: `WrappedStateUpperValue`
Type: scalar or vector
Value: `'2*pi'`
Default: `'pi'`

Wrapped state lower value

Specify the lower value for the wrap state.

Settings

Default: `-pi`

Dependencies

Wrap state enables this parameter.

Command-Line Information

Parameter: `WrappedStateLowerValue`
Type: scalar or vector
Value: `'0'`
Default: `'-pi'`

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

Parameter: `ShowSaturationPort`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `ShowStatePort`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, –1, a positive real scalar or vector.
- If you enter `auto` or –1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Command-Line Information

Parameter: `AbsoluteTolerance`
Type: string, scalar, or vector
Value: `'auto' | '-1' |` any positive real scalar or vector
Default: `'auto'`

Ignore limit and reset when linearizing

Cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the block's reset and output limitation options.

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

Parameter: `IgnoreLimit`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Enable zero-crossing detection

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

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

Parameter: `ZeroCross`
Type: string
Value: `'off' | 'on'`
Default: `'on'`

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

Parameter: `ContinuousStateAttributes`
Type: string
Value: `' '` | user-defined
Default: `' '`

Examples

The following example models show how to use the Integrator block:

- `sldemo_hardstop`
- `sldemo_suspn`

- `sldemo_wheelspeed_absbrake`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Yes, of the reset and external initial condition source ports
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled and you select the **Limit output** check box, one for detecting reset, one each to detect upper and lower saturation limits, and one when leaving saturation
Code Generation	Yes

See Also

`Discrete-Time Integrator`

Introduced before R2006a

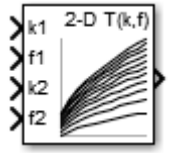
Interpolation Using Prelookup

Use precalculated index and fraction values to accelerate approximation of N-dimensional function

Library

Lookup Tables

Description



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. These two blocks have distributed algorithms. When combined together, they perform the same operation as the integrated algorithm in the n-D Lookup Table block. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility that can provide more efficient simulation and code generation. For more information, see "Efficiency of Performance" in the Simulink documentation.

Supported Block Operations

To use the Interpolation Using Prelookup block, you specify a set of table data values directly on the dialog box or feed values into the `T` input port. Typically, these table values correspond to the breakpoint data sets specified in Prelookup blocks. The Interpolation Using Prelookup block generates output by looking up or estimating table values based on index and interval fraction values fed from Prelookup blocks. Labels for the index and interval fraction appear as `k` and `f` on the Interpolation Using Prelookup block icon.

When inputs for index and interval fraction...	The Interpolation Using Prelookup block...
Map to values in breakpoint data sets	Outputs the table value at the intersection of the row, column, and higher dimension breakpoints
Do not map to values in breakpoint data sets, but are within range	Interpolates appropriate table values, using the **Interpolation method** you select
Do not map to values in breakpoint data sets, and are out of range	Extrapolates the output value, using the **Extrapolation method** you select

How The Block Interpolates a Subset of Table Data

You can use the **Number of sub-table selection dimensions** parameter to specify that interpolation occur only on a subset of the table data. To activate this interpolation mode, set this parameter to a positive integer. This value defines the number of dimensions to select, starting from the highest dimension of table data. Therefore, the value must be less than or equal to the **Number of table dimensions**.

Suppose that you have 3-D table data in your `Interpolation Using Prelookup` block. 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 selects the third dimension	Displays an input port with the label `sel1` that you use to select and interpolate 2-D tables
2	Interpolates the first dimension and selects the second and third dimensions	Displays two input ports with the labels `sel1` and `sel2` that you use to select and interpolate 1-D tables

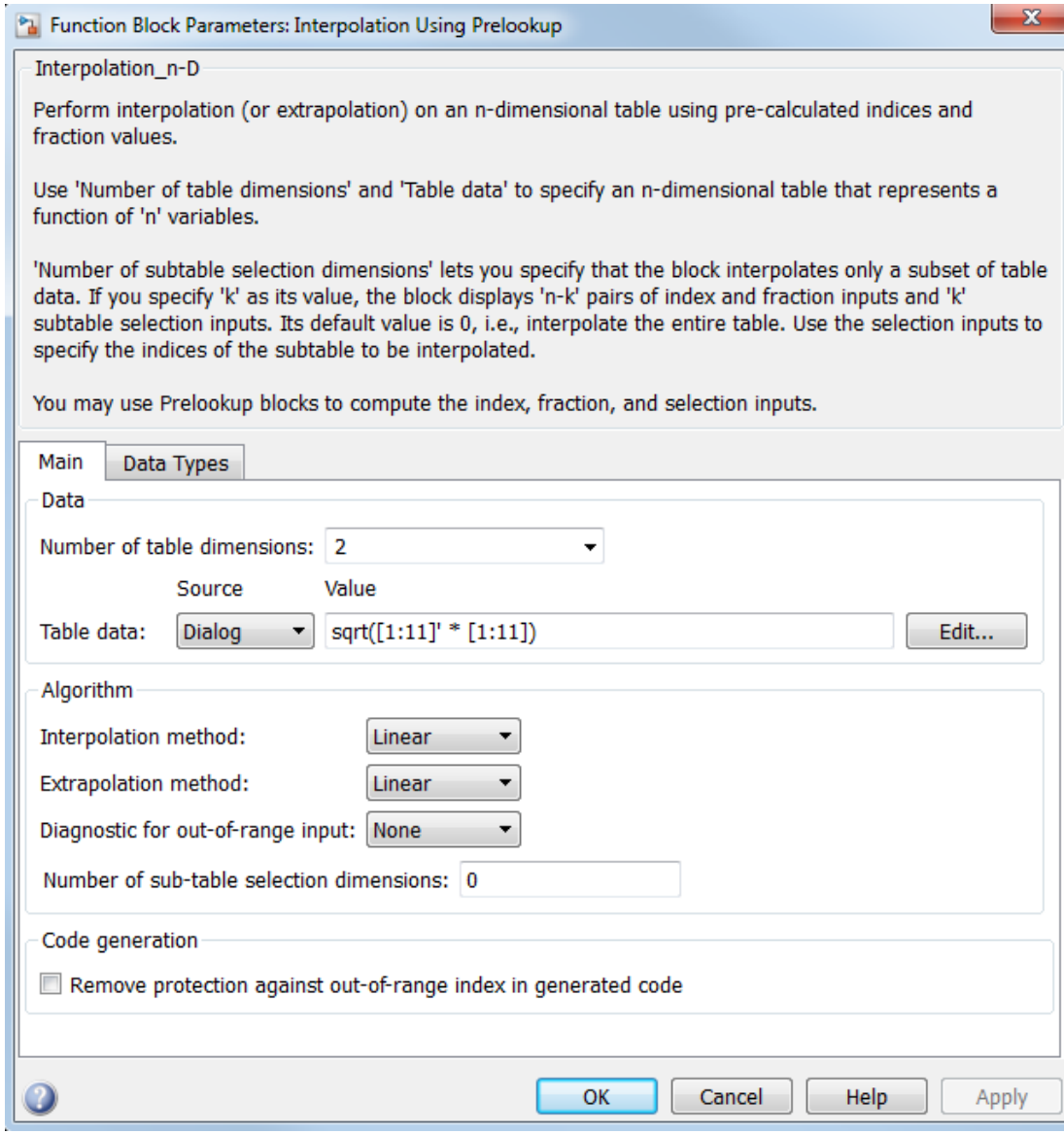
Subtable selection uses zero-based indexing. For an example of interpolating a subset of table data, type sldemo_bpcheck at the MATLAB command prompt.

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



- "Main tab" on page 1-984

- "Data Types tab" on page 1-987

Main tab

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 Lookup Tables" in the Simulink documentation).

Interpolation method

Select `Flat`, `Nearest`, or `Linear`. See "Interpolation Methods" in the Simulink documentation for more information.

Extrapolation method

Select `Clip` or `Linear`. See "Extrapolation Methods" in the Simulink documentation for more information. The **Extrapolation method** parameter is visible only when you select `Linear` as the **Interpolation method** parameter.

The Interpolation Using Prelookup block does not support `Linear` extrapolation when the input or output signals specify integer or fixed-point data types.

Valid index input may reach last index

Specify how block inputs for index (`k`) and interval fraction (`f`) access the last elements of *n*-dimensional table data. Index values are zero-based.

Check Box	Block Behavior
Selected	Returns the value of the last element in a dimension of its table when: • `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 Diagnostic 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 "Run Model Checks" 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 1-981.

Sample time

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

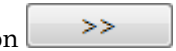
Data Types tab

Note: The parameters for table attributes (data type, minimum, and maximum) are not available when you set **Source** to `Input port`. In this case, the block inherits all table attributes from the `T` input port.

Table data > Data Type

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

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

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the table data type.

Tip Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two `Interpolation Using Prelookup` blocks with different output data types
- Sharing of custom storage table data in Simulink Coder generated code for blocks with different output data types

---

Table data > Minimum

Specify the minimum value for table data. The default value is `[]` (unspecified).

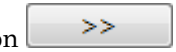
Table data > Maximum

Specify the maximum value for table data. The default value is `[]` (unspecified).

Intermediate results > Data Type

Specify the intermediate results data type. You can set it to:

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

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the intermediate results data type.

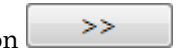
Tip Use this parameter to specify higher precision for internal computations than for table data or output data.

Output > Data Type

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

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

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

See "Control Signal Data Types" in the Simulink User's Guide for more information.

Output > Minimum

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

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

Internal rule priority

Specify the internal rule for intermediate calculations. Select `Speed` for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

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

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

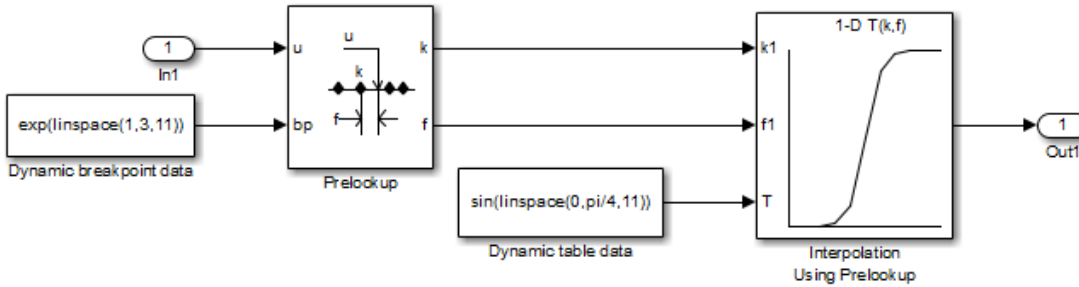
Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can

detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

In the following model, a `Constant` block feeds the table data values to the `T` input port of the `Interpolation Using Prelookup` block.



The Interpolation Using Prelookup block inherits the following table attributes from the `T` input port:

Table Attribute	Value
Minimum	`-Inf`
Maximum	`Inf`
Data type	`single`

Similarly, a Constant block feeds the breakpoint data set to the `bp` input port of the `Prelookup` block, which inherits the following breakpoint attributes:

Breakpoint Attribute	Value
Minimum	`-Inf`
Maximum	`Inf`
Data type	`single`

Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision,

floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.

For other examples, see “Prelookup and Interpolation Blocks” in the Simulink documentation.

Characteristics

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

See Also

`Prelookup`

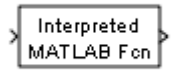
Introduced in R2006b

Interpreted MATLAB Function

Apply MATLAB function or expression to input

Library

User-Defined Functions



Description

The `Interpreted MATLAB Function` block applies the specified MATLAB function or expression to the input. The output of the function must match the output dimensions of the block.

Some valid expressions for this block are:

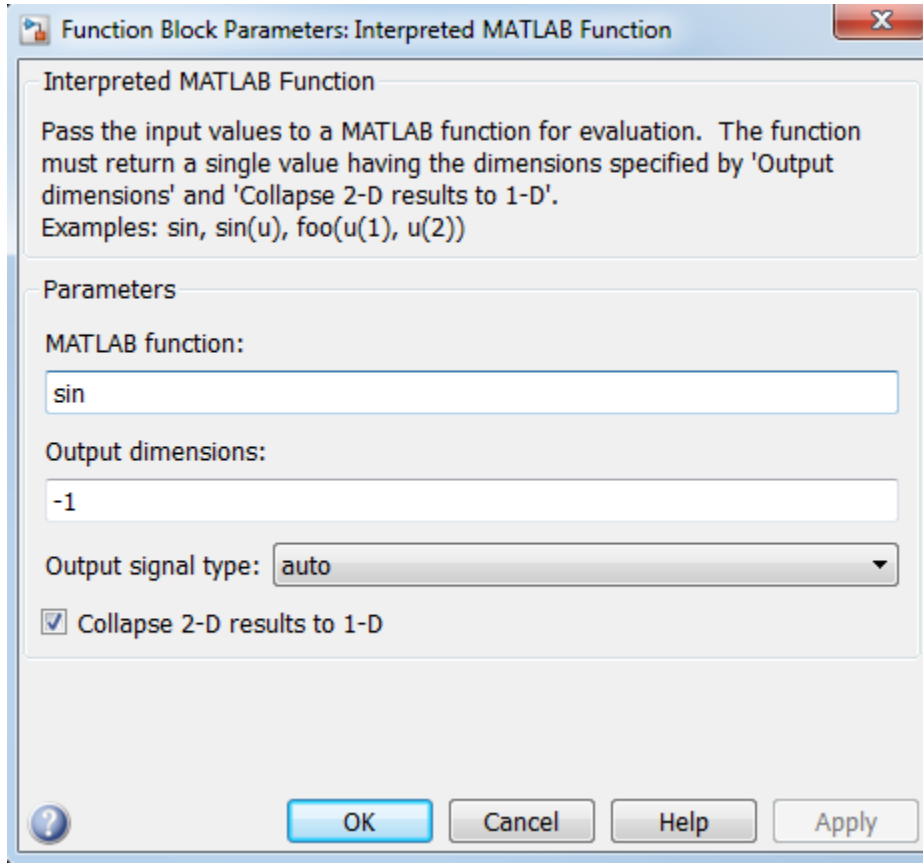
```
sin
atan2(u(1), u(2))
u(1)^u(2)
```

Note This block is slower than the Fcn block because it calls the MATLAB parser during each integration step. Consider using built-in blocks (such as the Fcn block or the Math Function block) instead. Alternatively, you can write the function as a MATLAB S-function or MEX-file S-function, then access it using the S-Function block.

Data Type Support

The `Interpreted MATLAB Function` block accepts one real or complex input of type `double` and generates real or complex output of type `double`, depending on the setting of the **Output signal type** parameter.

Parameters 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 block output signal, for example, `2` for a two-element vector. The output dimensions must match the dimensions of the value returned by the function or expression in the **MATLAB function** field.

Specify `-1` to inherit the dimensions from the output of the specified function or expression. To determine the output dimensions, Simulink runs the function or expression once before simulation starts.

Note: If you specify `-1` for this parameter and your function has persistent variables, then the variables might update before the simulation starts. If you need to use persistent variables, consider setting this parameter to a value other than `-1`.

Output signal type

Specify the output signal type of the block as `real`, `complex`, or `auto`. A value of `auto` sets the output type to be the same as the type of the input signal.

Collapse 2-D results to 1-D

Select this check box to output a 2-D array as a 1-D array containing the 2-D array's elements in column-major order.

Sample time

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

Characteristics

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

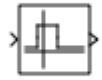
Introduced in R2011a

Interval Test

Determine if signal is in specified interval

Library

Logic and Bit Operations



Description

The Interval Test block outputs TRUE if the input is between the values specified by the **Lower limit** and **Upper limit** parameters. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the **Lower limit** or the **Upper limit** is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are selected in the dialog box.

Data Type Support

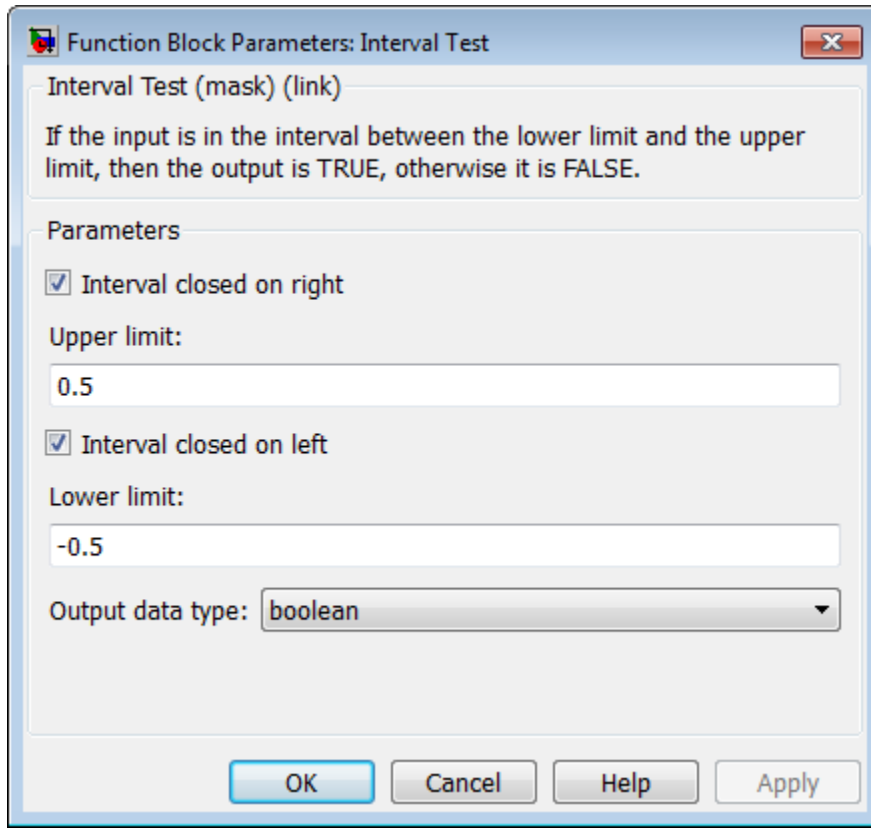
The Interval Test block accepts signals of the following data types:

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

  In this case, the **Upper limit** and **Lower limit** values must be of the same enumerated type.

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

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Interval Test Dynamic`

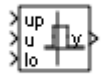
Introduced before R2006a

Interval Test Dynamic

Determine if signal is in specified interval

Library

Logic and Bit Operations



Description

The Interval Test Dynamic block outputs TRUE if the input is between the values of the external signals up and lo. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the signal up or the signal lo is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are selected in the dialog box.

Data Type Support

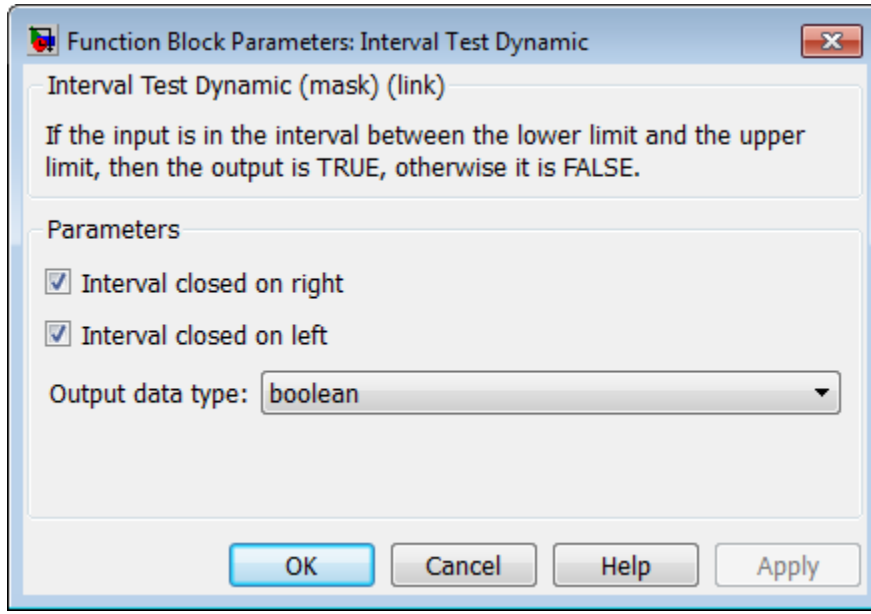
The Interval Test Dynamic block accepts signals of the following data types:

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

  In this case, all inputs must be of the same enumerated type.

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

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Direct Feedthrough	Yes

Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Interval Test`

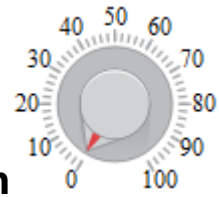
Introduced before R2006a

Knob

Set value on dial to tune parameters or variables

Library

Dashboard



Description

The Knob block enables you to control tunable parameters and variables in your model during simulation.

To control a tunable parameter or variable using the Knob block, double-click the Knob block to open the dialog box. Select a block in the model canvas. The tunable parameter or variable appears in the dialog box **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable to the block.

The tick range determines the continuous values generated for the tunable parameter or variable. You can modify the tick range by modifying the **Minimum**, **Maximum**, and **Tick Interval** values.

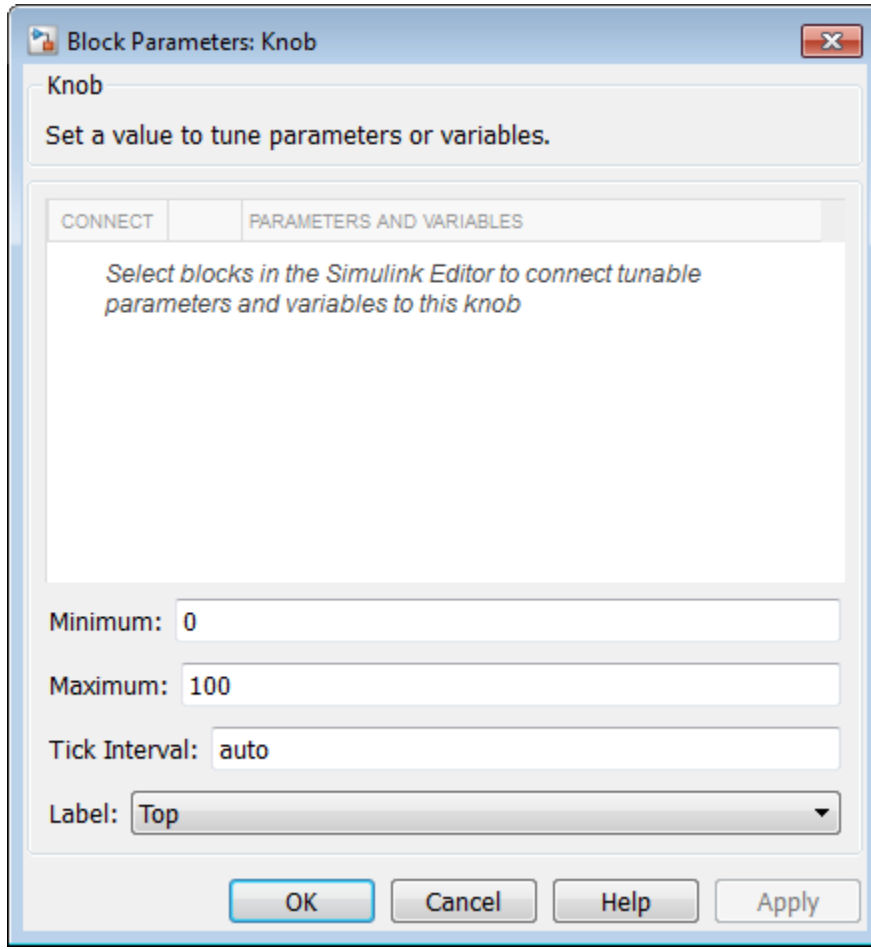
Limitations

The Knob block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the MDL format.	Save the model file to SLX format to be able to save connections and properties.

Limitation	Workaround
Parameters that index a variable array do not appear in the **Connection** table.	For example, a block parameter specified using the variable `engine(1)` will not appear in the table because the parameter uses an index of the variable `engine`, which is not a scalar variable. To make the parameter appear in the **Connection** table, change the block parameter field to a scalar variable, such as `engine_1`.

Parameters and Dialog Box



Connection

Select a block to connect and control a tunable parameter or variable.

To control a tunable parameter or variable, select a block in the model. The tunable parameter or variable appears in the **Connection** table. Select the option button next

to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable.

Settings

The table has a row for the tunable parameter or variable connected to the block. If there are no tunable parameters or variables selected in the model or the block is not connected to any tunable parameters or variables, then the table is empty.

Minimum

Minimum tick mark value.

Settings

Default: 0

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

Dependencies

The **Minimum** tick value must be less than the **Maximum** tick value.

Maximum

Maximum tick mark value.

Settings

Default: 100

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

Dependencies

The **Maximum** tick value must be greater than the **Minimum** tick value.

Tick Interval

Interval between major tick marks.

Settings

Default: `auto`

Specify this number as a finite, real, positive, integer, scalar value. Specify as `auto` for the block to adjust the tick interval automatically.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see "Tune and Visualize Your Model with Dashboard Blocks".

Introduced in R2015a

Lamp

Display color that reflects input value

Library

Dashboard

Description 

The `Lamp` block displays distinct input values from connected signals during simulation on a colored lamp.

To display data from a signal on the `Lamp` block, double-click the `Lamp` block to open the dialog box. Select a signal in the model canvas. The signal appears in the dialog box **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal to the block.

You can add or remove input values that are indicated by the `Lamp` block using the **States** table.

Limitations

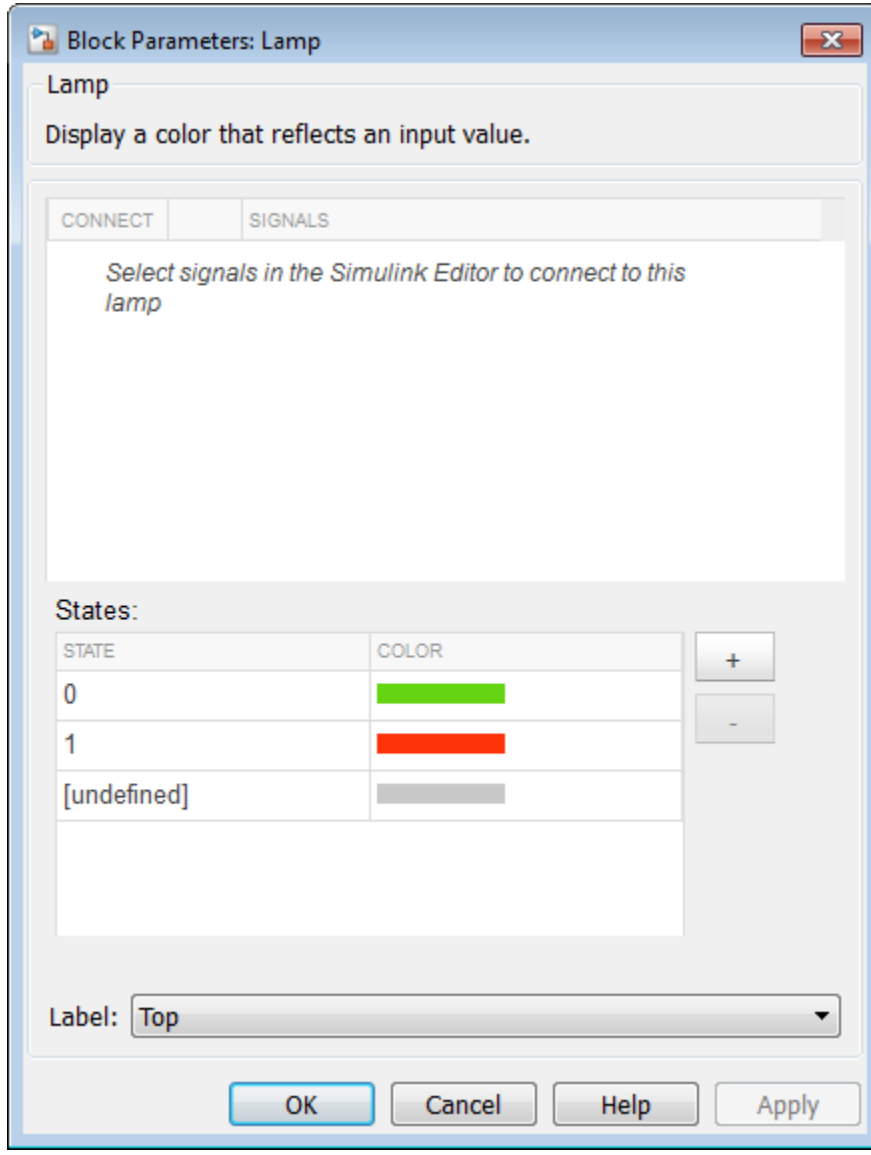
The `Lamp` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.

If you turn off streaming for a signal connected to a `Lamp` block, then the connection shows as broken. Signal data does not stream to the block. To view signal data again, double-click the `Lamp` block and reconnect the signal.

The External simulation mode is not supported for the `Lamp` block.

Parameters and Dialog Box



Connection

Select a signal to connect and display.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

Settings

The table has a row for the signal connected to the block. If there are no signals selected in the model or the block is not connected to any signals, then the table is empty.

States

State values and colors indicated on the lamp.

The states defined in the table determine the distinct input values indicated by the colored lamp, which are represented by the **Color** field. You can modify the input value states by editing the **State** and **Color** in the table.

To add a state, click the **+** button, enter the **State** value, and select a **Color**.

To remove a state, select the state in the table, and click the **-** button.

Settings

Default: `0`, `1`, and `undefined`

The `undefined` state is indicated when values that are not defined in the states table are input into the `Lamp` block.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top

Show the label at the top of the block.

Bottom

Show the label at the bottom of the block.

Hide

Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

Introduced in R2015a

Level-2 MATLAB S-Function

Use Level-2 MATLAB S-function in model

Library

User-Defined Functions

Description

This block allows you to use a Level-2 MATLAB S-function (see “Write Level-2 MATLAB S-Functions”) in a model. To do this, create an instance of this block in the model. Then enter the name of the Level-2 MATLAB S-function in the **S-function name** field of the block's parameter dialog box.

Note: Use the `S-Function` block to include a Level-1 MATLAB S-function in a block.

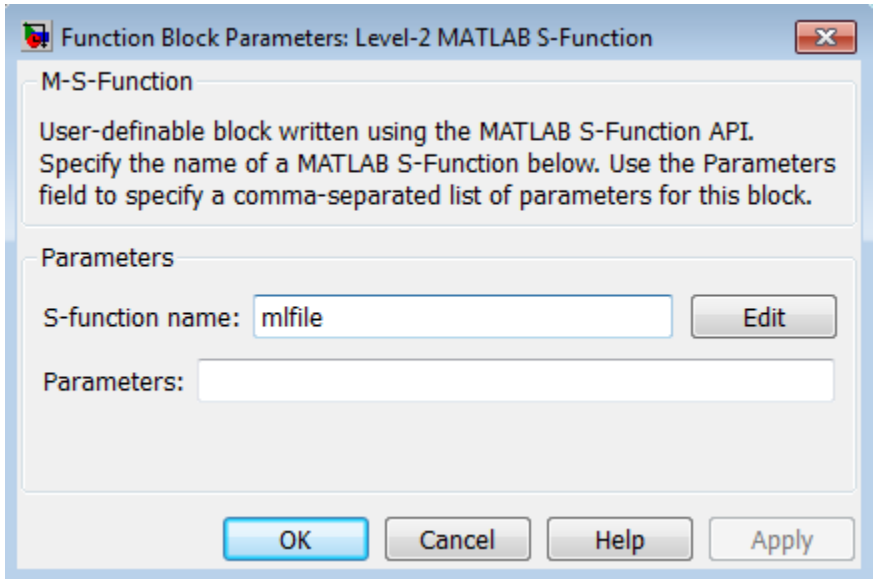
If the Level-2 MATLAB S-function defines any additional parameters, you can enter them in the **Parameters** field of the block's parameter dialog box. Enter the parameters as MATLAB expressions that evaluate to their values in the order defined by the MATLAB S-function. Use commas to separate each expression.

If a model includes a `Level-2 MATLAB S-Function` block, and an error occurs in the S-function, the `Level-2 MATLAB S-Function` block displays MATLAB stack trace information for the error in a dialog box. Click **OK** to close the dialog box.

Data Type Support

Depends on the MATLAB file that defines the behavior of a particular instance of this block.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Depends on the MATLAB S-function
Direct Feedthrough	Depends on the MATLAB S-function
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

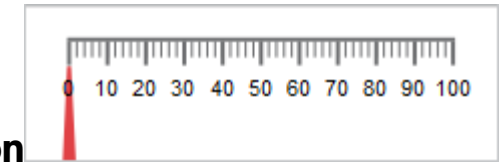
Introduced in R2010b

Linear Gauge

Display input value on linear scale

Library

Dashboard



Description

The `Linear Gauge` block displays connected signals during simulation on a linear scale.

To view data from a signal on the `Linear Gauge` block, double-click the `Linear Gauge` block to open the dialog box. Select a signal in the model canvas. The signal appears in the dialog box **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal to the block.

You can modify the tick range by modifying the **Minimum**, **Maximum**, and **Tick Interval** values.

You can also add scale colors that appear on the outside of the `Linear Gauge` block scale using the **Scale Colors** table.

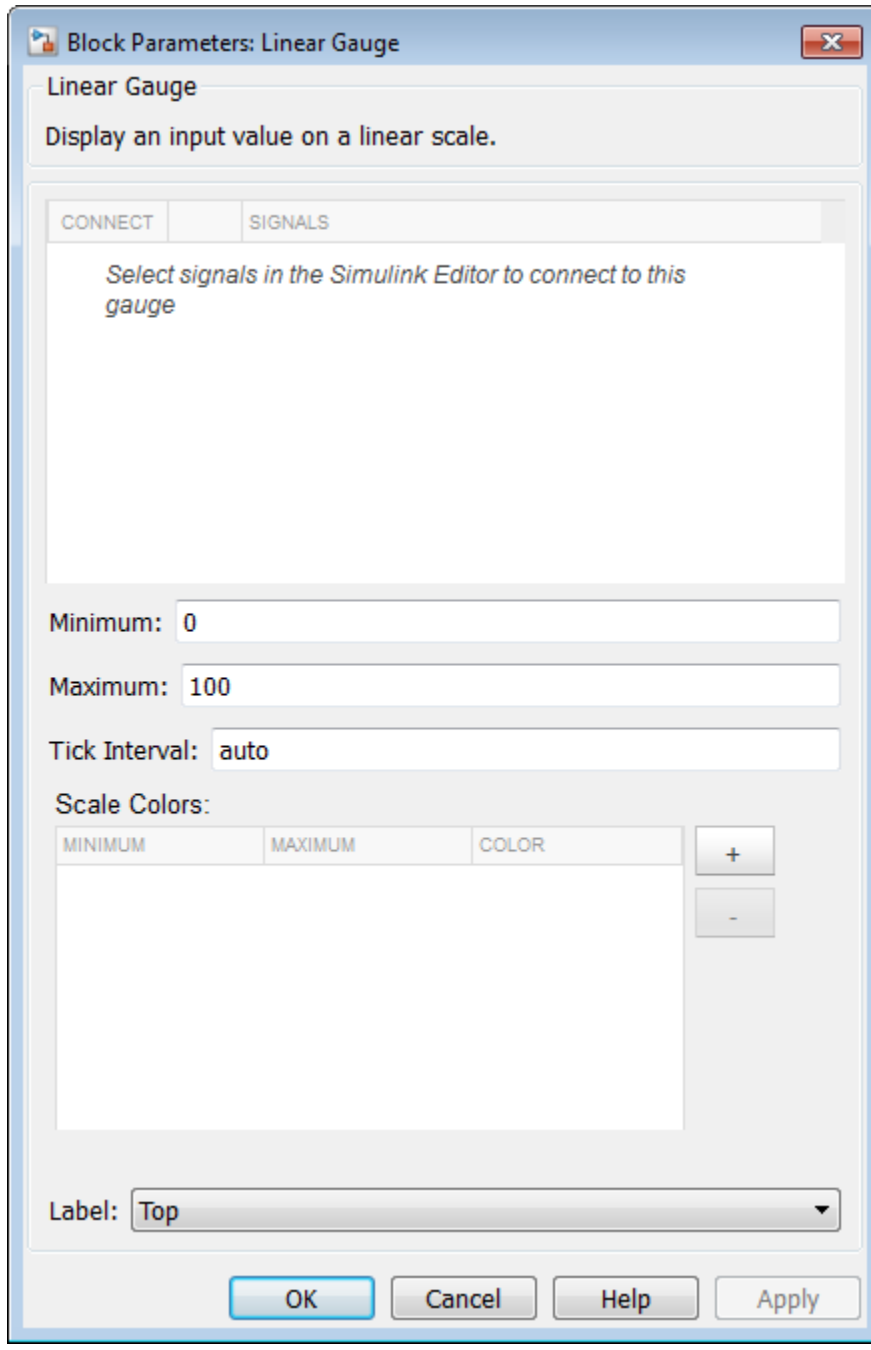
Limitations

The `Linear Gauge` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.

If you turn off streaming for a signal connected to any dashboard gauge, then the connection shows as broken. Signal data does not stream to the block. To view signal data again, double-click the gauge and reconnect the signal.

The External simulation mode is not supported for the `Linear Gauge` block.


Block Parameters: Linear Gauge
Linear Gauge
Display an input value on a linear scale.
CONNECT
SIGNALS
Select signals in the Simulink Editor to connect to this gauge
Minimum:
0
Maximum:
100
Tick Interval:
auto
Scale Colors:
MINIMUM
MAXIMUM
COLOR
+
-
Label:
Top
OK
Cancel
Help
Apply

Connection

Select a signal to connect and display.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

Settings

The table has a row for the signal connected to the block. If there are no signals selected in the model or the block is not connected to any signals, then the table is empty.

Minimum

Minimum tick mark value.

Settings

Default: 0

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

Dependencies

The **Minimum** tick value must be less than the **Maximum** tick value.

Maximum

Maximum tick mark value.

Settings

Default: 100

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

Dependencies

The **Maximum** tick value must be greater than the **Minimum** tick value.

Tick Interval

Interval between major tick marks.

Settings

Default: `auto`

Specify this number as a finite, real, positive, integer, scalar value. Specify as `auto` for the block to adjust the tick interval automatically.

Scale Colors

Specify ranges of color bands on the outside of the scale. Specify the minimum and maximum color range to display on the gauge.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

Introduced in R2015a

Logical Operator

Perform specified logical operation on input

Library

Logic and Bit Operations

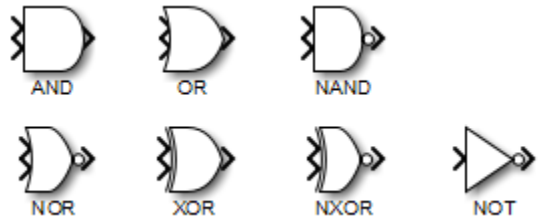
Description

The Logical Operator block performs the specified logical operation on its inputs. An input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero.

You select the Boolean operation connecting the inputs with the **Operator** parameter list. If you select `rectangular` as the **Icon shape** property, the block updates to display the name of the selected operator. The supported operations are given below.

Operation	Description
AND	TRUE if all inputs are TRUE
OR	TRUE if at least one input is TRUE
NAND	TRUE if at least one input is FALSE
NOR	TRUE when no inputs are TRUE
XOR	TRUE if an odd number of inputs are TRUE
NXOR	TRUE if an even number of inputs are TRUE
NOT	TRUE if the input is FALSE

If you select `distinctive` as the **Icon shape**, the block's appearance indicates its function. Simulink software displays a distinctive shape for the selected operator, conforming to the IEEE Standard Graphic Symbols for Logic Functions:



The number of input ports is specified with the **Number of input ports** parameter. The output type is specified with the **Output data type** parameter. An output value is 1 if TRUE and 0 if FALSE.

Note The output data type should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers, and any floating-point data type.

The size of the output depends on input vector size and the selected operator:

- If the block has more than one input, any nonscalar inputs must have the same dimensions. For example, if any input is a 2-by-2 array, all other nonscalar inputs must also be 2-by-2 arrays.

  Scalar inputs are expanded to have the same dimensions as the nonscalar inputs.

  If the block has more than one input, the output has the same dimensions as the inputs (after scalar expansion) and each output element is the result of applying the specified logical operation to the corresponding input elements. For example, if the specified operation is AND and the inputs are 2-by-2 arrays, the output is a 2-by-2 array whose top left element is the result of applying AND to the top left elements of the inputs, etc.
- For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector. The output is always a scalar.
- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the logical complements of the input vector elements.

When configured as a multi-input XOR gate, this block performs an addition- modulo-two operation as mandated by the IEEE Standard for Logic Elements.

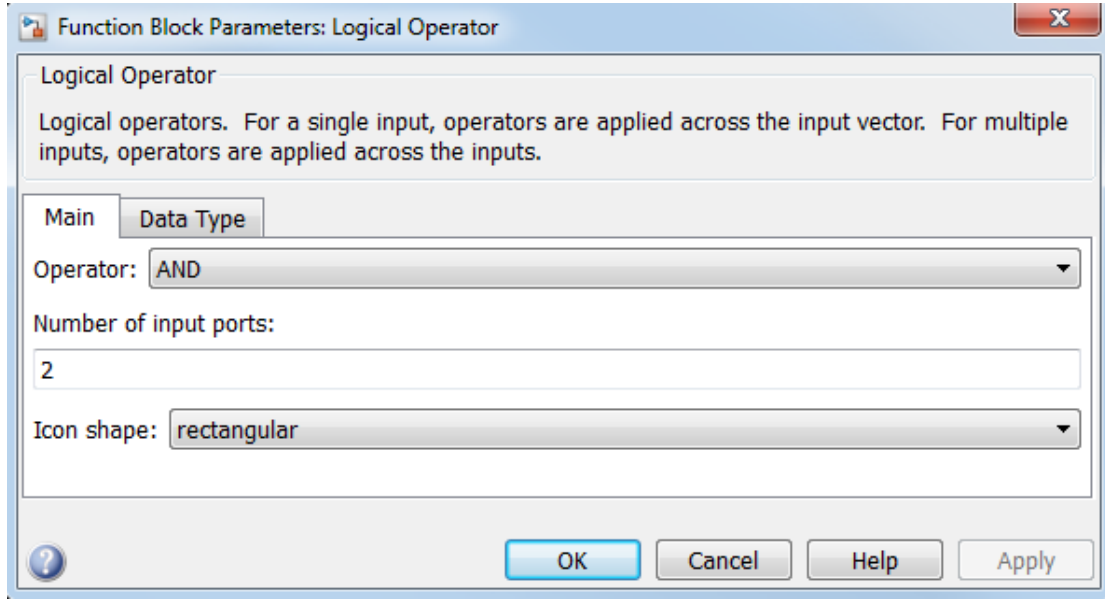
Data Type Support

The Logical Operator block accepts real signals of any numeric data type that Simulink supports, including fixed-point data types.

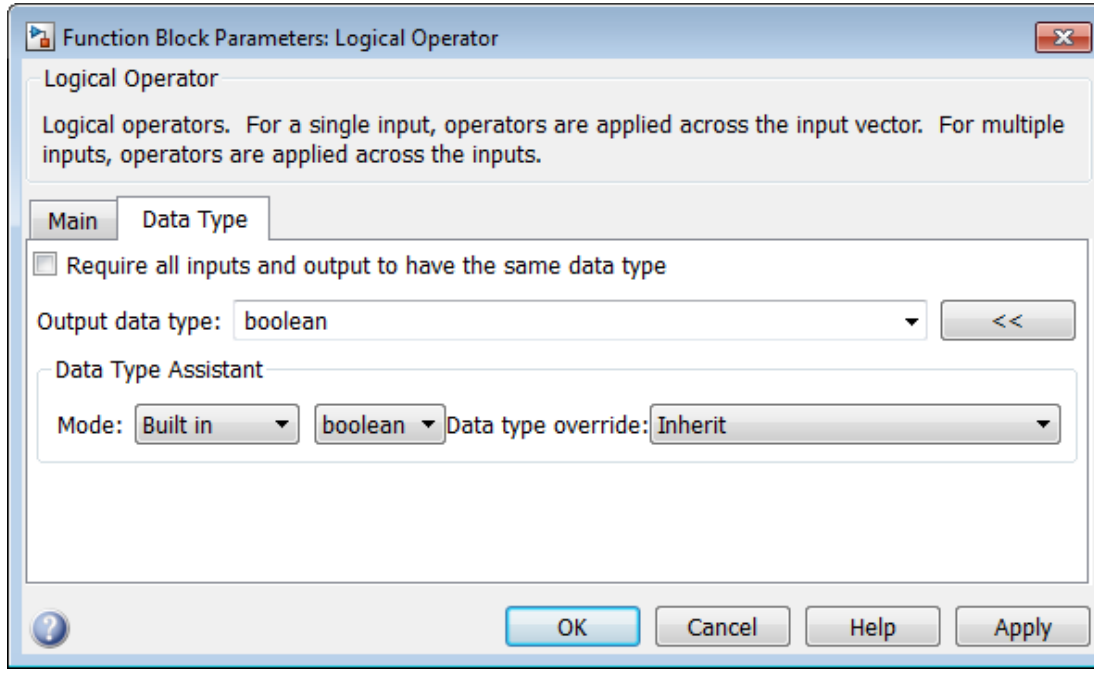
For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

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



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.
Main
Data Type
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
OK
Cancel
Help
Apply

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 “Control Signal Data Types”.

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 6-101 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 6-101 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 6-101 for the command-line information.

Sample time

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

Require all inputs and output to have the same data type

Require all inputs and the output to have the same data type.

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 6-101 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 6-101 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 6-101 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.

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.

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 6-101 for the command-line information.

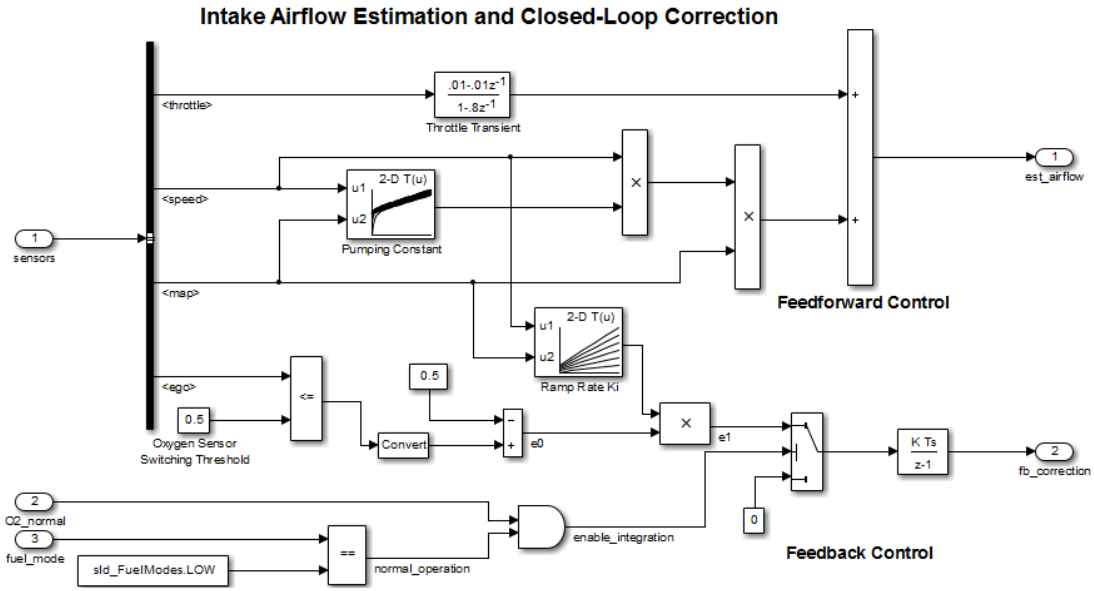
See Also

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:

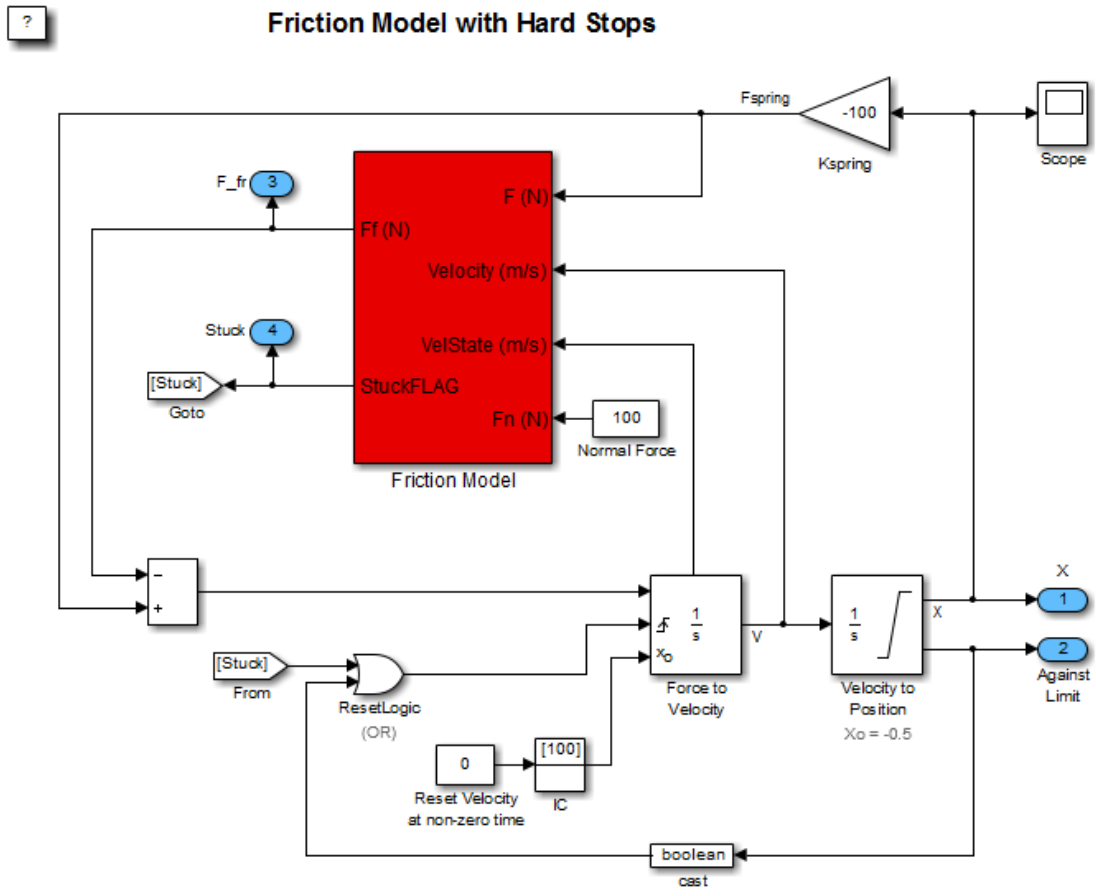


The output of the Logical Operator block (the `enable_integration` signal) feeds into the control port of a Switch block that activates feedback control.

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:

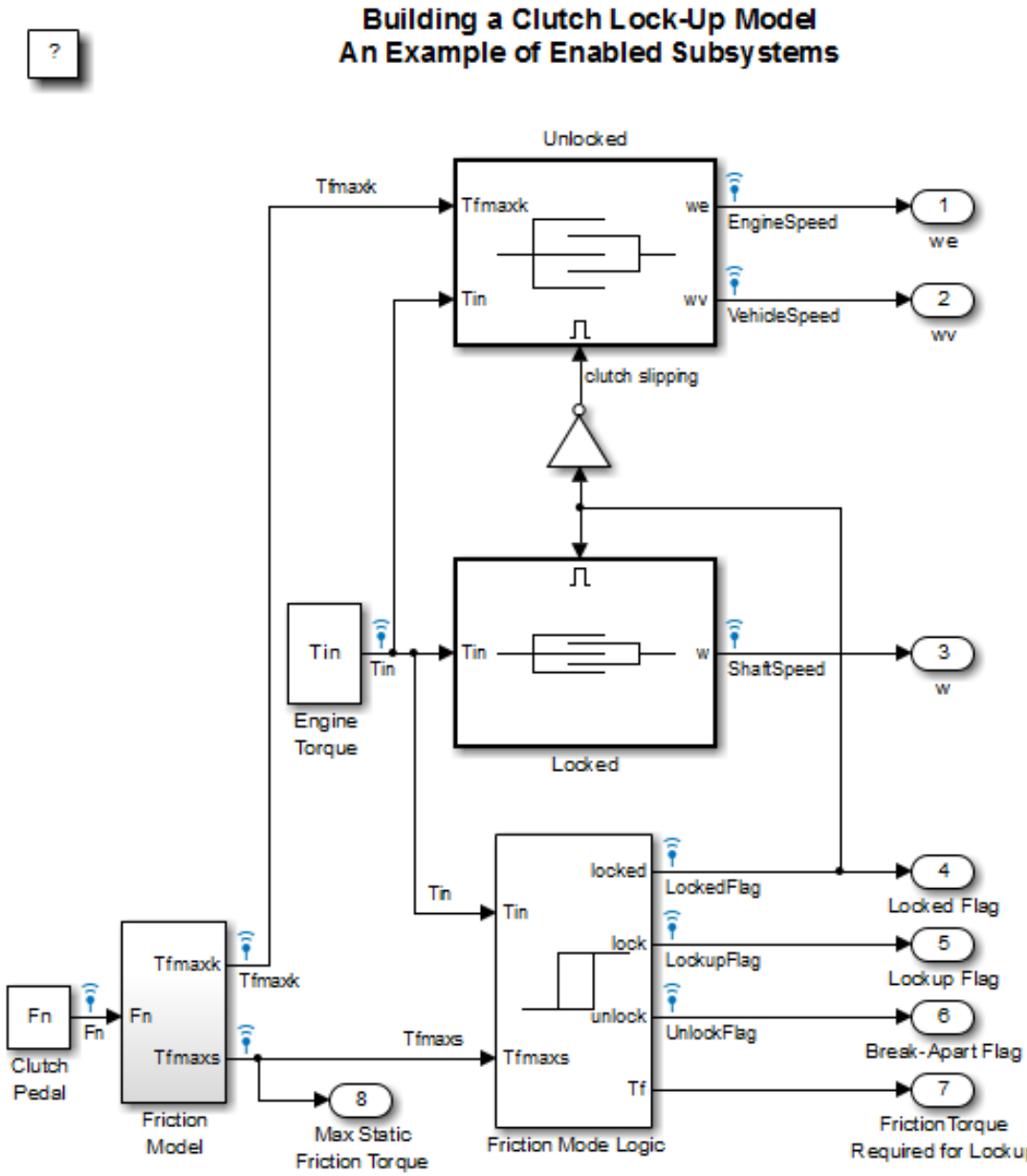


The output of the Logical Operator block feeds into the trigger port of an Integrator block to control whether velocity resets to the initial condition.

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
Tfmaxk
we
EngineSpeed
1
we
Tin
wv
VehicleSpeed
2
wv
clutch slipping
Tin
Tin
Engine
Torque
Tin
Tin
w
ShaftSpeed
3
w
Locked
Tin
locked
LockedFlag
4
Locked Flag
Tin
lock
LockupFlag
5
Lockup Flag
unlock
UnlockFlag
6
Break-Apart Flag
Tfmaxs
Tfmaxs
Tf
7
Friction Torque
Required for Lockup
Friction Mode Logic
Tfmaxk
Tfmaxk
Fn
Fn
Fn
Clutch
Pedal
Tfmaxs
Friction
Model
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

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

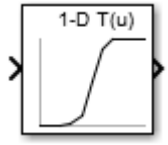
Introduced before R2006a

1-D Lookup Table

Approximate one-dimensional function

Library

Lookup Tables



Description

The `1-D Lookup Table` block is a one-dimensional version of the `n-D Lookup Table` block.

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

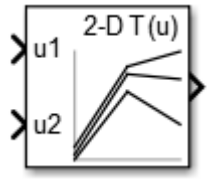
Introduced in R2011a

2-D Lookup Table

Approximate two-dimensional function

Library

Lookup Tables



Description

The `2-D Lookup Table` block is a two-dimensional version of the `n-D Lookup Table` block.

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

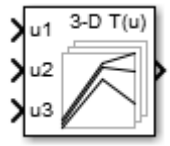
Introduced in R2011a

n-D Lookup Table

Approximate N-dimensional function

Library

Lookup Tables



Description

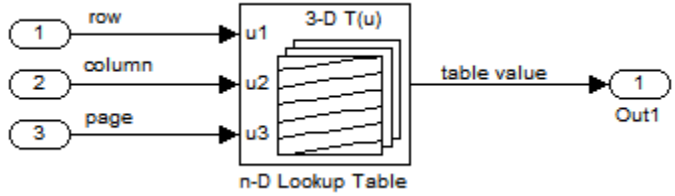
Supported Block Operations

The n-D Lookup Table block evaluates a sampled representation of a function in N variables

$$y = F(x_1, x_2, x_3, ..., x_N)$$

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

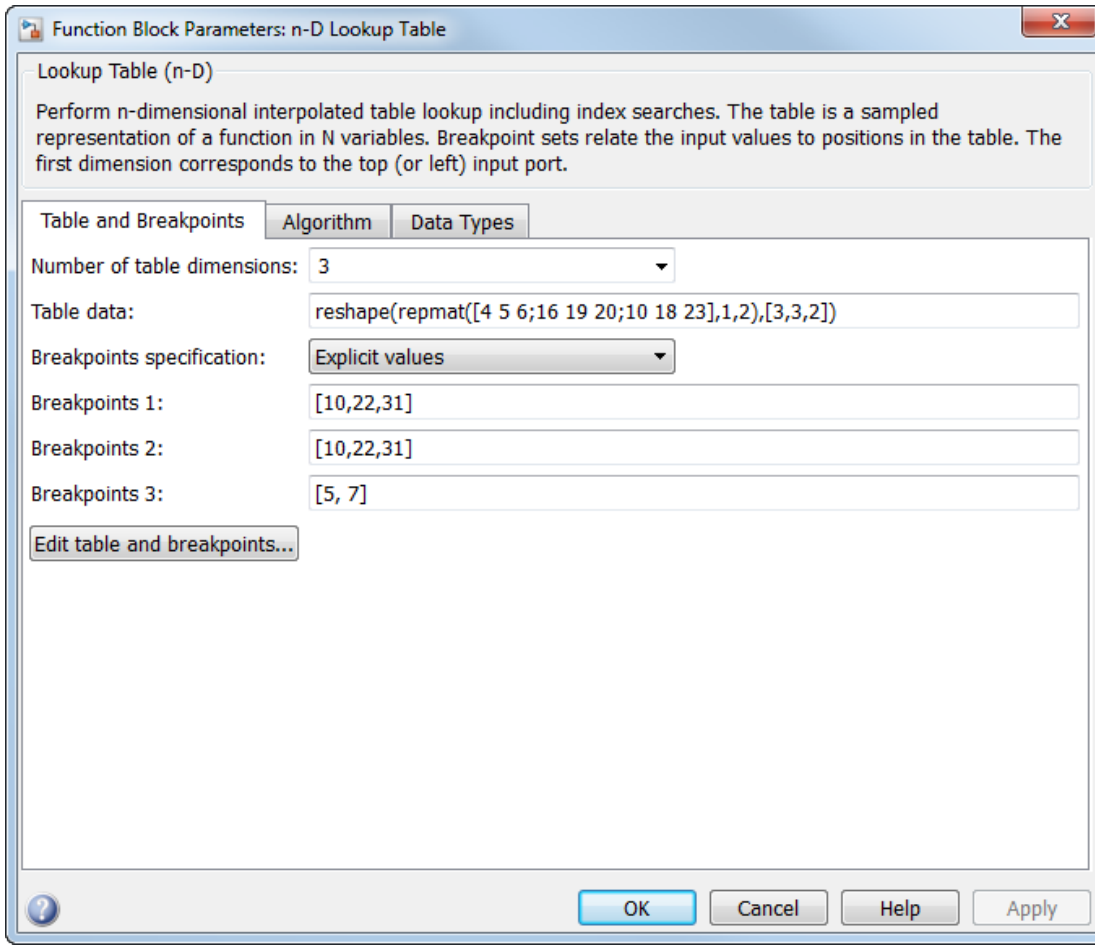


Table and Breakpoints tab

Number of table dimensions

Enter the number of dimensions of the lookup table by specifying an integer from `1` to `30`. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

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 specification

Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- To explicitly specify breakpoint data, set this parameter to `Explicit values` and enter breakpoint data in the text box next to the **Breakpoints** parameters.
- To specify parameters that generate evenly spaced breakpoints, set this parameter to `Even spacing` and enter values for the **First point** and **Spacing** parameters for each dimension of breakpoint data. The block calculates the number of points to generate from the table data.

First point

Specify the first point in your evenly spaced breakpoint data. This parameter is available when **Breakpoints specification** is set to `Even spacing`.

Spacing

Specify the spacing between points in your evenly-spaced breakpoint data. This parameter is available when **Breakpoints specification** is set to `Even spacing`.

Breakpoints

Specify the breakpoint data explicitly or as evenly-spaced breakpoints, based on the value of the **Breakpoints specification** parameter.

- If you set **Breakpoints specification** to Even spacing, enter the parameters **First point** and **Spacing** in each **Breakpoints** row to generate evenly-spaced

breakpoints in the respective dimension. Your table data determines the number of evenly spaced points.

- If you set **Breakpoints specification** to `Explicit values`, enter the breakpoint set that corresponds to each dimension of table data in each **Breakpoints** row. For each dimension, specify breakpoints as a 1-by-n or n-by-1 vector whose values are strictly monotonically increasing.

Edit table and breakpoints

Click this button to open the Lookup Table Editor. For more information, see "Edit Lookup Tables" in the Simulink documentation.

Sample time

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

Algorithm tab

Interpolation method

Select `Flat`, `Nearest`, `Linear`, or `Cubic spline`. See "Interpolation Methods" in the Simulink documentation for more information.

If you select `Cubic spline`, the block supports only scalar signals. The other interpolation methods support nonscalar signals.

Extrapolation method

Select `Clip`, `Linear`, or `Cubic spline`. See "Extrapolation Methods" in the Simulink documentation for more information.

To select `Cubic spline` for **Extrapolation method**, you must also select `Cubic spline` for **Interpolation method**.

Use last table value for inputs at or above last breakpoint

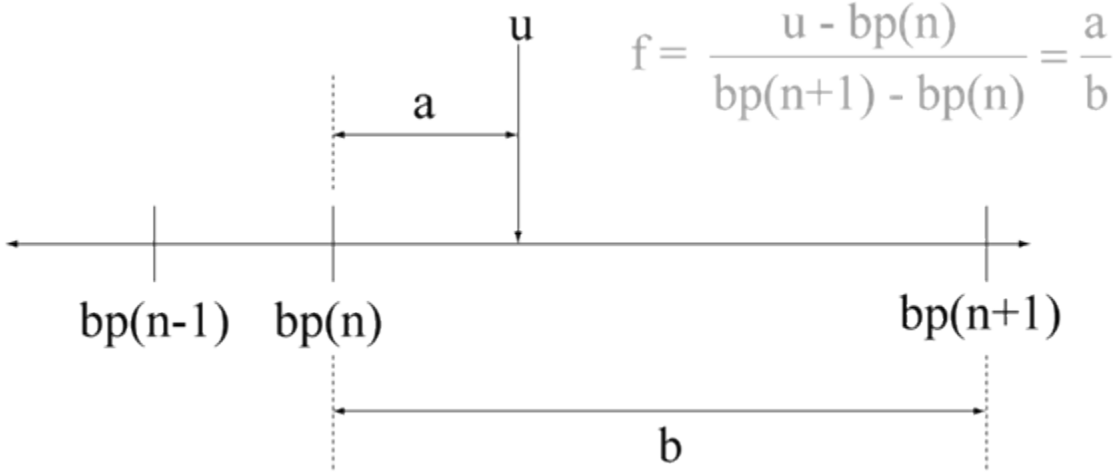
Using this check box, specify the indexing convention that the block uses to address the last element of a breakpoint set and its corresponding table value. This check box is relevant if the input is larger than the last element of the breakpoint data. This parameter is visible only when:

- **Interpolation method** is `Linear`.

- **Extrapolation method** is `Clip`.

Check Box	Block Uses Index Of The...	Interval Fraction
Selected	Last element of breakpoint data on the **Table and Breakpoints** tab	0
Cleared	Next-to-last element of breakpoint data on the **Table and Breakpoints** tab	1

Given an input u within range of a breakpoint set bp, the interval fraction f, in the range $0 \le f \le 1$, is computed as shown below.



Suppose the breakpoint set is `[1 4 5]` and input u is `5.5`. If you select this check box, the index is that of the last element (`5`) and the interval fraction is 0. If you clear this checkbox, the index is that of the next-to-last element (`4`) and the interval fraction is 1.

Diagnostic for out-of-range input

Specify whether to produce a warning or error when the input is out of range. Options include:

- `None` — 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 Diagnostic 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 “Run Model Checks” in the Simulink documentation.

Index search method

Select `Evenly spaced points`, `Linear search`, or `Binary search`. Each search method has speed advantages in different circumstances:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting `Evenly spaced points` to calculate table indices.

  This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input signals do not vary much between time steps, selecting `Linear search` with **Begin index search using previous index result** produces the best performance.
  - If input signals jump more than one or two table intervals per time step, selecting `Binary search` produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

Note: The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is `Evenly spaced points`.

Begin index search using previous index result

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

Use one input port for all input data

Select this check box to use only one input port that expects a signal that is N elements wide for an N-dimensional table. This option is useful for removing line clutter on a block diagram with many lookup tables.

Note: When you select this check box, one input port with the label `u` appears on the block.

Support tunable table size in code generation

Select this check box to enable tunable table size in the generated code. This option enables you to change the size and values of the lookup table and breakpoint data in the generated code without regenerating or recompiling the code.

If you set **Interpolation method** to `Cubic spline`, this check box is not available.

Maximum indices for each dimension

Specify the maximum index values for each table dimension using zero-based indexing. You can specify a scalar or vector of positive integer values using the following data types:

- Built-in floating-point types: `double` and `single`
- Built-in integer types: `int8`, `int16`, `int32`, `uint8`, `uint16`, and `uint32`

Here are some examples of valid specifications:

- `[4 6]` for a 5-by-7 table
- `[int8(2) int16(5) int32(9)]` for a 3-by-6-by-10 table
- A `Simulink.Parameter` whose value on generating code is one less than the dimensions of the table data. For more information, see "Tunable Table Size in the Generated Code" on page 1-1056.

This parameter is available when you select **Support tunable table size in code generation**. On tuning this parameter in the generated code, provide the new table data and breakpoints along with the tuned parameter value.

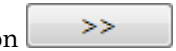
Data Types tab

Note: The dialog box can expand to show additional data type options. Up to 30 breakpoint data type specifications can appear.

Table data > Data Type

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

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

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the table data type.

Tip Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two `n-D Lookup Table` blocks with different output data types
- Sharing of custom storage table data in the generated code for blocks with different output data types

Table data > Minimum

Specify the minimum value for table data. The default value is `[]` (unspecified).

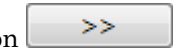
Table data > Maximum

Specify the maximum value for table data. The default value is `[]` (unspecified).

Breakpoints > Data Type

Specify the data type for a set of breakpoint data. You can set it to:

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

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the breakpoint data type.

See "Specify Data Types Using Data Type Assistant" in the Simulink documentation for more information.

Tip Specify a breakpoint data type different from the corresponding input data type for these cases:

- Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal
- Sharing of prescaled breakpoint data between two `n-D Lookup Table` blocks with different input data types

- Sharing of custom storage breakpoint data in the generated code for blocks with different input data types

---

Breakpoints > Minimum

Specify the minimum value that a set of breakpoint data can have. The default value is `[ ]` (unspecified).

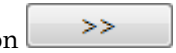
Breakpoints > Maximum

Specify the maximum value that a set of breakpoint data can have. The default value is `[ ]` (unspecified).

Fraction > Data Type

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

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

Click the **Show data type assistant** button 

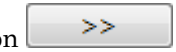
to display the **Data Type Assistant**, which helps you set the fraction data type.

See "Specify Data Types Using Data Type Assistant" in the Simulink documentation for more information.

Intermediate results > Data Type

Specify the intermediate results data type. You can set it to:

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

Click the **Show data type assistant** button 

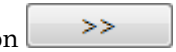
to display the **Data Type Assistant**, which helps you set the intermediate results data type.

Tip Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.

Output > Data Type

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

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

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

See "Control Signal Data Types" for more information.

Output > Minimum

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

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

Internal rule priority

Specify the internal rule for intermediate calculations. Select `Speed` for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

Require all inputs to have the same data type

Select to require all inputs to have the same data type.

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

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Fixed-Point Tool" 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`

Parameter	Value
Breakpoints 1	`p.BP1`
Breakpoints 2	`p.BP2`
Support tunable table size in code generation	`on`
Maximum indices for each dimension	`p.MaxIdx`

The generated *`model`*`_types.h` header file contains a type definition that looks something like this:

```
typedef struct {
  uint32_T MaxIdx[2];
  real_T BP1[3];
  real_T BP2[3];
  real_T Table[9];
} slLookupTable;
```

The generated *`model`*`.c` file contains code that looks something like this:

```
/* Exported block parameters */
slLookupTable p = {
  { 2U, 2U },

  { 1.0, 2.0, 3.0 },

  { 1.0, 4.0, 16.0 },

  { 4.0, 16.0, 10.0, 5.0, 19.0, 18.0, 6.0, 20.0, 23.0 }
} ;

/* More code */

/* Model output function */
static void ex_lut_nd_tunable_table_output(int_T tid)
{
  /* Lookup_n-D: '<Root>/n-D Lookup Table' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
  Y = look2_binlcpw(U1, U2, p.BP1, p.BP2, p.Table, ...
p.MaxIdx, p.MaxIdx[0] + 1U);
```

```
  /* Outport: '<Root>/Out1' */
  ex_lut_nd_tunable_table_Y.Out1 = Y;

  /* tid is required for a uniform function interface.
   * Argument tid is not used in the function. */
  UNUSED_PARAMETER(tid);
}
```

The highlighted line of code specifies a tunable table size for the lookup table. You can change the size and values of the lookup table and breakpoint data without regenerating or recompiling the code.

Characteristics

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

See Also

`Prelookup`, `Interpolation Using Prelookup`

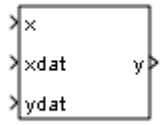
Introduced in R2011a

Lookup Table Dynamic

Approximate one-dimensional function using dynamic table

Library

Lookup Tables



Description

How This Block Differs from Other Lookup Table Blocks

The Lookup Table Dynamic block computes an approximation of a function `y = f(x)` using `xdat` and `ydat` vectors. The lookup method can use interpolation, extrapolation, or the original values of the input.

Using the Lookup Table Dynamic block, you can change the table data without stopping the simulation. For example, you can incorporate new table data if the physical system you are simulating changes.

Inputs for Breakpoint and Table Data

The `xdat` vector is the breakpoint data, which must be *strictly monotonically increasing*. The value of the next element in the vector must be greater than the value of the preceding element after conversion to a fixed-point data type. Due to quantization, `xdat` can be strictly monotonic for a floating-point data type, but not after conversion to a fixed-point data type.

The `ydat` vector is the table data, which is an evaluation of the function at the breakpoint values.

Note: The inputs to `xdat` and `ydat` cannot be scalar (one-element array) values. If you provide a scalar value to either of these inputs, you see an error upon simulation. Provide a 1-by-n vector to both the `xdat` and `ydat` inputs.

Lookup Table Definition

You define the lookup table by feeding `xdat` and `ydat` as 1-by-n vectors to the block. To reduce ROM usage in the generated code for this block, you can use different data types for `xdat` and `ydat`. However, these restrictions apply:

- The `xdat` breakpoint data and the `x` input vector must have the same sign, bias, and fractional slope. Also, the precision and range for `x` must be greater than or equal to the precision and range for `xdat`.
- The `ydat` table data and the `y` output vector must have the same sign, bias, and fractional slope.

Tip Breakpoints with even spacing can make Simulink Coder generated code division-free. For more information, see `fixpt_evenspace_cleanup` in the Simulink documentation and "Identify questionable fixed-point operations" 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

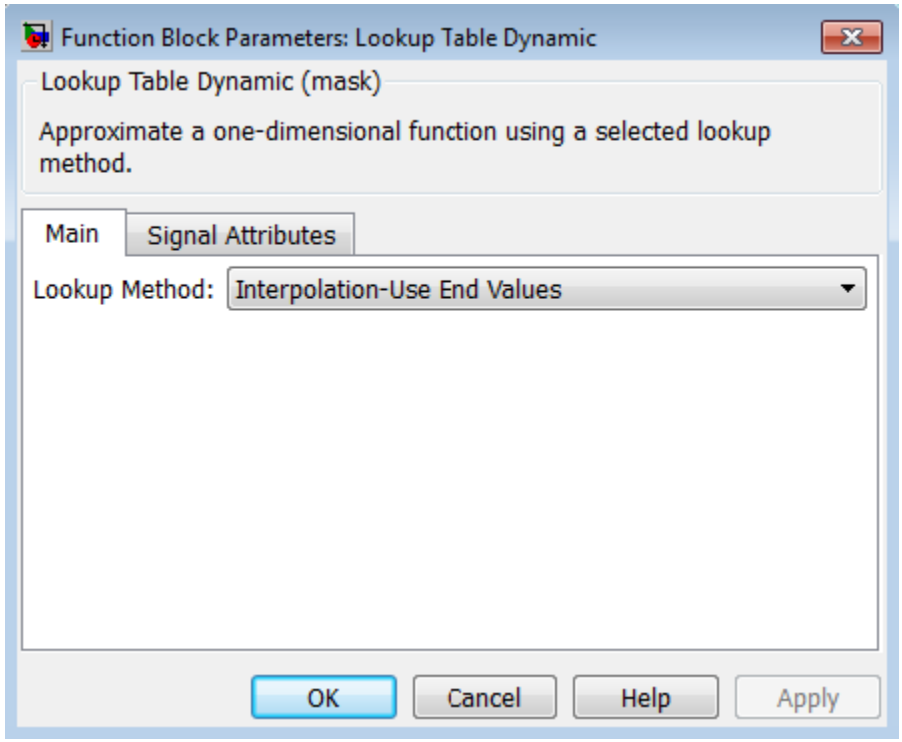
The Lookup Table Dynamic block accepts signals of the following data types:

- 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



- “Main tab” on page 1-1062
- “Signal Attributes tab” on page 1-1063

Main tab

Lookup Method

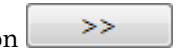
Specify the lookup method. For details, see “How the Block Generates Output” on page 1-1060.

Signal Attributes tab

Output data type

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

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

Click the **Show data type assistant** button 

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

See "Control Signal Data Types" for more information.

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

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

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"

For an example of...	See...
Block output for different lookup methods	“Example Output for Lookup Methods”

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`n-D Lookup Table`

Introduced before R2006a

Magnitude-Angle to Complex

Convert magnitude and/or a phase angle signal to complex signal

Library

Math Operations

Description

Supported Operations

The Magnitude-Angle to Complex block converts magnitude and phase angle inputs to a complex output. The angle input must be in radians.

The block supports the following combinations of input dimensions when there are two block inputs:

- Two inputs of equal dimensions
- One scalar input and the other an n-dimensional array

If the block input is an array, the output is an array of complex signals. The elements of a magnitude input vector map to the magnitudes of the corresponding complex output elements. Similarly, the elements of an angle input vector map to the angles of the corresponding complex output elements. If one input is a scalar, it maps to the corresponding component (magnitude or angle) of all the complex output signals.

Effect of Out-of-Range Input on CORDIC Approximations

If you use the CORDIC approximation method (see “Definitions” on page 1-1066), 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 one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

Data Type Support

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

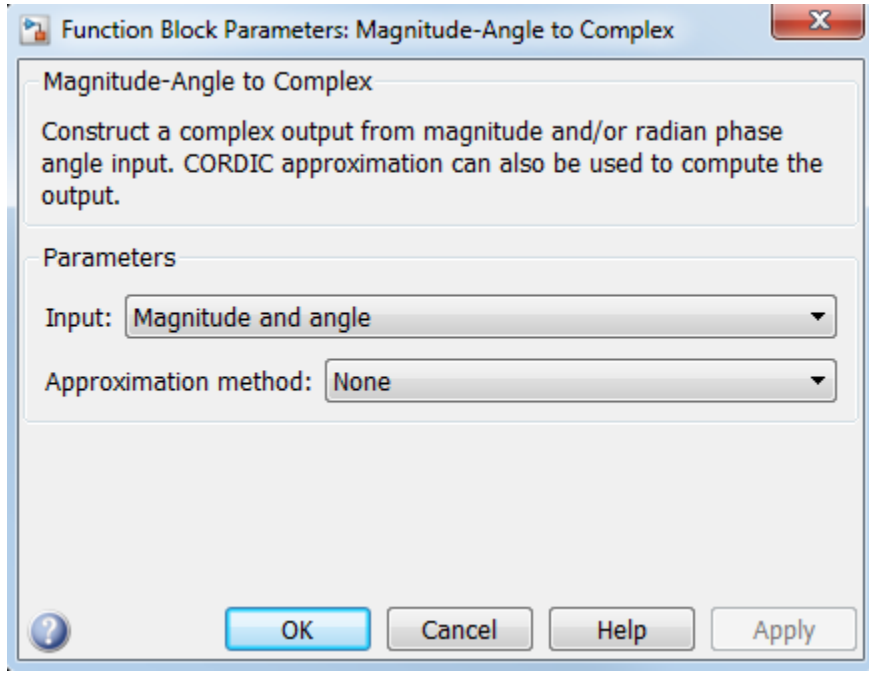
- Floating point
- Fixed point (only when **Approximation method** is `CORDIC`)

The following restrictions also apply:

- If one input uses a floating-point type, the other input must use the same data type. For example, both signals must be `double` or `single`.
- If one input uses a fixed-point type, the other input must also use a fixed-point type.

Parameters and Dialog Box

The dialog box for this block appears as follows:



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

Input	What to Specify
`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

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

Characteristics

Data Types	Double | Single | Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

References

[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

See Also

`Complex to Magnitude-Angle`

Introduced before R2006a

Manual Switch

Switch between two inputs

Library

Signal Routing

Description

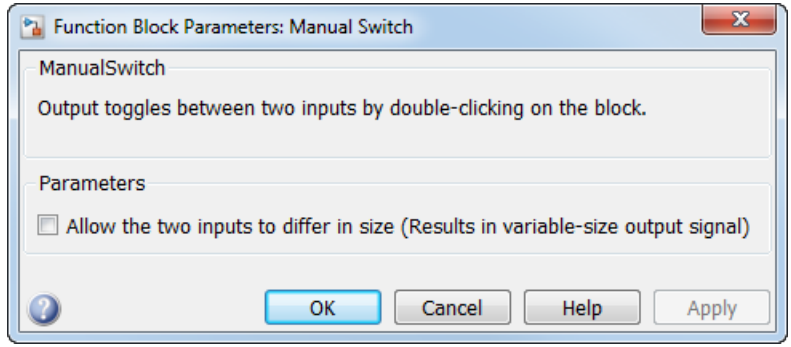
The Manual Switch block is a toggle switch that selects one of its two inputs to pass through to the output. To toggle between inputs, double-click the block. The block propagates the selected input to the output, while the block discards the unselected input. You can interactively control the signal flow by setting the switch before you start the simulation or by changing the switch while the simulation is executing. The Manual Switch block retains its current state when you save the model.

Data Type Support

The Manual Switch block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters 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**.



- "Allow the two inputs to differ in size" on page 1-1073
- "Sample time" on page 1-298

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

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

Examples

The following models show how to use the Manual Switch block:

- `sldemo_auto_climatecontrol`
- `sldemo_fuelsys`
- `sldemo_doublebounce`

Characteristics

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

Introduced before R2006a

Math Function

Perform mathematical function

Library

Math Operations

Description

The Math Function block performs numerous common mathematical functions.

Tip To perform square root calculations, use the `Sqrt` block.

You can select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
`exp`	Exponential	e^u	`exp`
`log`	Natural logarithm	$\ln u$	`log`
`10^u`	Power of base 10	10^u	`10.^u` (see `power`)
`log10`	Common (base 10) logarithm	$\log u$	`log10`
`magnitude^2`	Complex modulus	$\|u\|^2$	`(abs(u)).^2` (see `abs` and `power`)
`square`	Power 2	u^2	`u.^2` (see `power`)

Function	Description	Mathematical Expression	MATLAB Equivalent
pow	Power	u^v	power
conj	Complex conjugate	$\bar{u}$	conj
reciprocal	Reciprocal	$1/u$	1./u (see rdivide)
hypot	Square root of sum squares	$(u^2+v^2)^{0.5}$	hypot
rem	Remainder after division	—	rem
mod	Modulus after division	—	mod
transpose	Transpose	u^T	u.' (see "Array vs. Matrix Operations")
hermitian	Complex conjugate transpose	u^H	u' (see "Array vs. Matrix Operations")

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	—
hypot	yes, on two inputs	yes, on two inputs (two vectors or two matrices	—

Function	Scalar Operations	Element-Wise Vector and Matrix Operations	Vector and Matrix Operations
		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	—	—	—

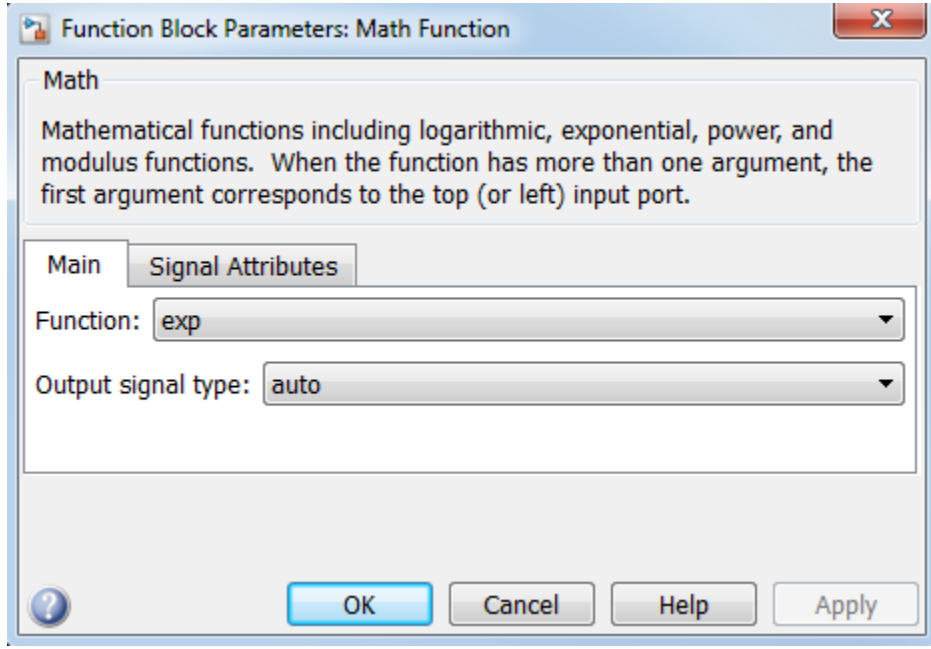
Function	single	double	boolean	built-in integer	fixed point
`magnitude^2`	yes	yes	—	yes	yes
`square`	yes	yes	—	yes	yes
`pow`	yes	yes	—	—	—
`conj`	yes	yes	—	yes	yes
`reciprocal`	yes	yes	—	yes	yes
`hypot`	yes	yes	—	—	—
`rem`	yes	yes	—	yes	—
`mod`	yes	yes	—	yes	—
`transpose`	yes	yes	yes	yes	yes
`hermitian`	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

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`.

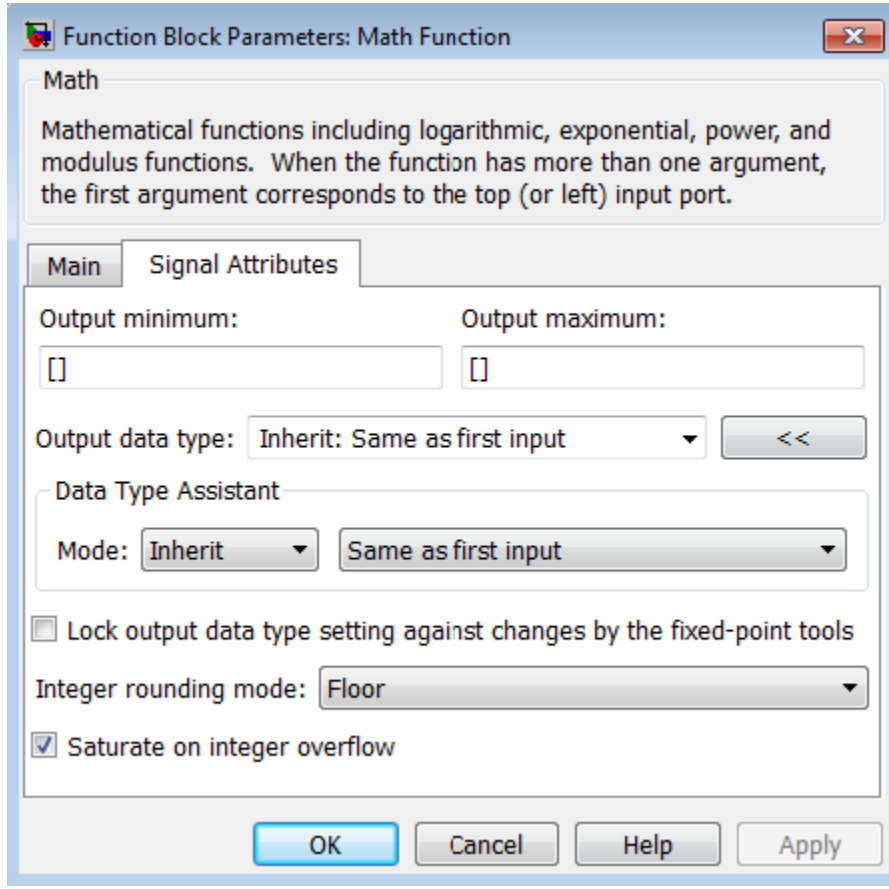
Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
`exp`, `log`, `10u`, `log10`, `square`, `pow`, `reciprocal`, `conjugate`, `transpose`, `hermitian`	`real`	`real`	`real`	`complex`
	`complex`	`complex`	`error`	`complex`

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
`magnitude squared`	`real`	`real`	`real`	`complex`
	`complex`	`real`	`real`	`complex`
`hypot`, `rem`, `mod`	`real`	`real`	`real`	`complex`
	`complex`	`error`	`error`	`error`

Sample time

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

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



Note Some parameters on this pane are available only when the function you select in the **Function** parameter supports fixed-point data types.

Output minimum

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

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

Output maximum

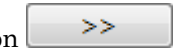
Specify the maximum value that the block can output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

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

Output data type

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

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

Click the **Show data type assistant** button 

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

See "Control Signal Data Types" in Simulink User's Guide for more information.

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

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

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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
			Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

Characteristics

Direct Feedthrough	Yes

Sample Time	Inherited from driving block
Scalar Expansion	Yes, of the input when the function requires two inputs
Dimensionalized	Yes
Multidimensionalized	Yes, for all functions except `hermitian` and `transpose`
Zero-Crossing Detection	No

See Also

`Sqrt`, `Trigonometric Function`

Introduced before R2006a

MATLAB Function

Include MATLAB code in models that generate embeddable C code

Library

User-Defined Functions

Description

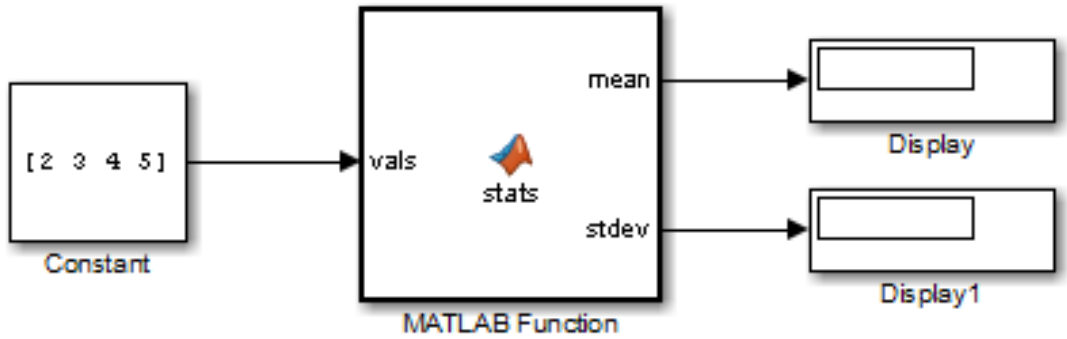
With a `MATLAB Function` block, you can write a MATLAB function for use in a Simulink model. The MATLAB function you create executes for simulation and generates code for a Simulink Coder target. If you are new to the Simulink and MATLAB products, see “What Is a MATLAB Function Block?” and “Create Model That Uses MATLAB Function Block” for an overview.

Double-clicking the MATLAB Function block opens its editor, where you write the MATLAB function, as in this example:

```
1   function [mean,stdev] = stats(vals)
2   % #codegen
3
4   % calculates a statistical mean and a standard
5   % deviation for the values in vals.
6
7   len = length(vals);
8   mean = avg(vals,len);
9   stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
10  coder.extrinsic('plot');
11  plot(vals,'-+');
12
13  function mean = avg(array,size)
14  mean = sum(array)/size;
```

To learn more about this editor, see “MATLAB Function Block Editor”.

You specify input and output data to the `MATLAB Function` block in the function header as arguments and return values. The argument and return values of the preceding example function correspond to the inputs and outputs of the block in the model:



You can also define data, input triggers, and function call outputs using the Ports and Data Manager, which you access from the MATLAB Function Block Editor by selecting **Edit Data**. See "Ports and Data Manager".

The `MATLAB Function` block generates efficient embeddable code based on an analysis that determines the size, class, and complexity of each variable. This analysis imposes the following restrictions:

- The first assignment to a variable defines its, size, class, and complexity.

  See "Best Practices for Defining Variables for C/C++ Code Generation".
- You cannot reassign variable properties after the initial assignment except when using variable-size data or reusing variables in the code for different purposes.

  See "Reassignment of Variable Properties".

In addition to language restrictions, the `MATLAB Function` block supports a subset of the functions available in MATLAB. A list of supported functions is given in "Functions and Objects Supported for C and C++ Code Generation — Alphabetical List". These functions include functions in common categories, such as:

- Arithmetic operators like `plus`, `minus`, and `power`. For more information, see "Array vs. Matrix Operations".
- Matrix operations like `size`, and `length`
- Advanced matrix operations like `lu`, `inv`, `svd`, and `chol`
- Trigonometric functions like `sin`, `cos`, `sinh`, and `cosh`

See "Functions and Objects Supported for C and C++ Code Generation — Category List" for a complete list of function categories.

Note Although the code for this block attempts to produce exactly the same results as MATLAB, differences might occur due to rounding errors. These numerical differences, which might be a few `eps` initially, can magnify after repeated operations. Reliance on the behavior of `nan` is not recommended. Different C compilers can yield different results for the same computation.

Note: In the `MATLAB Function` block, the `%#codegen` directive is included to emphasize that the block's MATLAB algorithm is always intended for code generation. The

`%#codegen` directive, or the absence of it, does not change the error checking behavior in the context of the `MATLAB Function` block. For more information see "Compilation Directive %#codegen".

To support visualization of data, the `MATLAB Function` block supports calls to MATLAB functions for simulation only. See "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.

From `MATLAB Function` blocks, you can also call functions defined in a `Simulink Function` block. You can call Stateflow functions with **Export Chart Level Functions (Make Global)** and **Allow exported functions to be called by Simulink** checked in the chart Properties dialog box.

In the Ports and Data Manager, you can declare a block input to be a Simulink parameter instead of a port. The `MATLAB Function` block also supports inheritance of types and size for inputs, outputs, and parameters. You can also specify these properties explicitly. See "Type Function Arguments", "Size Function Arguments", and "Add Parameter Arguments" for descriptions of variables that you use in `MATLAB Function` blocks.

Recursive calls are not allowed in `MATLAB Function` blocks.

By default, `MATLAB Function` blocks have direct feedthrough enabled. To disable it, in the Ports and Data Manager, clear the **Allow direct feedthrough** check box. Nondirect feedthough enables semantics to ensure that outputs rely only on current state. Using nondirect feedthrough enables you to use `MATLAB Function` blocks in a feedback loop and prevent algebraic loops.

Data Type Support

The `MATLAB Function` block accepts inputs of any type that Simulink supports, including fixed-point and enumerated types. For more information, see "Data Types Supported by Simulink".

Data types supported by MATLAB but not supported by Simulink may not be passed between the Simulink model and the function within the `MATLAB Function` block. These types may be used within the `MATLAB Function` block.

For more information on fixed-point support for this block, refer to “Fixed-Point Data Types with MATLAB Function Block” and “MATLAB Function Block with Data Type Override”.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes (default). To disable, in the Ports and Data Manager, clear the **Allow direct feedthrough** check box.
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced in R2011a

MATLAB System

Include System object in model

Library

User-Defined Functions

Description

The `MATLAB System` block brings existing System objects (based on matlab.System) into Simulink. It also enables you to use System object APIs to develop new blocks for Simulink. For more information on this block, see "MATLAB System Block".

For interpreted execution, the model simulates the block using the MATLAB execution engine.

For code generation, the model simulates the block using code generation (using the subset of MATLAB® code supported for code generation). The `MATLAB System` block supports only a subset of the functions available in MATLAB. See "Functions and Objects Supported for C and C++ Code Generation — Alphabetical List" for a complete list of functions. These functions include those in common categories, such as:

- "Array vs. Matrix Operations", like `plus`, `minus`, and `power`
- Matrix operations, like `size` and `length`
- Advanced matrix operations, like `lu`, `inv`, `svd`, and `chol`
- Trigonometric functions, like `sin`, `cos`, `sinh`, and `cosh`

System Objects

To use the `MATLAB System` block, you must first have a new System object™ or use an existing one. For more information, see "System Object Integration".

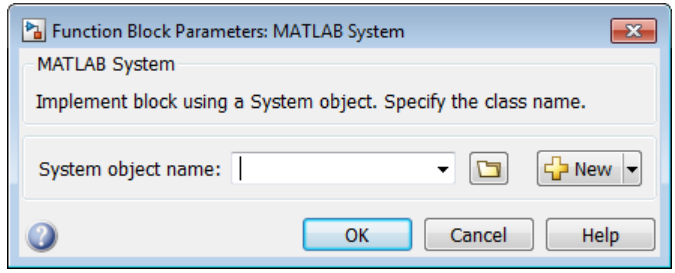
Data Type Support

The `MATLAB System` block accepts inputs of most types that Simulink supports. It does not support enumerated data types or buses. For more information, see "Data Types Supported by Simulink".

For information on fixed-point support for this block, see "Code Acceleration and Code Generation from MATLAB".

The `MATLAB System` block supports Simulink frames. For more information, see "Sample- and Frame-Based Concepts".

Parameters and Dialog Box



System object name

Specify the full name of the user-defined System object class without the file extension. This entry is case sensitive. The class name must exist on the MATLAB path.

You can specify a System object name in one of these ways:

- Enter the name in the text box.
- Click the list arrow attached to the text box. If valid System objects exist in the current folder, the names appear in the list. Select a System object from this list.
- Browse to a folder that contains a valid System object. If the folder is not on your MATLAB path, the software prompts you to add it.

If you need to create a System object, you can create one from a template by clicking **New**.

After you save the System object, you can enter the name in the **System object name** text box.

Settings

Default: None

Tips

Use the full name of the user-defined System object class name. The block does not accept a MATLAB variable that you have assigned to a System object class name.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

New

Click this button to create a System object from a template.

Select one of these options.

- `Basic`

  Starts MATLAB Editor and displays a template for a simple System object using the fewest System object methods.
- `Advanced`

  Starts MATLAB Editor and displays a template for a more advanced System object using most of the System object methods.
- `Simulink Extension`

  Starts MATLAB Editor and displays a file that contains utilities for customizing the block for Simulink. This is the same file available in MATLAB when you select **New** > **System Object** > **Simulink Extension**.

After you save the System object, you can enter the name in the **System object name** text box.

Settings

Default: `Basic`

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Simulate using

Select the simulation mode.

Settings

Default: `Code generation`

`Code generation`

On the first model run, simulate and generate code for `MATLAB System` block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, system objects accept a maximum of 32 inputs.

`Interpreted execution`

Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

Dependency

After you assign a valid System object class name to the block, the next time you open the block dialog box, the parameter is visible. This parameter appears for every `MATLAB System` block. You cannot remove it.

- If the block has no tabs, this parameter appears at the bottom of the dialog box.
- If the block has multiple tabs, this parameter appears at the bottom of the first tab of the dialog box.

Saturate on integer overflow

Specify whether overflows saturate.

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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.

Dependency

This check box appears when you use the showFiSettingsImpl method in the System object.

Command-Line Information

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Treat these inherited Simulink signal types as fi objects

Specify which inherited data types to treat as fi data types.

Settings

Default: `Fixed-point`

`Fixed-point`

Treat fixed-point data types as fi data types.

`Fixed-point & Integer`

Treat fixed-point and integer data types as fi data types.

Dependency

This check box appears when you use the showFiSettingsImpl method in the System object.

MATLAB System fimath

Specify which fixed-point math settings to use.

Settings

Default: Same as MATLAB

`Fixed-point`

Use the current MATLAB fixed-point math settings.

Specify Other

Enable the edit box for specifying the desired fixed-point math settings. For information on setting fixed-point math, see `fimath`.

Dependency

This check box appears when you use the showFiSettingsImpl method in the System object.

Characteristics

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

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

See Also

`MATLAB Function`

More About

- "MATLAB System Block"
- "What Are System Objects?"

Introduced in R2013b

Memory

Output input from previous time step

Library

Discrete

Description

The Memory block holds and delays its input by one major integration time step. When placed in an iterator subsystem, it holds and delays its input by one iteration. This block accepts continuous and discrete signals. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds and delays all elements of the vector by the same time step.

You specify the block output for the first time step using the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior. However, you cannot specify the sample time. This block's sample time depends on the type of solver used, or you can specify to inherit it. The **Inherit sample time** parameter determines whether sample time is inherited or based on the solver.

Tip Avoid using the Memory block when both these conditions are true:

- Your model uses the variable-step solver `ode15s` or `ode113`.
- The input to the block changes during simulation.

When the Memory block inherits a discrete sample time, the block is analogous to the `Unit Delay` block. However, the Memory block does not support state logging. If logging the final state is necessary, use a `Unit Delay` block instead.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
`Unit Delay`	Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.	• `sldemo_enginewc` (Compression subsystem)
Memory	Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.	• `sldemo_bounce` • `sldemo_clutch` (Friction Mode Logic/Lockup FSM subsystem)
`Zero-Order Hold`	Convert an input signal with a continuous sample time to an output signal with a discrete sample time.	• `sldemo_radar_eml` • `aero_dap3dof`

Overview of Block Capabilities

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of initial condition	Yes	Yes	No, because the block output at time t = 0 must match the input value.

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of sample time	Yes	No, because the block can only inherit sample time (from the driving block or the solver used for the entire model).	Yes
Support for frame-based signals	Yes	No	Yes
Support for state logging	Yes	No	No

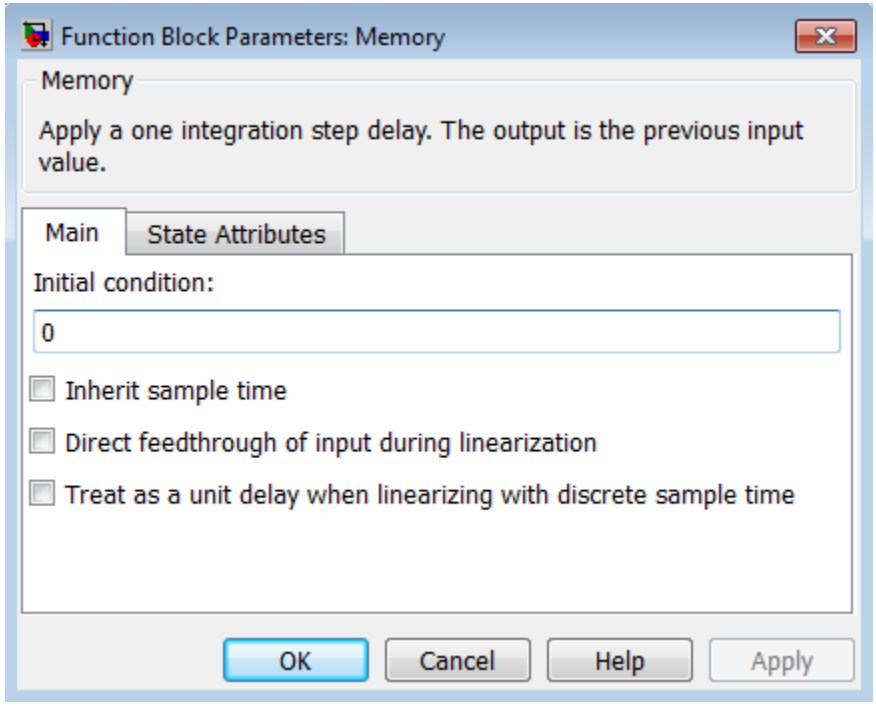
Data Type Support

The Memory block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

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

Parameters and Dialog Box

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



Initial condition

Specify the output at the initial integration step. This value must be 0 when you do not use a built-in input data type. Simulink does not allow the initial output of this block to be `inf` or `NaN`.

Inherit sample time

Select to inherit the sample time from the driving block:

- If the driving block has a discrete sample time, the block inherits the sample time.
- If the driving block has a continuous sample time, selecting this checkbox has no effect. The sample time depends on the type of solver used for simulating the model.

When this check box is cleared, the block sample time depends on the type of solver used for simulating the model:

- If the solver is a variable-step solver, the block sample time is continuous but fixed in minor time step: `[0, 1]`.

- If the solver is a fixed-step solver, the `[0, 1]` sample time converts to the solver step size after sample-time propagation.

Direct feedthrough of input during linearization

Select to output the input during linearization and trim. This selection sets the block mode to direct feedthrough.

Selecting this check box can cause a change in the ordering of states in the model when using the functions `linmod`, `dlinmod`, or `trim`. To extract this new state ordering, use the following commands.

First compile the model using the following command, where `model` is the name of the Simulink model.

```
[sizes, x0, x_str] = model([],[],[],'lincompile');
```

Next, terminate the compilation with the following command.

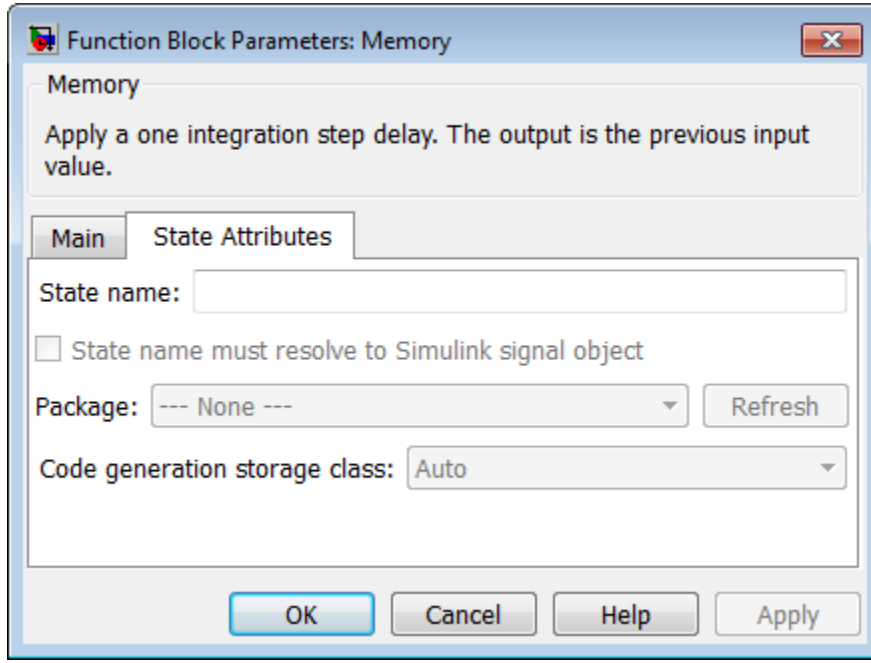
```
model([],[],[],'term');
```

The output argument, `x_str`, which is a cell array of the states in the Simulink model, contains the new state ordering. When passing a vector of states as input to the `linmod`, `dlinmod`, or `trim` functions, the state vector must use this new state ordering.

Treat as a unit delay when linearizing with discrete sample time

Select to linearize the `Memory` block to a unit delay when the `Memory` block is driven by a signal with a discrete sample time.

The **State Attributes** pane of the Memory block dialog box appears as follows:



State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see "Discrete Block State Naming in Generated Code" 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**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

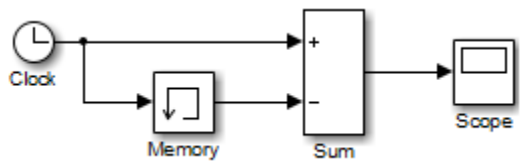
- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

See "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation for more information.

Examples of Memory Block Usage

Usage with the Clock Block

The following model shows how to display the step size in a simulation. The Sum block subtracts the time at the previous step, which the Memory block generates, from the current time, which the `Clock` block generates.

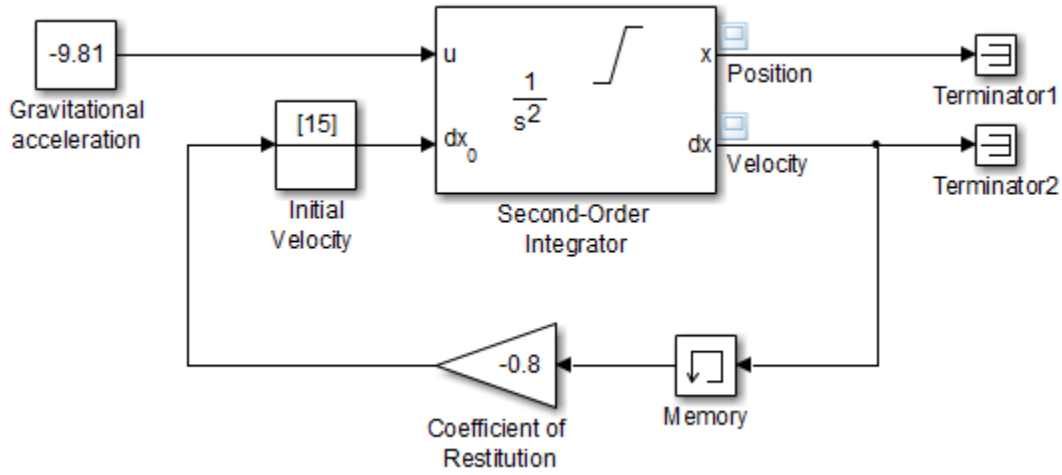


Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a fixed-step solver. Therefore, the sample time of the Memory block is the solver step size, or 1.

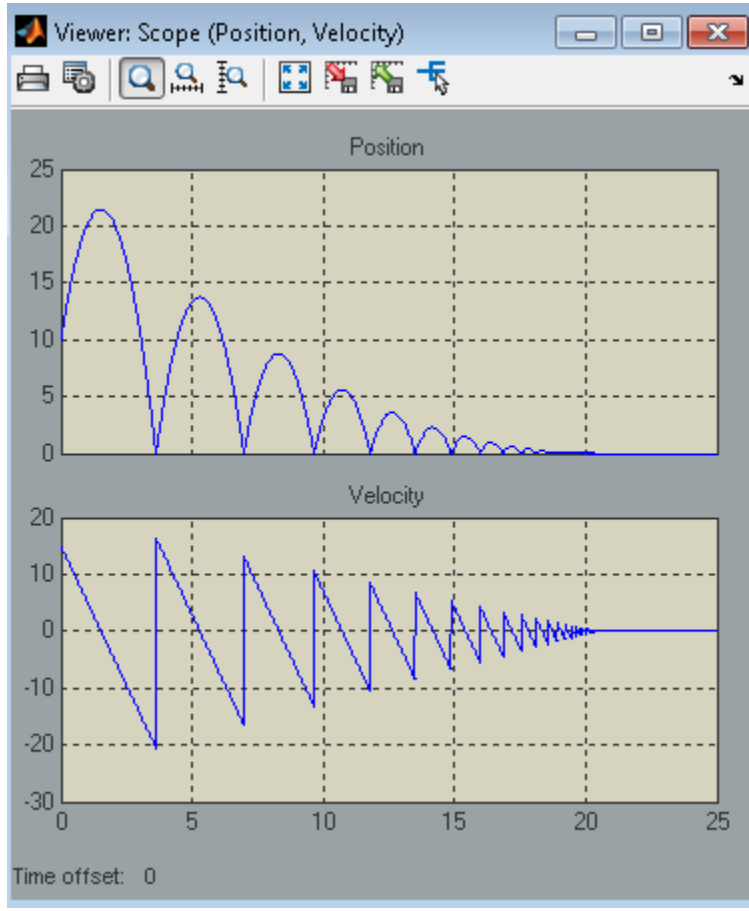
If you replace the Memory block with a `Unit Delay` block, you get the same results. The Unit Delay block inherits a discrete sample time of 1.

Usage with the Second-Order Integrator Block

The `sldemo_bounce` model shows how a bouncing ball reacts after being tossed into the air. The **dx** port of the `Second-Order Integrator` block and the `Memory` block capture the velocity of the ball just before it hits the ground.



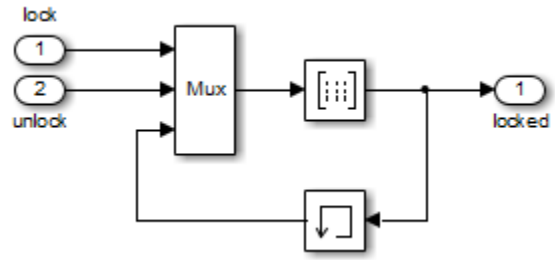
Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step `(ode23)` solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: `[0, 1]`. When you run the model, you get the following results:



If you replace the Memory block with a `Unit Delay` block, you get the same results. However, a warning also appears due to the discrete Unit Delay block inheriting a continuous sample time.

Usage with the Combinatorial Logic Block

The `sldemo_clutch` model shows how you can use the `Memory` block with the `Combinatorial Logic` block to implement a finite-state machine. This construct appears in the `Friction Mode Logic/Lockup FSM` subsystem.



Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step `(ode23)` solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: `[0, 1]`.

Bus Support

The Memory block is a bus-capable block. The input can be a virtual or nonvirtual bus signal subject to the following restrictions:

- **Initial condition** must be zero, a nonzero scalar, or a finite numeric structure.
- If **Initial condition** is zero or a structure, and you specify a **State name**, the input cannot be a virtual bus.
- If **Initial condition** is a nonzero scalar, you cannot specify a **State name**.

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Memory block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a `Rate Transition` block to change the sample time of an individual signal, or of all signals in a bus. See “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. You can specify the **Initial condition** parameter with:

- The value `0`. In this case, all of the individual signals in the array of buses use the initial value `0`.

- An array of structures that specifies an initial condition for each of the individual signals in the array of buses.
- A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Depends on the type of solver used. If you select the **Inherit sample time** check box, the block inherits sample time from the driving block.
Direct Feedthrough	No, except when you select **Direct feedthrough of input during linearization**
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Unit Delay`, `Zero-Order Hold`

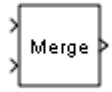
Introduced before R2006a

Merge

Combine multiple signals into single signal

Library

Signal Routing



Description

The `Merge` block combines its inputs into a single output line whose value at any time is equal to the most recently computed output of its driving blocks. You can specify any number of inputs by setting the block's **Number of inputs** parameter.

Use `Merge` blocks only to interleave input signals that update at different times into a combined signal in which the interleaved values retain their separate identities and times. To combine signals that update at the same time into an array or matrix signal, use a `Concatenate` block.

`Merge` blocks assume that all driving signals share the same signal memory. The shared signal memory should be accessed only in mutually exclusive fashion. Therefore, always use alternately executing subsystems to drive `Merge` blocks. 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.

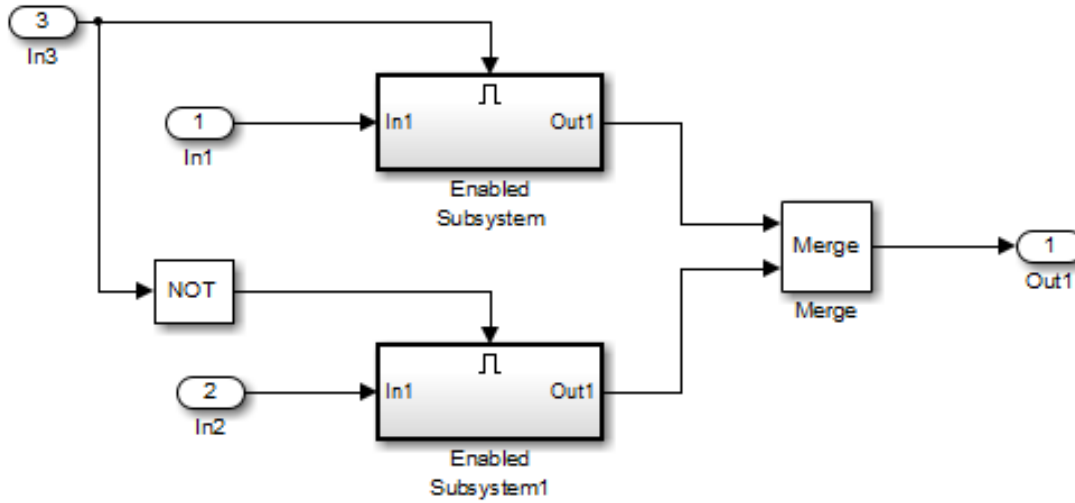
- Write your control logic to ensure that at most one of the driving conditionally-executed subsystems executes at any time step.
- Do not connect more than one input of a `Merge` block to the same conditionally-executed subsystem.
- Always connect a `Merge` block to at least two input signals.
- Ensure 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 1-1110 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`.
- If the output of a Model block is coming from a MATLAB Function block or a Stateflow chart, do not connect that output port to the input port of the Merge block.
- In the code generation workflow, when the `Merge` block receives a constant value and non-constant sample times, one of these conditions must hold. Otherwise Simulink displays an error.
  - The source of the constant value is a grounded signal.
  - The source of the constant value is a constant block with a non-tunable parameter.
    - There is only one constant block that feeds the `Merge` block.
    - All other input signals to the `Merge` block are from conditionally executed subsystems.
    - The `Merge` block and outport blocks of all conditionally executed subsystems should not specify any initial outputs.

Merge Block Usage

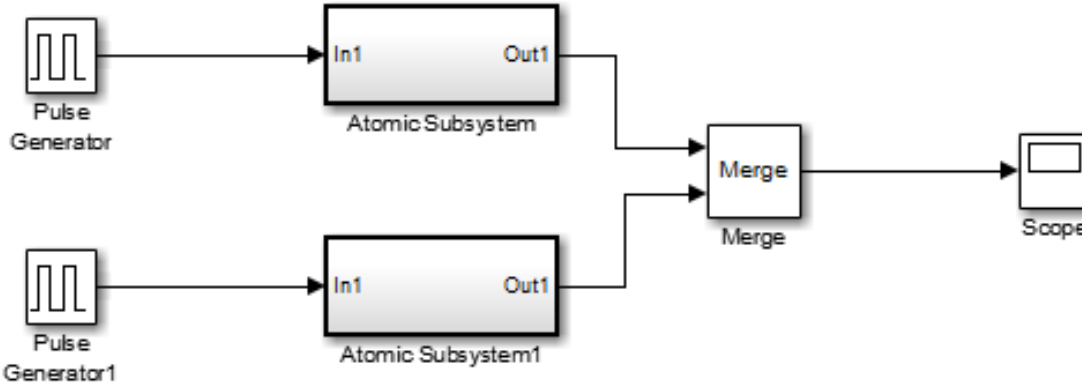
For each input of a `Merge` block, the topmost non-atomic and nonvirtual source must be a conditionally-executed subsystem that is not an Iterator Subsystem.

You can use the Model Advisor to check `Merge` block usage in your model. For more information, see "Check usage of Merge blocks" on page 8-41.

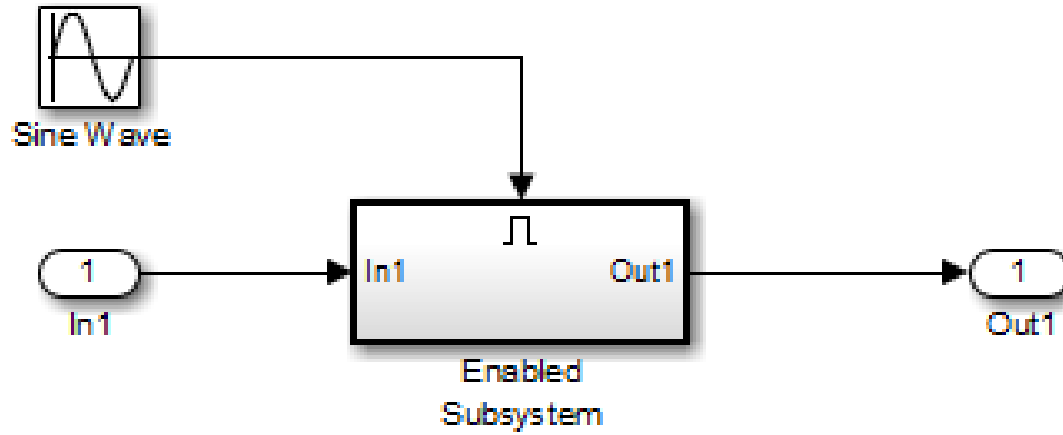
The following schematic shows valid `Merge` block usage, merging signals from two conditionally-executed subsystems.



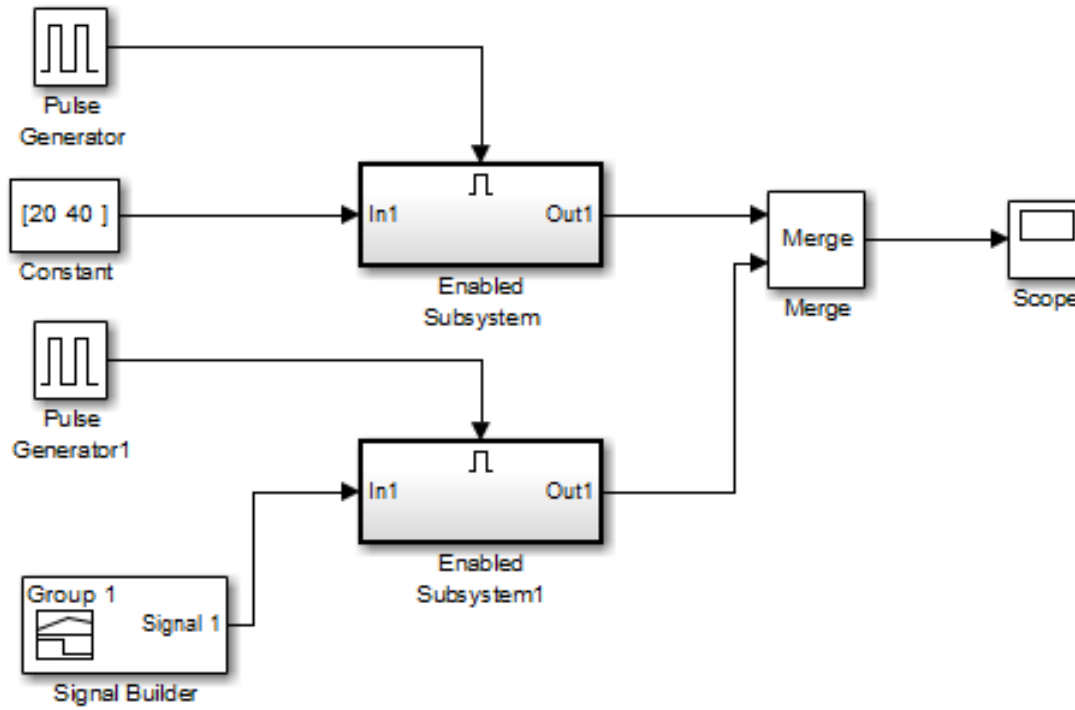
The following example is also a valid `Merge` block usage, where the topmost nonatomic, nonvirtual source is a conditionally executed subsystem.



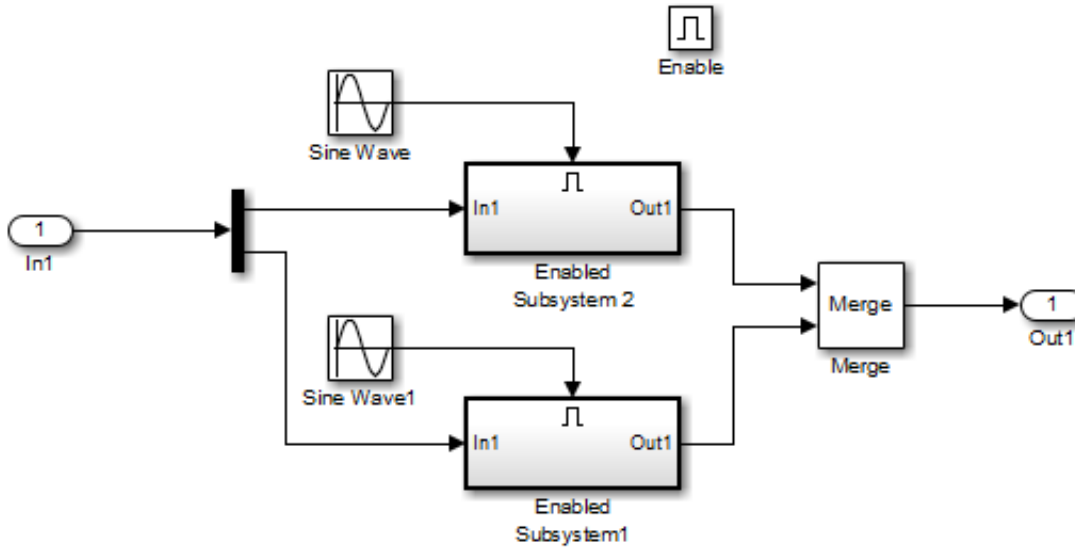
Each Atomic Subsystem block contains an enabled subsystem.



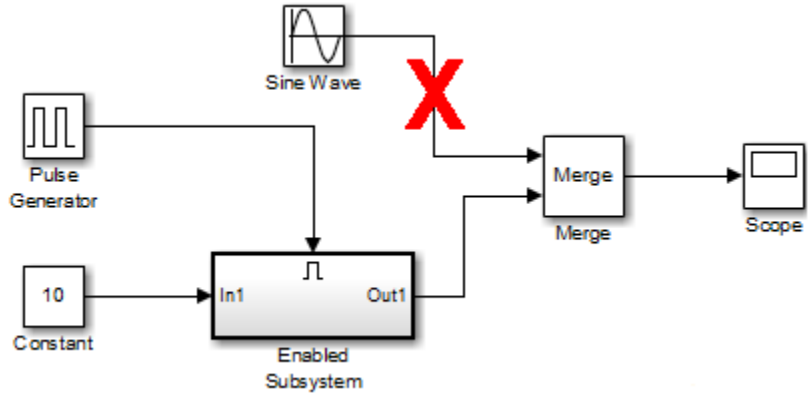
You can also use multiple Merge blocks at different levels of the model hierarchy. The following example contains a Merge block at the model root.



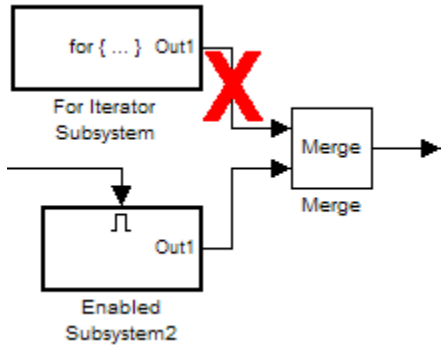
A Merge block is also located inside the Enabled Subsystem block, one level down.



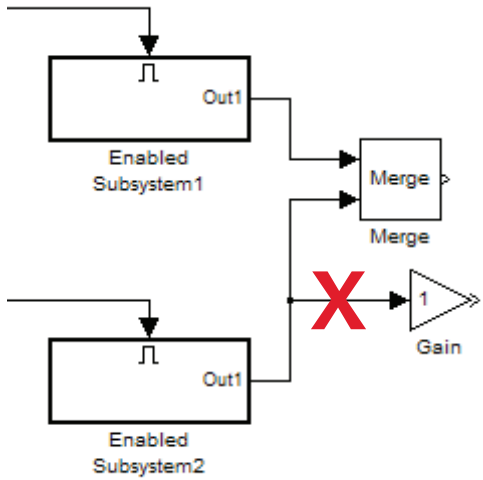
A `Merge` block *cannot* connect to a `Sine Wave` block because a `Sine Wave` block is not a conditionally-executed subsystem.



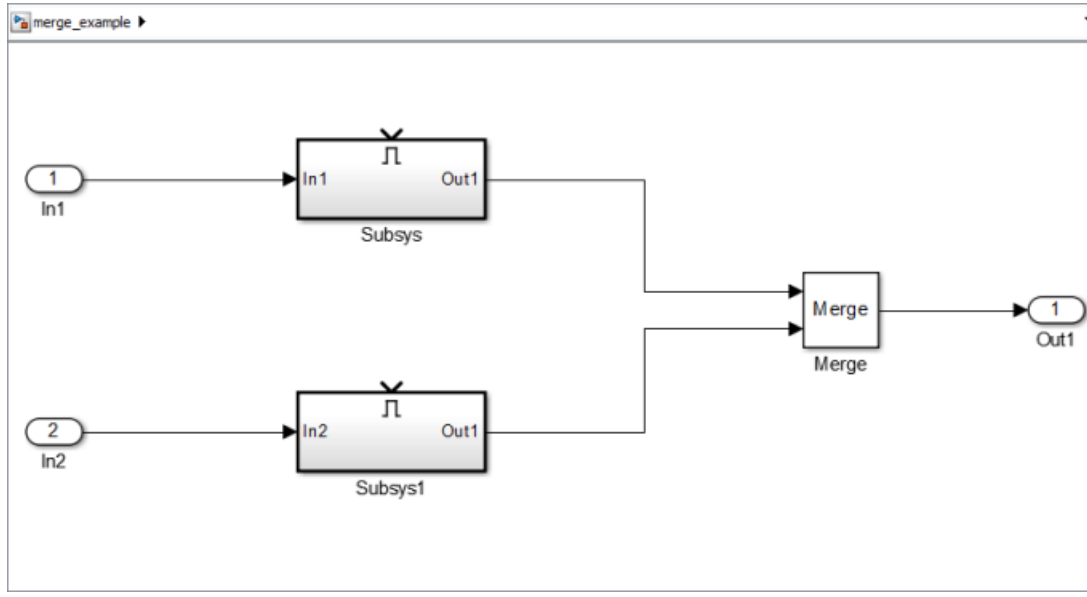
A `Merge` block *cannot* connect to a `For Iterator Subsystem`.

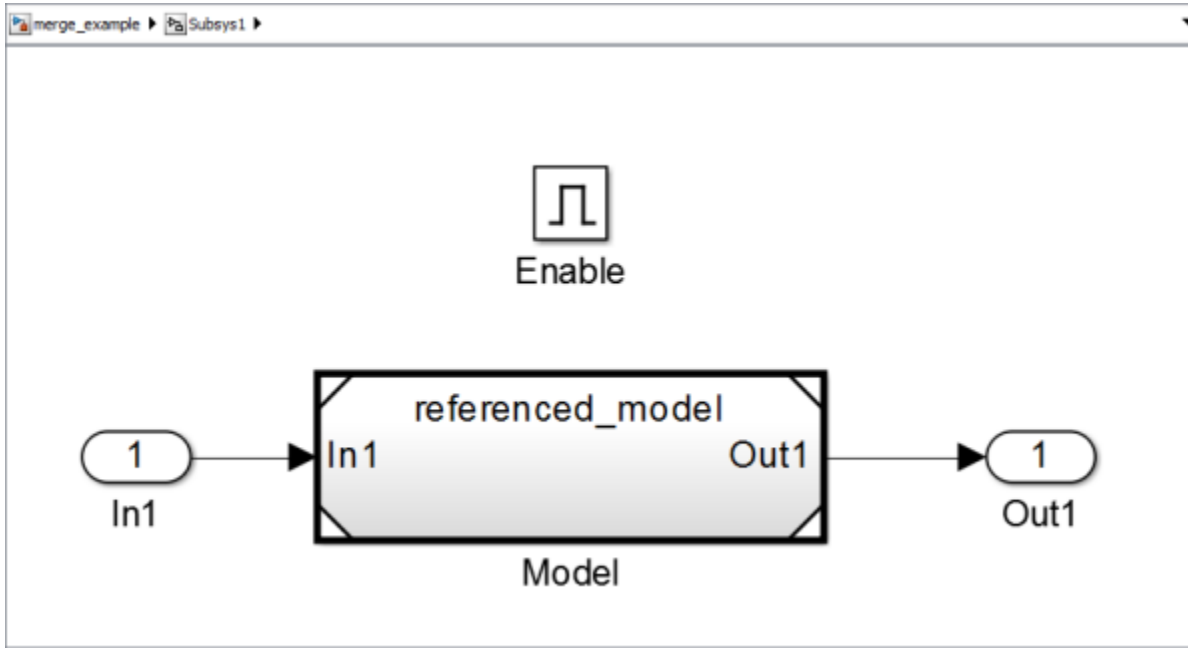


A `Merge` block *cannot* connect to a branched signal.

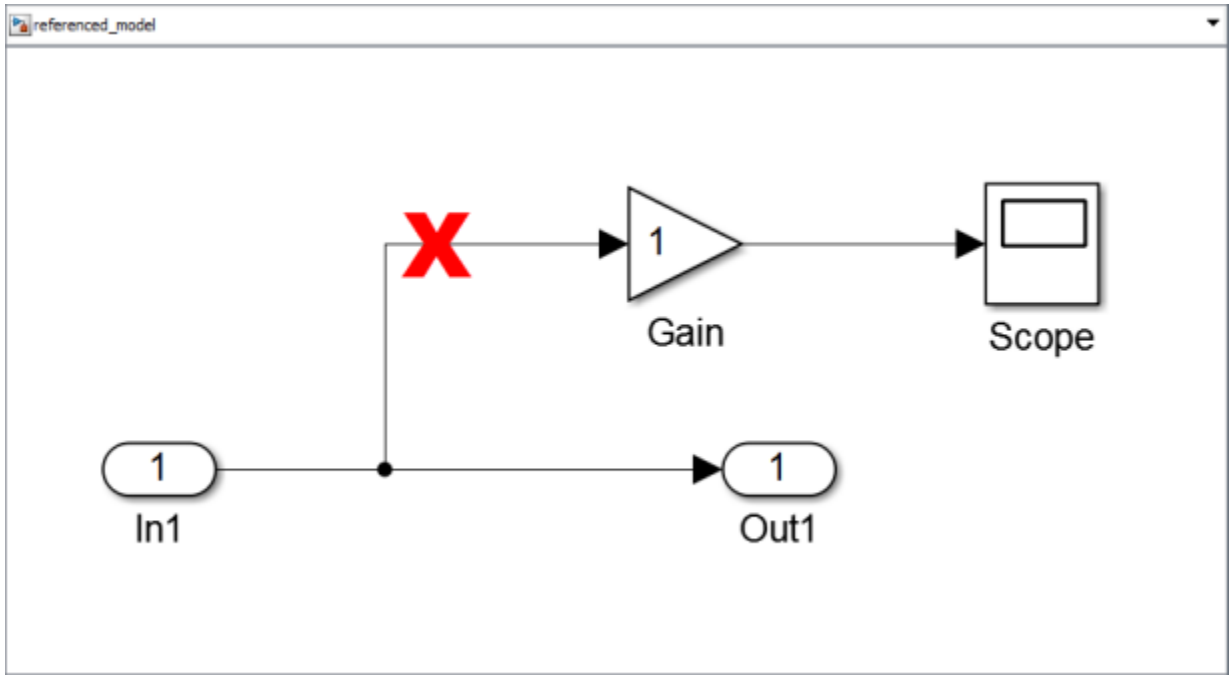


In the following model, the referenced model has a signal that branches. The subsystem `Subsys1` includes a Model block that references `referenced_model`. It includes a block that inputs to a block in the referenced model and also inputs to the `Merge` block that is outside of the referenced model.


merge_example
1
In1
In1
Out1
Subsys
2
In2
In2
Out1
Subsys1
Merge
Merge
1
Out1

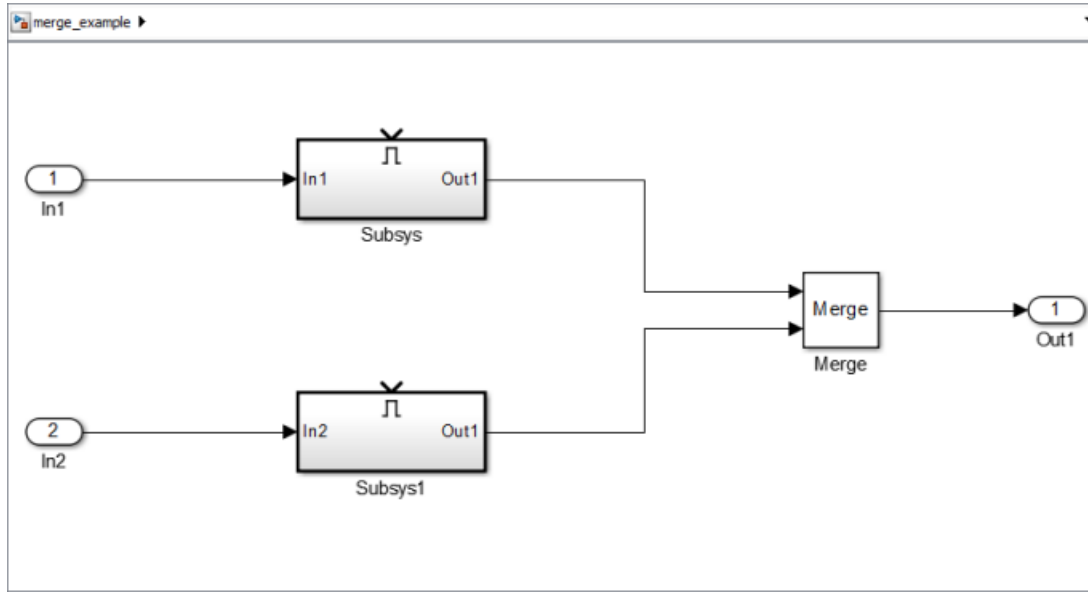

merge_example
Subsys1
Enable
referenced_model
1
In1
In1
Out1
1
Out1
Model

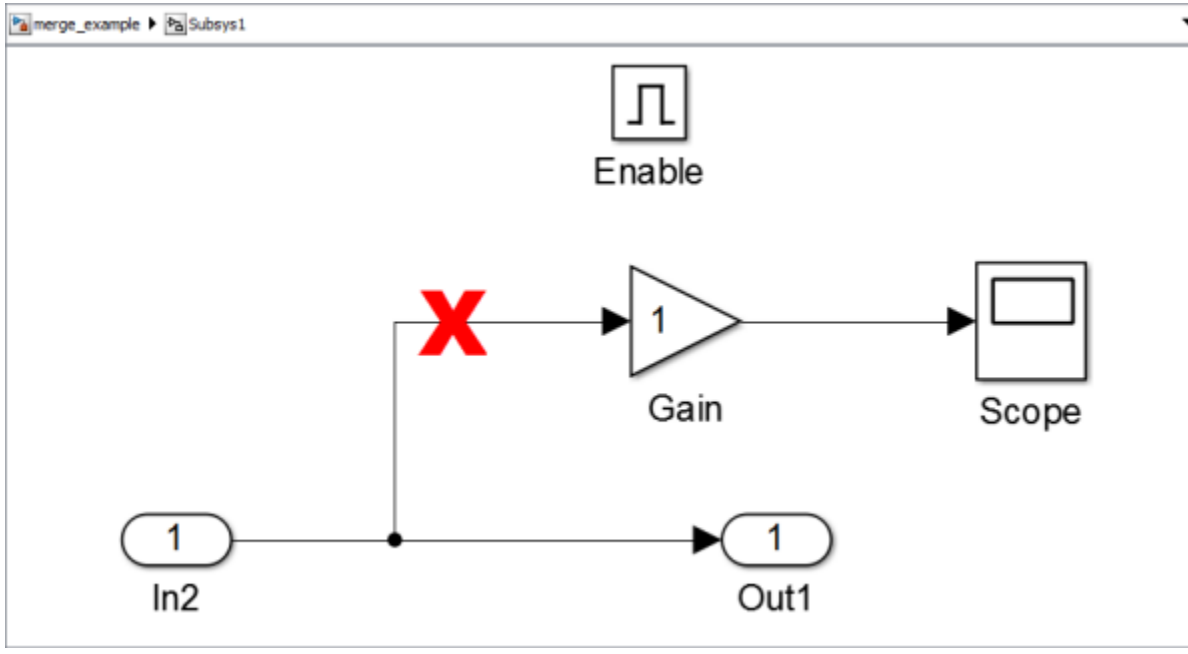
The referenced model includes a signal that incorrectly branches to a `Gain` block and to the `Out1` `Outport` block, which connects to the `Merge` block that is outside of the referenced model.



The following example also shows a branched signal in a subsystem that connects to a Merge block, which is *not allowed* if you use the default setting of `Classic` for the **Model Configuration Parameters** > **Diagnostics** > **Underspecified initialization detection** parameter.

If you set the **Underspecified initialization detection** parameter to `Simplified`, then the following example does *not* generate an error. For more information on simplified initialization mode, see "Underspecified initialization detection".


merge_example
1
In1
In1
Out1
Subsys
2
In2
In2
Out1
Subsys1
Merge
Merge
1
Out1


merge_example
Subsys1
Enable
1
Gain
Scope
1
In2
1
Out1

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, the block's initial output depends on the initialization mode and the driving blocks. In Simplified initialization mode, for an unspecified (empty matrix `[ ]`) value of **Initial output**, the block uses the default initial value of the output data type. For information on the default initial value, see "Initializing Signal Values". In Classic initialization mode, for an unspecified (empty matrix `[ ]`) value of **Initial output**, the initial output of the block equals the most recently evaluated initial output of the driving blocks. Since the initialization ordering for these sources may vary, initialization may be inconsistent for the simulation and the code generation of a model. For example, the following model can produce inconsistent initialization:

- The model contains a Merge block with two inputs: one driven by a Stateflow chart and the other driven by a conditionally executed subsystem (such as an Enabled Subsystem).
- The Merge block **Initial output** parameter is unspecified (that is, specified as empty matrix (`[ ]`)) and the model uses Classic initialization mode.
- The Stateflow chart initializes the output being merged to `val1`.
- The conditionally executed subsystem initializes the output being merged to different value `val2`.
- Both the Stateflow chart and the conditionally executed subsystem do not execute at the first time step.

Because the initialization ordering may vary, the output of the Merge block at the first time step is `val1` if the Stateflow chart initializes last and `val2` if the conditionally executed subsystem initializes last. The initialization ordering is different for simulation and code generation.

To address this issue, use one of the following approaches:

- Set the **Initial output** parameter of the Merge block, unless the output port of the Merge block connects to another Merge block.
- Turn on simplified initialization mode, set **Configuration Parameters** > **All Parameters** > **Underspecified initialization detection** 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.

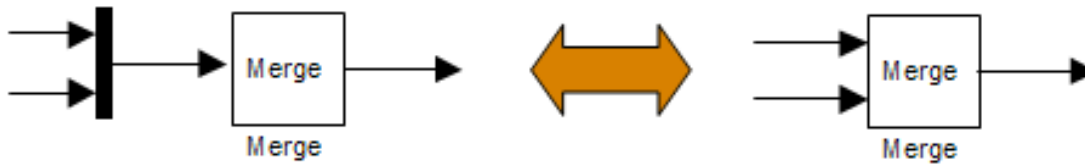
To upgrade your model to simplified initialization mode, use the Model Advisor check "Check usage of Merge blocks" on page 8-41.

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

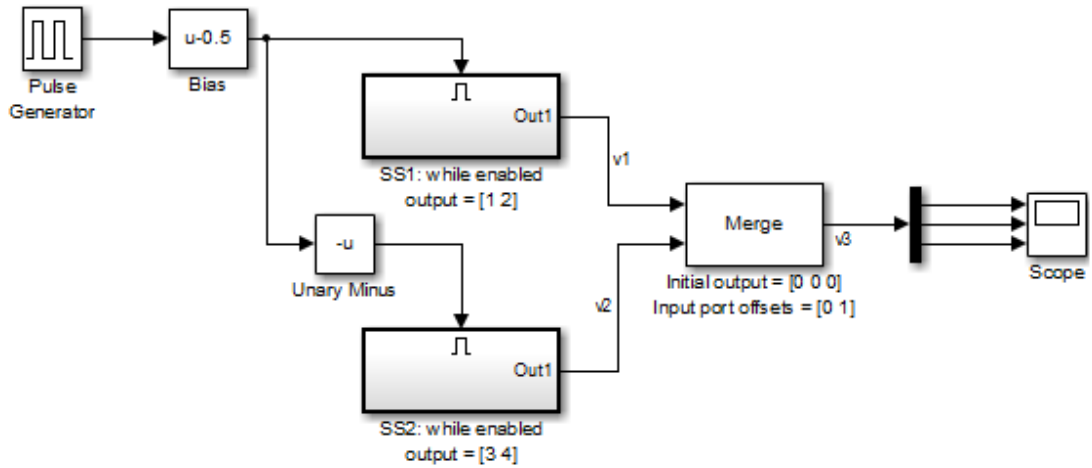
The `Merge` block accepts only inputs of equal dimensions and outputs a signal of the same dimensions as the inputs, unless you select the **Allow unequal port widths** parameter.

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

$$\max(w_1+o_1,\ w_2+o_2,\ \ldots\ w_n+o_n)$$

where w_1, ... w_n are the widths of the input signals and o_1, ... o_n are the offsets for the input signals.

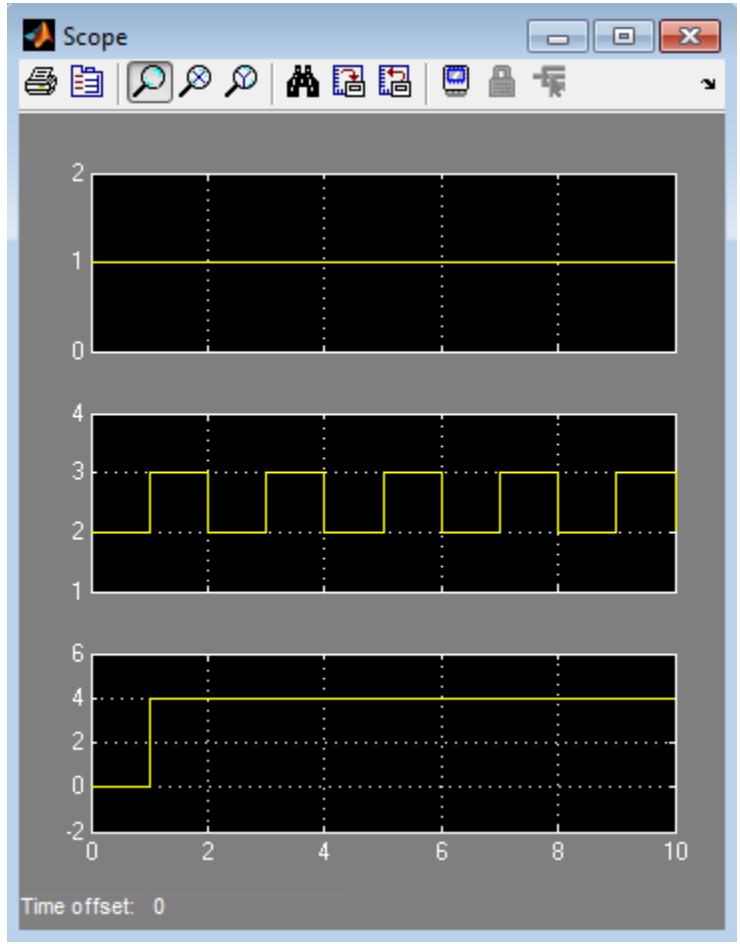
Suppose that you have the following block diagram:



The `Merge` block has the following output width:

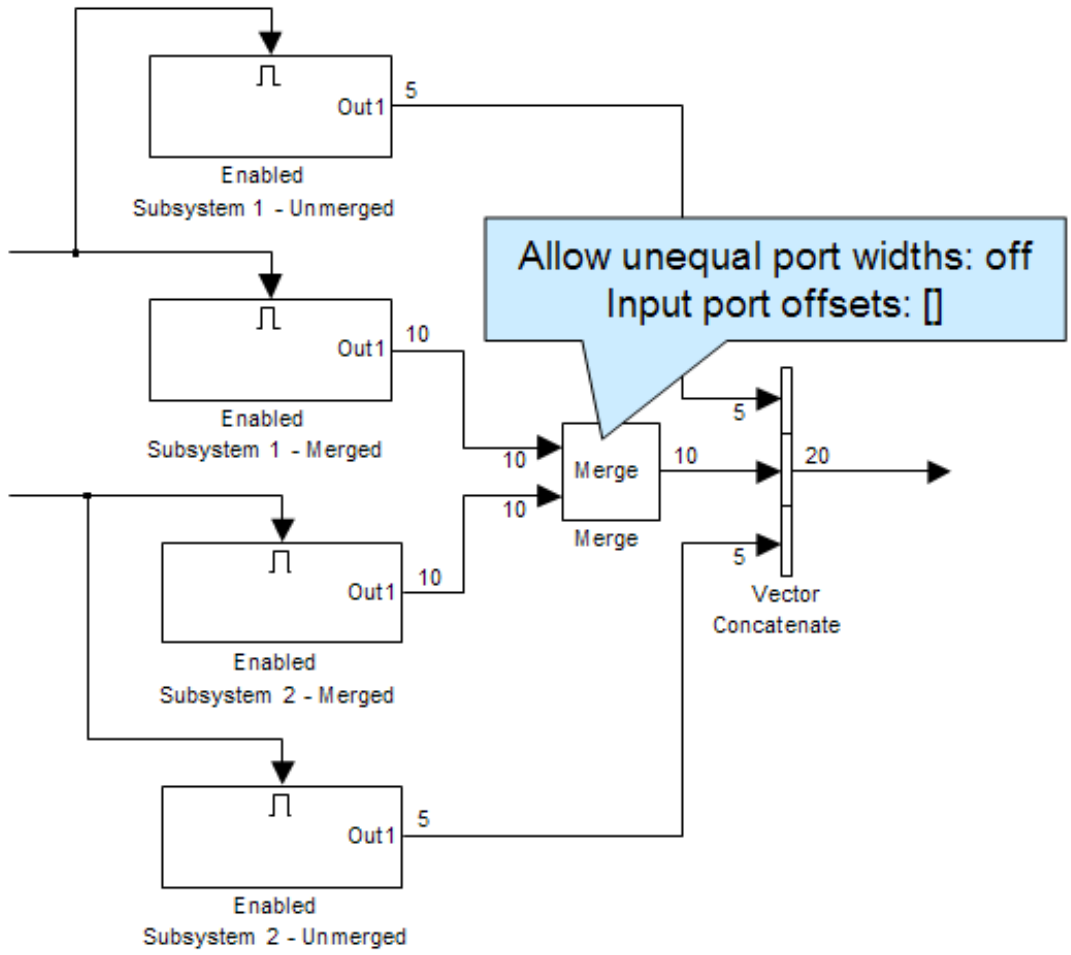
```
max(2+0,2+1)=3
```

In this example, the offset of `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:



If you use Simplified Initialization Mode, you must clear the **Allow unequal port widths** check box. The input port offsets for all input signals must be zero.

Consider using `Merge` blocks only for signal elements that require true merging. Other elements can be combined with merged elements using the `Concatenate` block, as shown in the following example.

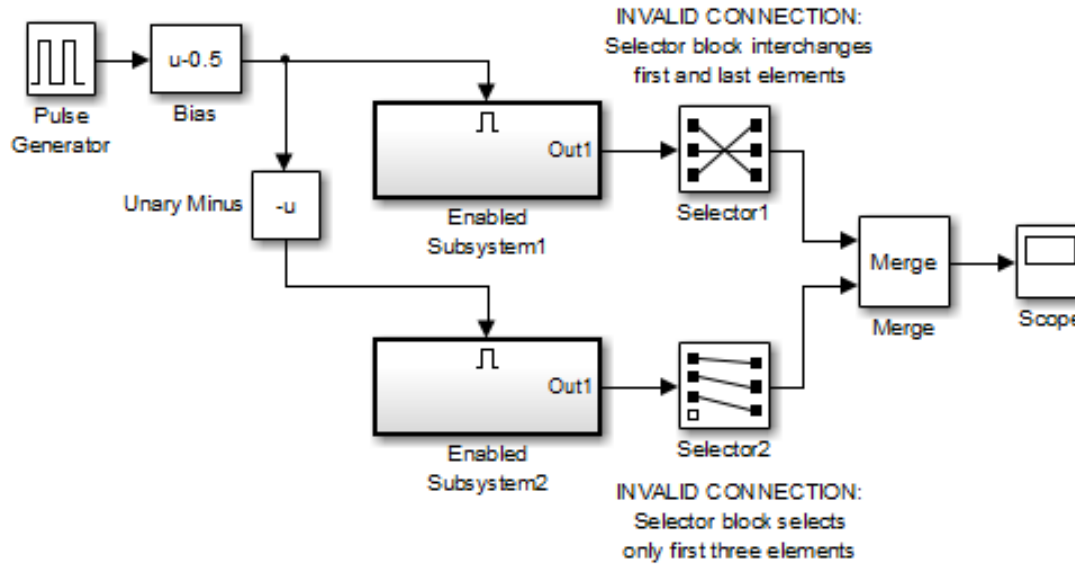


For more information on simplified initialization mode, see “Underspecified initialization detection”.

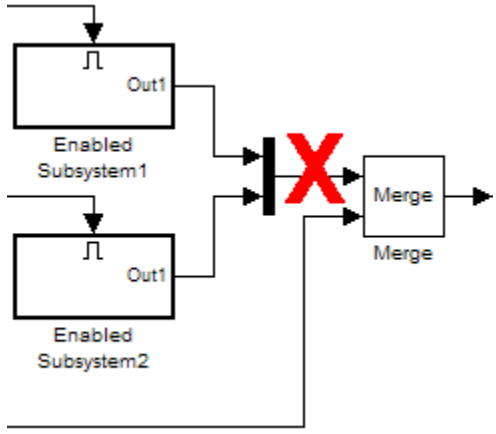
Combining or Reordering of Input Signals

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.

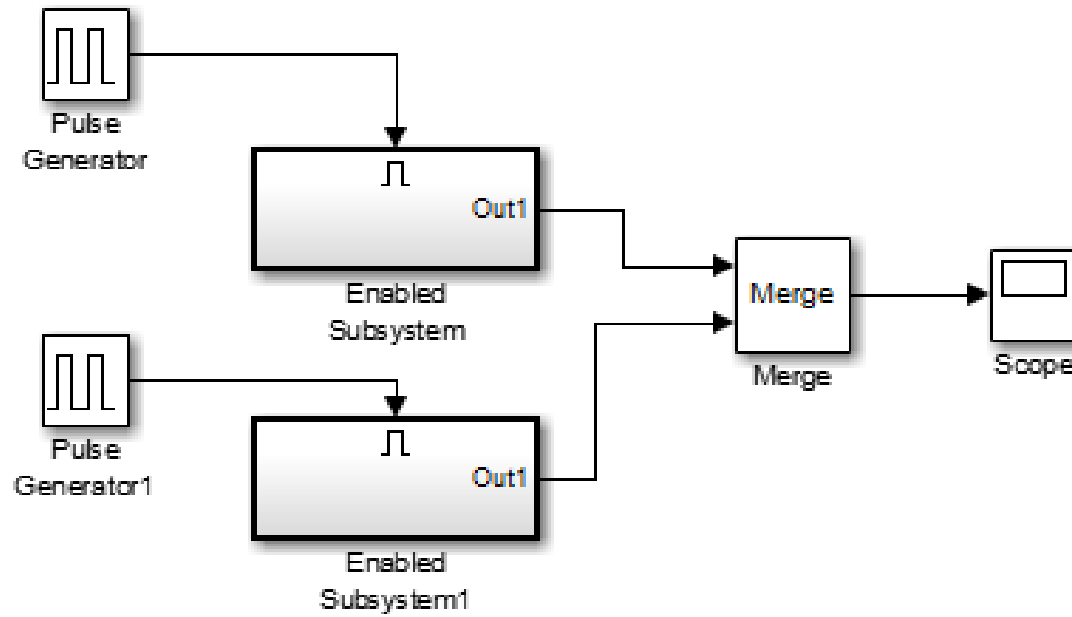
For example, in the following block diagram, the Merge block does not accept the output of the first Selector block because the Selector block interchanges the first and last elements of the vector signal. Similarly, the `Merge` block does not accept the output of the second `Selector` block because the `Selector` block selects only the first three elements.



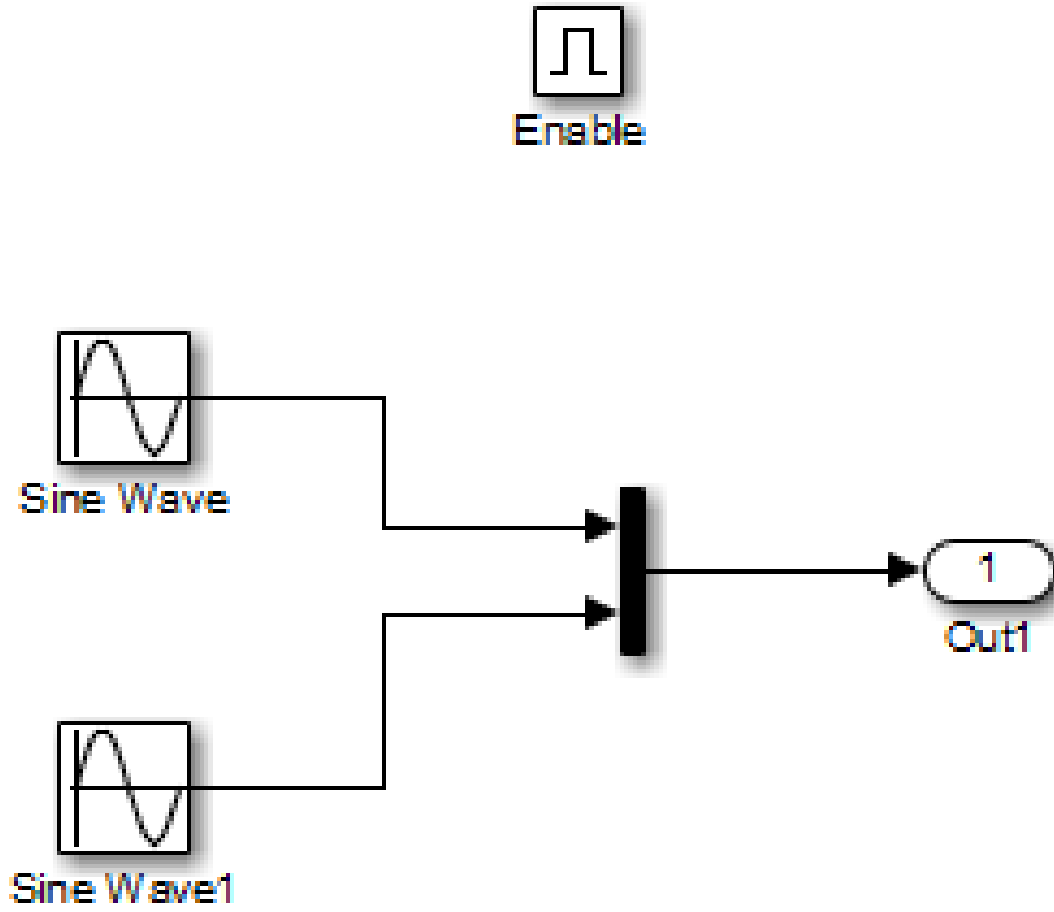
If you use simplified initialization mode, the following arrangement is *not allowed* because two signals are being combined outside of a conditionally-executed subsystem.



You can, however, combine or reorder `Merge` block input signals within a conditionally-executed subsystem. For example, the following model is valid.



Each Enabled Subsystem contains the following blocks.



For more information on simplified initialization mode, see “Underspecified initialization detection”.

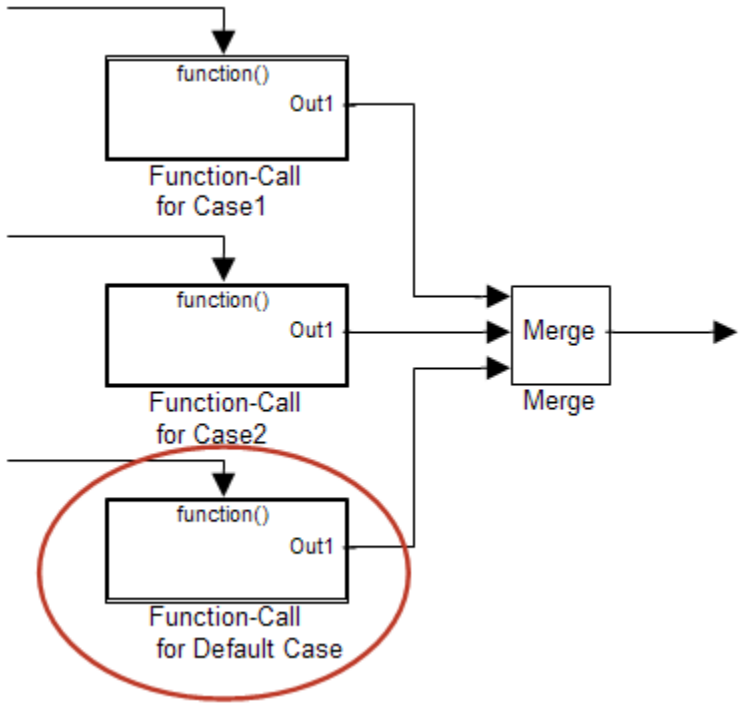
Conditionally-Executed Subsystem Outport Reset

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:



For more information on simplified initialization mode, see “Underspecified initialization detection”.

Merging S-Function Outputs

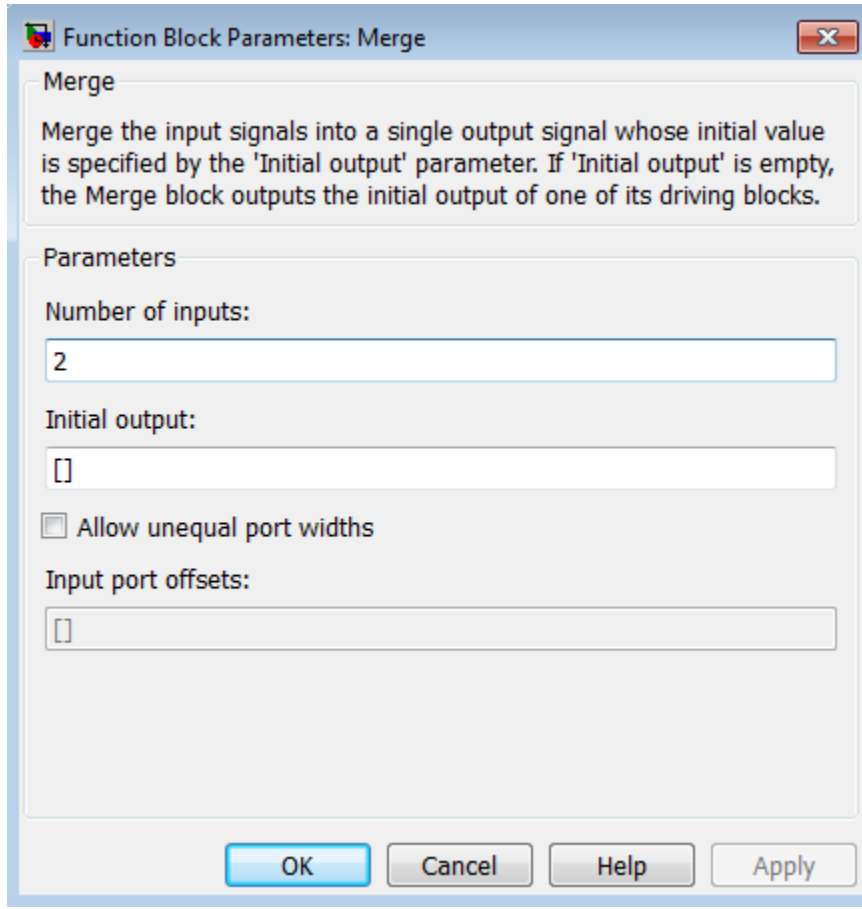
The Merge block can merge a signal from an S-Function block only if the memory used to store the S-Function block's output is reusable. Simulink software displays an error message if you attempt to update or simulate a model that connects a nonreusable port of an S-Function block to a Merge block. See `ssSetOutputPortOptimOpts` for more information.

Data Type Support

The Merge block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. All inputs must be of the same data type and numeric type.

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

Parameters and Dialog Box



Number of inputs

Specify the number of input ports to merge.

Initial output

Specify the initial value of output For more information, see .

Allow unequal port widths

Select this check box to allow the block to accept inputs having different numbers of elements.

Input port offsets

Enter a vector to specify the offset of each input signal relative to the beginning of the output signal.

Bus Support

The Merge block is a bus-capable block. The inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- The number of inputs must be greater than one.
- **Initial output** must be zero, a nonzero scalar, or a finite numeric structure.
- **Allow unequal port widths** must be disabled.
- All inputs to the merge must be buses and must be equivalent (same hierarchy with identical names and attributes for all elements).

For information about specifying an initial condition structure, see "Specify Initial Conditions for Bus Signals".

All signals in a nonvirtual bus input to a Merge block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a `Rate Transition` block to change the sample time of an individual signal, or of all signals in a bus. See "Composite Signals" and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Merge block. For details about defining and using an array of buses, see "Combine Buses into an Array of Buses". Using an array of buses with a Merge block involves these limitations:

- **Allow unequal port widths** — Clear this parameter.
- **Number of inputs** — Set to a value of 2 or greater.
- **Initial condition** — You can specify this parameter with:
  - The value `0`. In this case, all of the individual signals in the array of buses use the initial value `0`.
  - An array of structures that specifies an initial condition for each of the individual signals in the array of buses.

- A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

Characteristics

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

Introduced before R2006a

MinMax

Output minimum or maximum input value

Library

Math Operations

Description

The MinMax block outputs either the minimum or the maximum element or elements of the inputs. You can choose the function to apply by selecting one of the choices from the **Function** parameter list.

If the block has one input port, the input must be a scalar or a vector. The block outputs a scalar equal to the minimum or maximum element of the input vector.

If the block has multiple input ports, all nonscalar inputs must have the same dimensions. The block expands any scalar inputs to have the same dimensions as the nonscalar inputs. The block outputs a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

The MinMax block ignores any input value that is `NaN`, except when every input value is `NaN`. When all input values are `NaN`, the output is `NaN`, either as a scalar or the value of each output vector element.

Data Type Support

The MinMax block accepts and outputs real signals of the following data types:

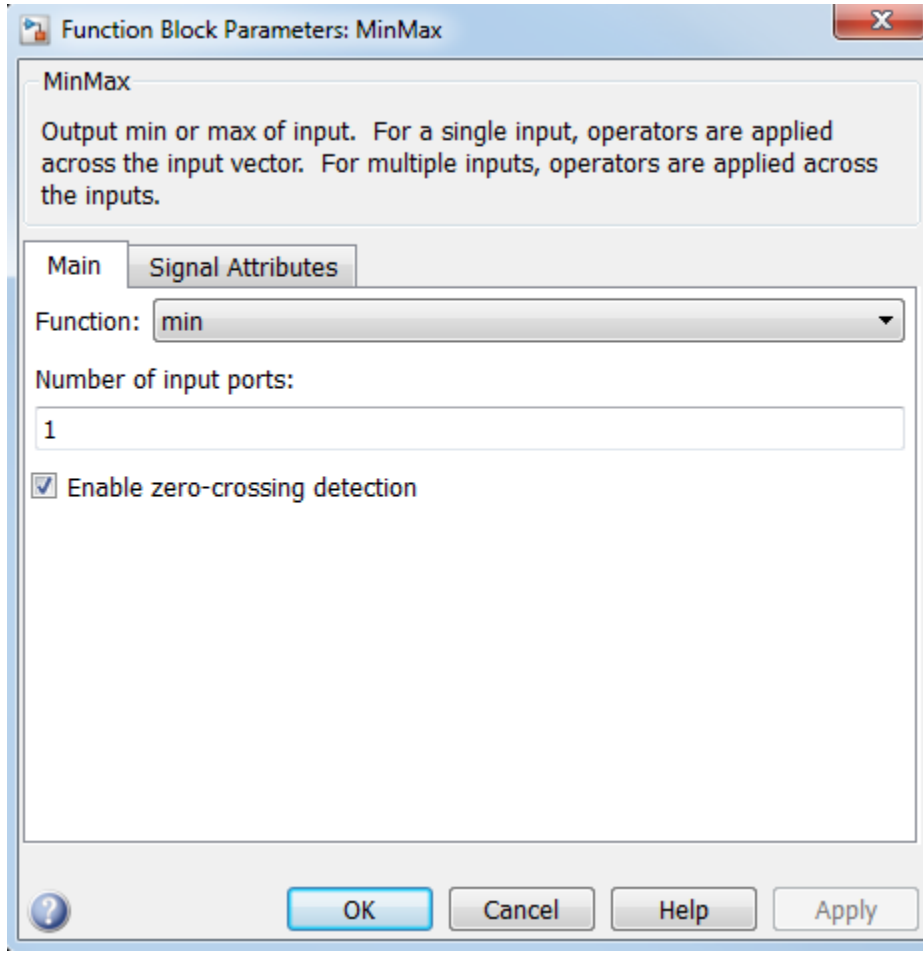
- Floating point
- Built-in integer

- 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

Specify whether to apply the function `min` or `max` to the input.

Number of input ports

Specify the number of inputs to the block.

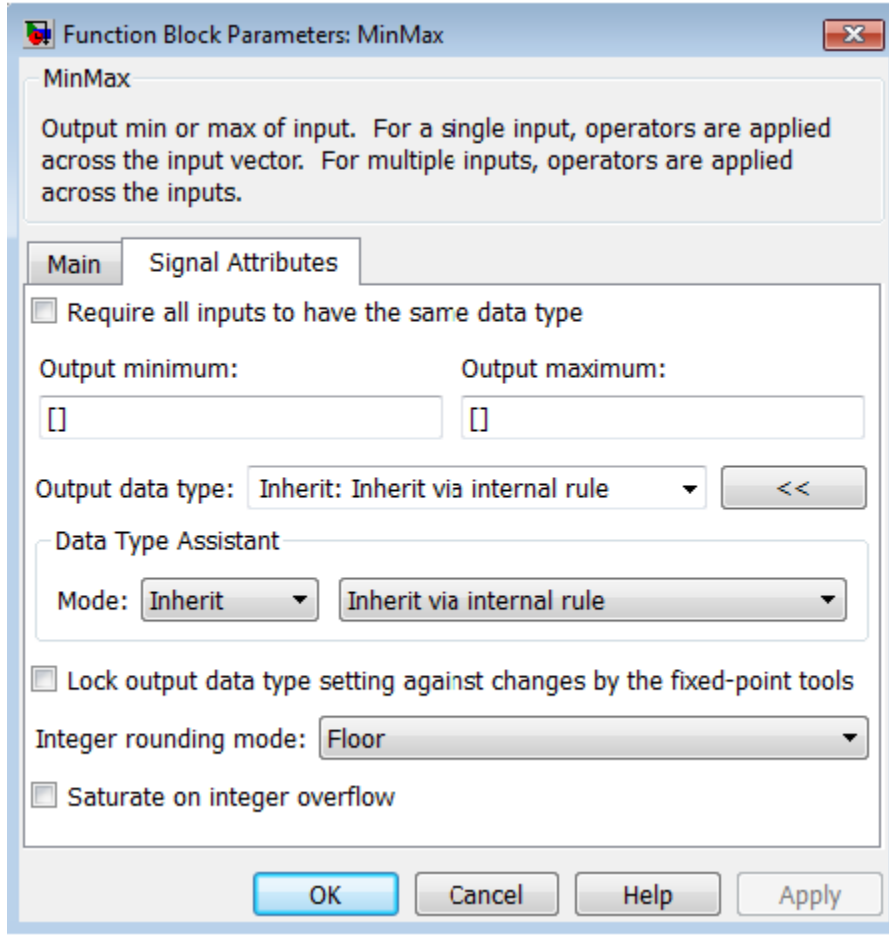
Enable zero-crossing detection

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

Sample time

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

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



Require all inputs to have the same data type

Select this check box to require that all inputs have the same data type.

Output minimum

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

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

Output maximum

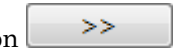
Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- 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 "Control Signal Data Types" in the Simulink User's Guide for more information.

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

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

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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127.

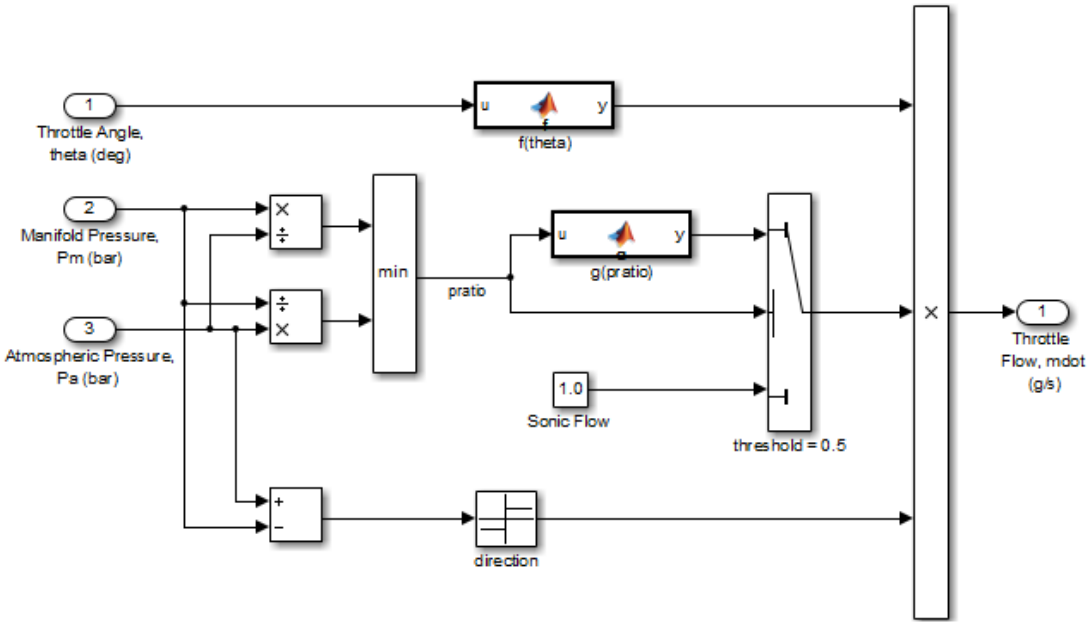
Action	Reasons for Taking This Action	What Happens for Overflows	Example
			Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

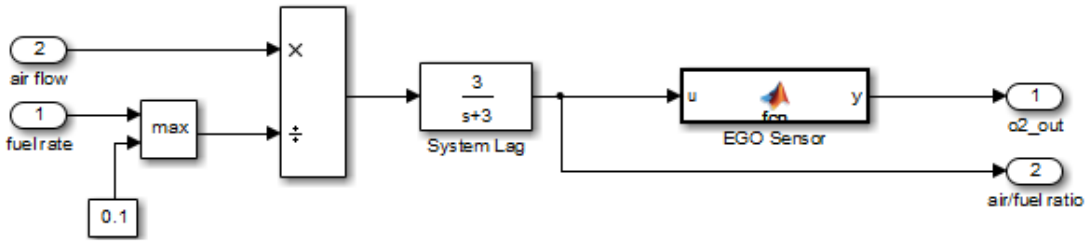
Examples

The `sldemo_fuelsys` model shows how to use the `MinMax` block.

In the `Engine Gas Dynamics/Throttle & Manifold/Throttle` subsystem, the `MinMax` block uses the `min` operator:



In the `Engine Gas Dynamics/Mixing & Combustion` subsystem, the `MinMax` block uses the `max` operator:



Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point

Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

`MinMax Running Resettable`

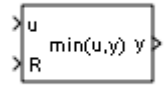
Introduced before R2006a

MinMax Running Resettable

Determine minimum or maximum of signal over time

Library

Math Operations



Description

The MinMax Running Resettable block outputs the minimum or maximum of all past inputs u. You specify whether the block outputs the minimum or the maximum with the **Function** parameter.

The block can reset its state based on an external reset signal R. When the reset signal R is TRUE, the block resets the output to the value of the **Initial condition** parameter.

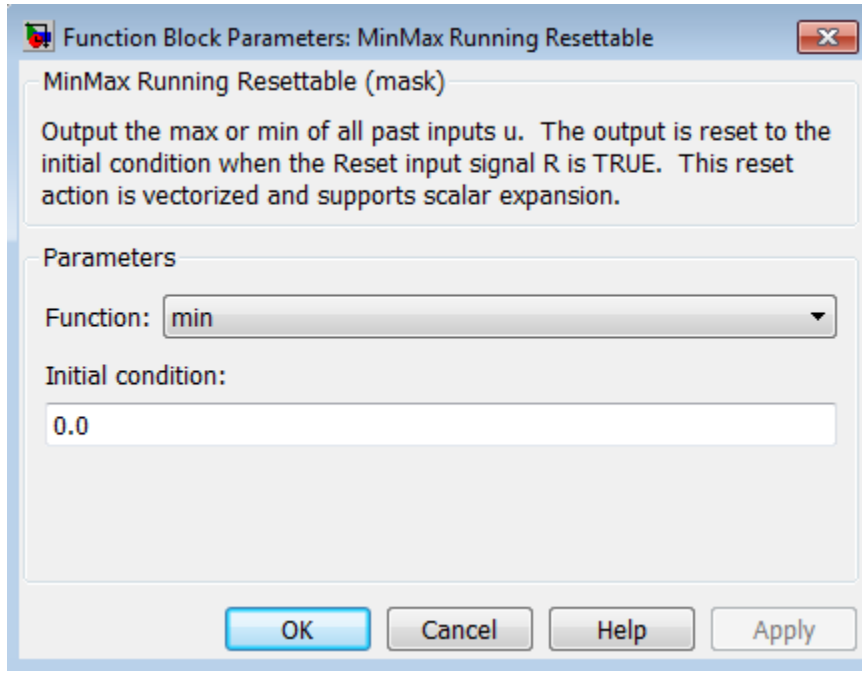
The input can be a scalar, vector, or matrix signal. If you specify a scalar **Initial condition** parameter, the block expands the parameter to have the same dimensions as a nonscalar input. The block outputs a signal having the same dimensions as the input. Each output element equals the running minimum or maximum of the corresponding input elements.

Data Type Support

The MinMax Running Resettable block accepts and outputs real signals of any numeric data type that Simulink supports, except `Boolean`. The MinMax Running Resettable block supports fixed-point data types.

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

Parameters and Dialog Box



Function

Specify whether the block outputs the minimum or the maximum.

Initial condition

Specify initial condition.

Characteristics

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

See Also

`MinMax`

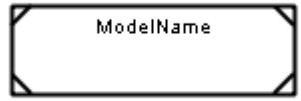
Introduced before R2006a

Model, Model Variants

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`.

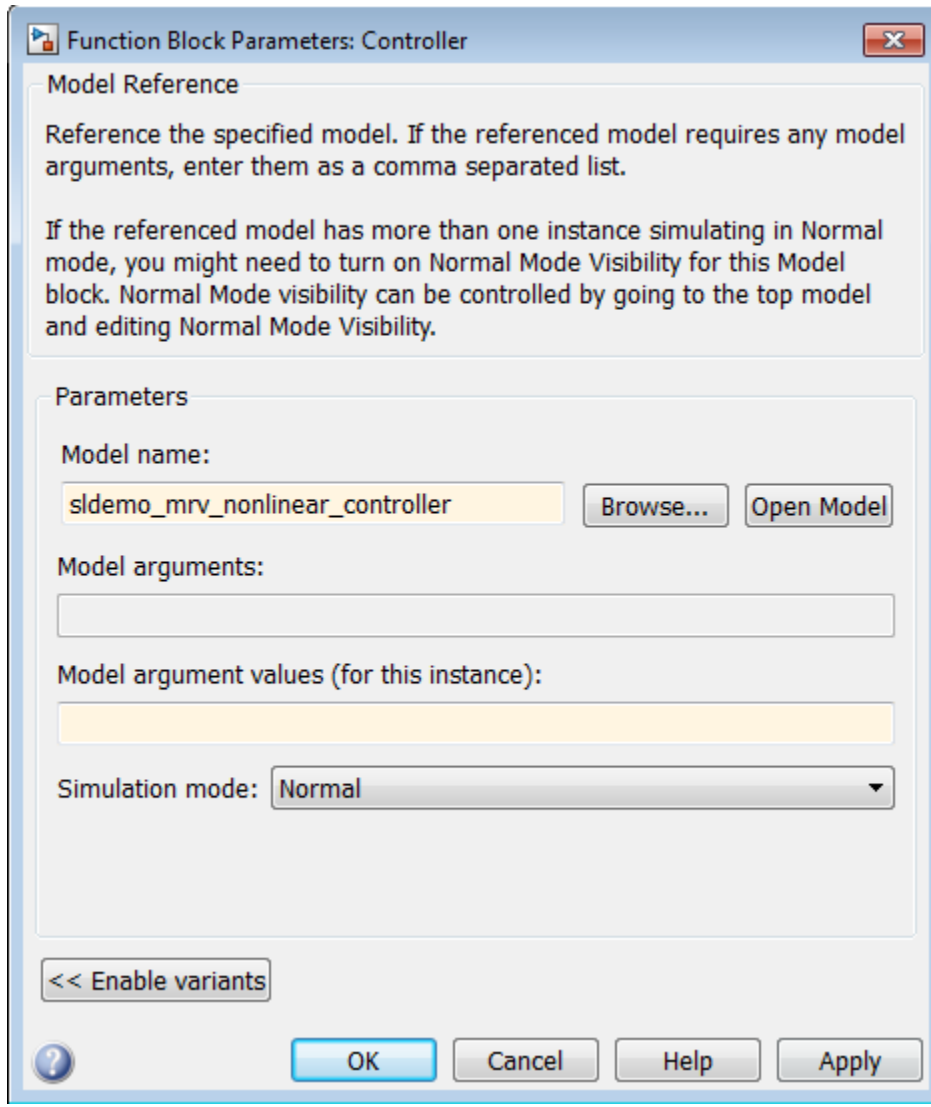
The Model block supports signal label propagation. For details specific to model referencing and model variants, see:

- "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.

Parameters and Dialog Box



- "Model name" on page 1-1147
- "Model arguments" on page 1-1148

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 “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.

You can enter the values as literal values, variable names, MATLAB expressions, and Simulink parameter objects. Any symbols used resolve to values as described in "Symbol Resolution Process". All values must be numeric (including objects with numeric values).

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.

Note: If the model referenced by this block is a variant choice, setting the simulation mode of the top model does not change the simulation mode of the variant model. Specify the simulation mode of each variant choice using the **Simulation mode** dropdown in the parameters dialog box for the `Model` block.

Command-Line Information

Parameter: `SimulationMode`
Type: string
Value: `'Accelerator' | 'Normal' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'`
Default: `'Accelerator'`

See Also

- “Model Arguments”
- “Choosing a Simulation Mode”
- “Host-Target Communication for PIL”
- “Numerical Equivalence Testing”

Code interface

Specify whether the generated code is from top model or referenced model.

Settings

Default: `Model reference`

`Model reference`

Code generated from referenced model as part of a model reference hierarchy. Code generation uses the `slbuild('model', 'ModelReferenceRTWTarget')` command.

`Top model`

Code generated from top model with the standalone code interface. Code generation uses the `slbuild('model')` command.

Dependency

Setting the **Simulation mode** parameter to `Software-in-the-loop (SIL)` or `Processor-in-the-loop (PIL)` enables this parameter.

Command-Line Information

Parameter: `CodeInterface`
Type: string
Value: `'Model reference' | 'Top model'`
Default: `'Model reference'`

See Also

- "Software-in-the-Loop (SIL) Simulation"
- "Processor-in-the-Loop (PIL) Simulation"
- `slbuild`

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.

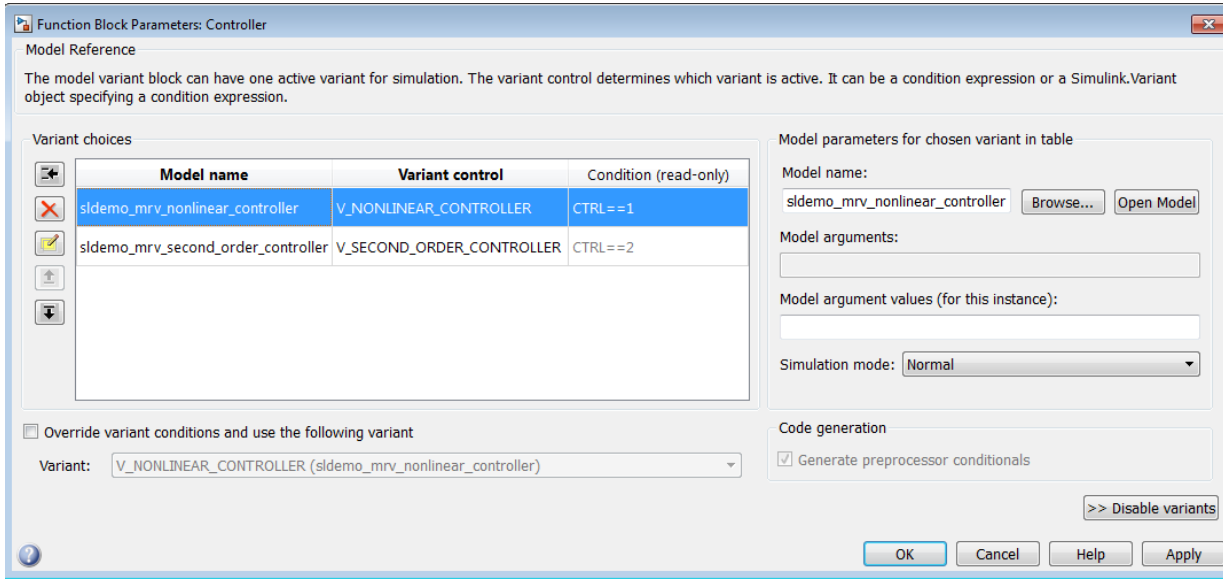
Settings

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`.



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.	
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: `''`

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 **Browse**.
- To confirm your selection, click **Open Model**.

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 box, 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 "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

- "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

- “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

- "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

"Create, Export, and Reuse Variant Controls"

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

- "Create, Export, and Reuse Variant Controls"
- `Simulink.Variant`

Variant control

Enter the variant activation condition or the variant control that contains the expression for variant activation.

The variant control can be a boolean condition expression or a `Simulink.Variant` object representing a boolean condition expression. If you want to generate code for your model, define control variables as `Simulink.Parameter` objects.

Settings

Default: `Variant`

Dependency

Adding a `Model` block inside a `Variant Subsystem` block 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`

Generate preprocessor conditionals

When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`).

When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

Settings

Default: Disabled

Dependencies

- The check box is available for generating only ERT targets.
- **Override variant conditions and use following variant** is cleared (`'off'`)

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. See:

- "Algebraic Loops" for information about direct feed through and algebraic loops.
- "Highlight Algebraic Loops in the Model" 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"
  - "Remove 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

Generic Real Time (`grt`) and Embedded Real Time (`ert`) based targets provide the option **Model Configuration Parameters > All Parameters > 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"
  - "Remove Algebraic Loops"
- Manually insert one or more Unit Delay blocks as needed to break the algebraic loop.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Direct Feedthrough	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.
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	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 “Represent Subsystem and Model Variants in Generated Code”.

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"

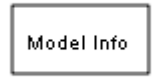
Introduced before R2006a

Model Info

Display model properties and text in model

Library

Model-Wide Utilities



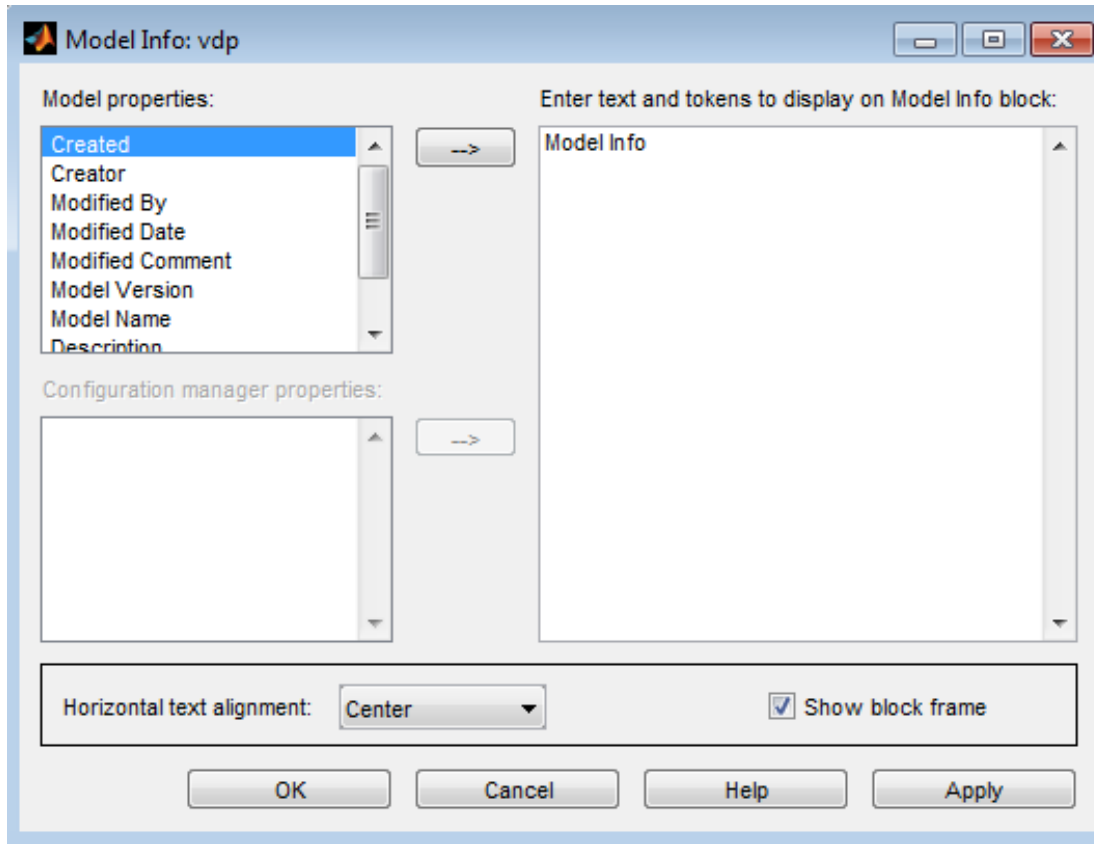
Description

The Model Info block displays model properties and text about a model on the mask of the block. Use the Model Info block dialog box to specify the content and format of the text that the block displays. You can select model properties to display on the block. In the text displayed on the block mask, Simulink replaces the property name with the current value of the property in the model.

Data Type Support

Not applicable.

Parameters 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 Simulink Project instead of the Model Info block. See “Source Control in Simulink Project”.

Characteristics

Data Types	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

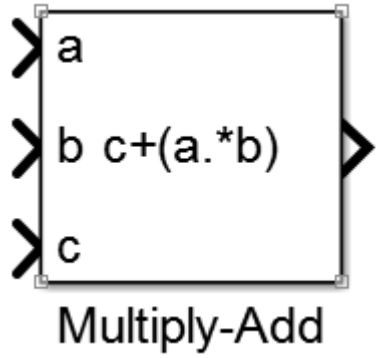
Introduced before R2006a

Multiply-Add

Multiply-add combined operation

Library

HDL Coder / HDL Operations



Description

The `Multiply-Add` block computes the product of the first two inputs, a and b, and adds the result to the third input, c. The inputs can be vectors or scalars.

Operation Precision

The multiplication operation is full precision, regardless of the output type. The **Integer rounding mode** and **Saturate on integer overflow** settings apply only to the addition operation.

HDL Code Generation

Use the `Multiply-Add` block to map a combined multiply-add or a multiply-subtract operation to a DSP unit in your target hardware.

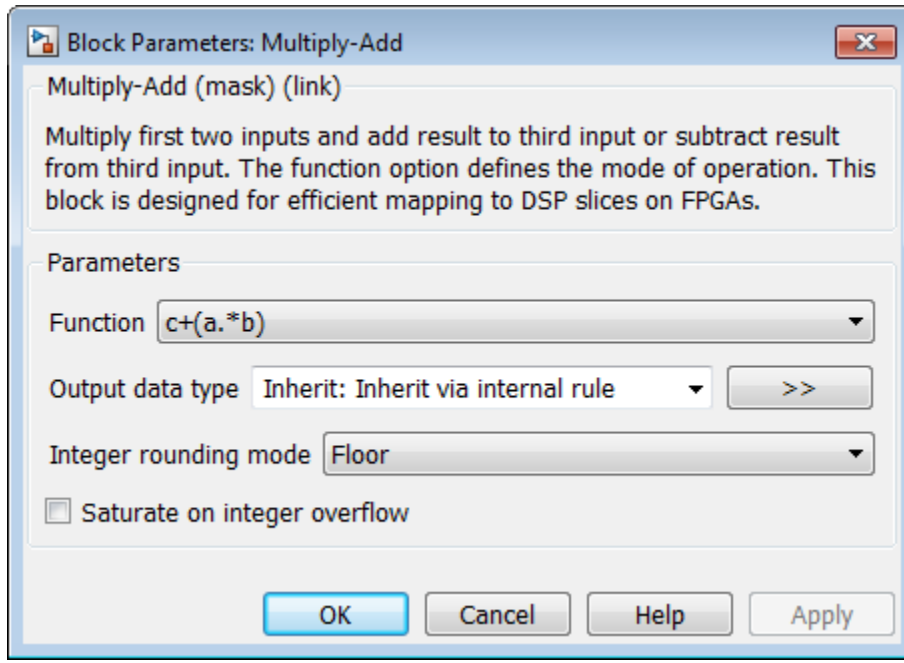
When you generate HDL code for your model, HDL Coder configures the multiply-add operation so that your synthesis tool can map to a DSP unit.

Data Type Support

The `Multiply-Add` block accepts and outputs signals of any numeric data type that Simulink supports, including fixed-point data types.

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

Dialog Box and Parameters



Function

Specify the function to perform a combined multiply and add or a multiply and subtract operation.

Settings

Default: `c+(a.*b)`

You can set the function to:

- `c+(a.*b)`
- `c-(a.*b)`
- `(a.*b)-c`

Output data type

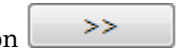
Specify the output data type.

Settings

Default: `Inherit: Inherit via internal rule`

Set the output data type to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button 

to display the Data Type Assistant dialog box, which helps you to set the **Output data type** parameter.

For more information, see “Control Signal Data Types” in Simulink User's Guide .

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

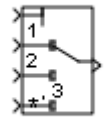
Introduced in R2015b

Multiport Switch

Choose between multiple block inputs

Library

Signal Routing



Description

Note: A variant of the Multiport Switch block is the Index Vector block. For information on the Index Vector block, see "Multiport Switch Configured as an Index Vector Block" on page 1-1181.

The Multiport Switch block determines which of several inputs to the block passes to the output. The block bases this decision on the value of the first input. The first input is the control input and the remaining inputs are the data inputs. The value of the control input determines which data input passes to the output.

The table summarizes how the block interprets the control input and determines the data input that is passed to the output.

Control Input	Truncation	Setting for Data Port Order	Block Behavior During Simulation	
			Indexing to select data input	Out-of-range condition
Integer value	None	`Zero-based contiguous`	Zero-based indexing	The control input is less than `0` or greater than the number of data inputs minus one.

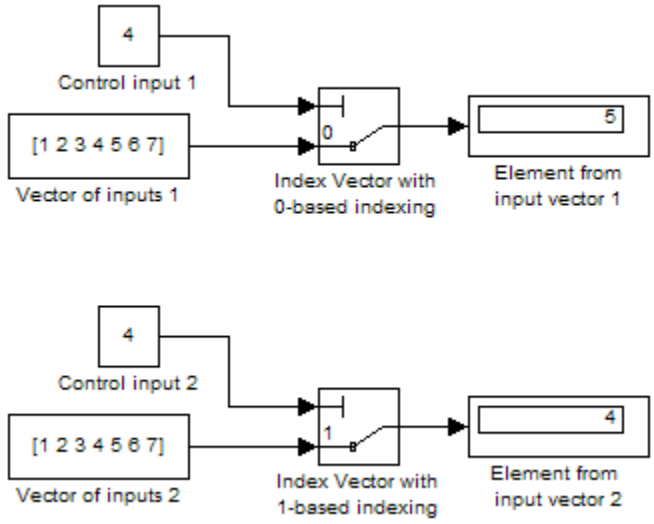
Control Input	Truncation	Setting for Data Port Order	Block Behavior During Simulation	
			Indexing to select data input	Out-of-range condition
		`One-based contiguous`	One-based indexing	The control input is less than `1` or greater than the number of data inputs.
		`Specify indices`	Indices you specify	The control input does not correspond to any specified data port index.
Not an integer value	The block truncates the value to an integer by rounding to zero.	`Zero-based contiguous`	Zero-based indexing	The truncated control input is less than `0` or greater than the number of data inputs minus one.
		`One-based contiguous`	One-based indexing	The truncated control input is less than `1` or greater than the number of data inputs.
		`Specify indices`	Indices you specify	The truncated control input does not correspond to any specified data port index.

For information on how the block handles the out-of-range condition, see "How the Block Handles an Out-of-Range Control Input" on page 1-1182.

Multiport Switch Configured as an Index Vector Block

An `Index Vector` is a special configuration of a Multiport Switch block in which you specify one data input and the control input is zero-based. The block output is the element of the input vector whose index matches the control input. For example, if the input vector is `[18 15 17 10]` and the control input is `3`, the element that matches the index of 3 (zero-based) is 10, and that becomes the output value.

This model shows how the Index Vector works with zero-based and one-based indexing.



Index Vector is a block in the Simulink **Signal Routing** library. Alternatively, configure a Multiport Switch block to work as an Index Vector block by setting **Number of data ports** to `1` and **Data port order** to `Zero-based contiguous`.

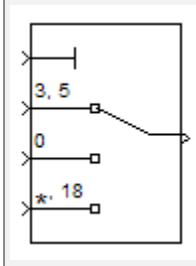
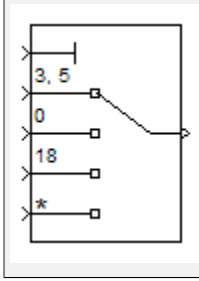
How the Block Handles an Out-of-Range Control Input

For an input with an integer value less than `intmax('int32')`, the input is out of range when the value does not match any data port indices. For a control input that is not an integer value, the input is out of range when the *truncated* value does not match any data port indices. In both cases, the block behavior depends on your settings for **Data port for default case** and **Diagnostic for default case**.

Note: If the control input is larger than `intmax('int32')`, the block wraps the input value to an integer.

Behavior for Simulation

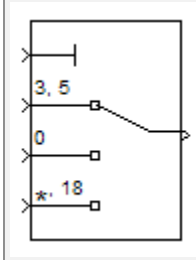
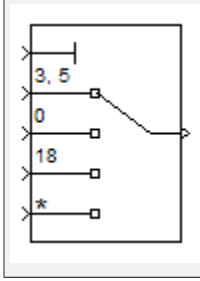
The following behavior applies only to simulation for your model.

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.

Data port for default case	Diagnostic for default case		
	None	Warning	Error
			
`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.

Rules That Determine the Block Behavior

You specify the number of data inputs with **Number of data ports**.

- If you set **Number of data ports** to 1, the block behaves as an *index selector* or *index vector* and not as a multiport switch. For more details, see "Multiport Switch Configured as an Index Vector Block" on page 1-1181.
- If you set **Number of data ports** to an integer greater than 1, the block behaves as a multiport switch. The block output is the data input that corresponds to the value of the control input. If at least one of the data inputs is a vector, the block output is a vector. In this case, the block expands any scalar inputs to vectors.
- If all the data inputs are scalar, the output is a scalar.

Guidelines on Setting Parameters for Enumerated Control Port

When the control port on the `Multiport Switch` block is of enumerated type, follow these guidelines:

Scenario	What to Do	Rationale
The enumerated type contains a value that represents invalid, out-of-range, or uninitialized values.	• Set **Data port order** to `Specify indices`. • Set **Data port indices** to use this value for the last data port. • Set **Data port for default case** to `Last data port`.	This block configuration handles invalid values that the enumerated type explicitly represents.
The enumerated type contains only valid enumerated values. However, a data input port can get invalid values of enumerated type.	• Set **Data port for default case** to `Additional data port`.	This block configuration handles invalid values that the enumerated type does not explicitly represent.
The enumerated type contains only valid enumerated values. Data input ports can never get invalid values of enumerated type.	• Set **Data port for default case** to `Last data port`. • Set **Diagnostic for default case** to `None`.	This block configuration avoids unnecessary diagnostic action.
The block does not have a data input port for every value of the enumerated type.	• Set **Data port for default case** to `Additional data port`.	This block configuration handles enumerated values that do not have a data input port, along with invalid values.

Data Type Support

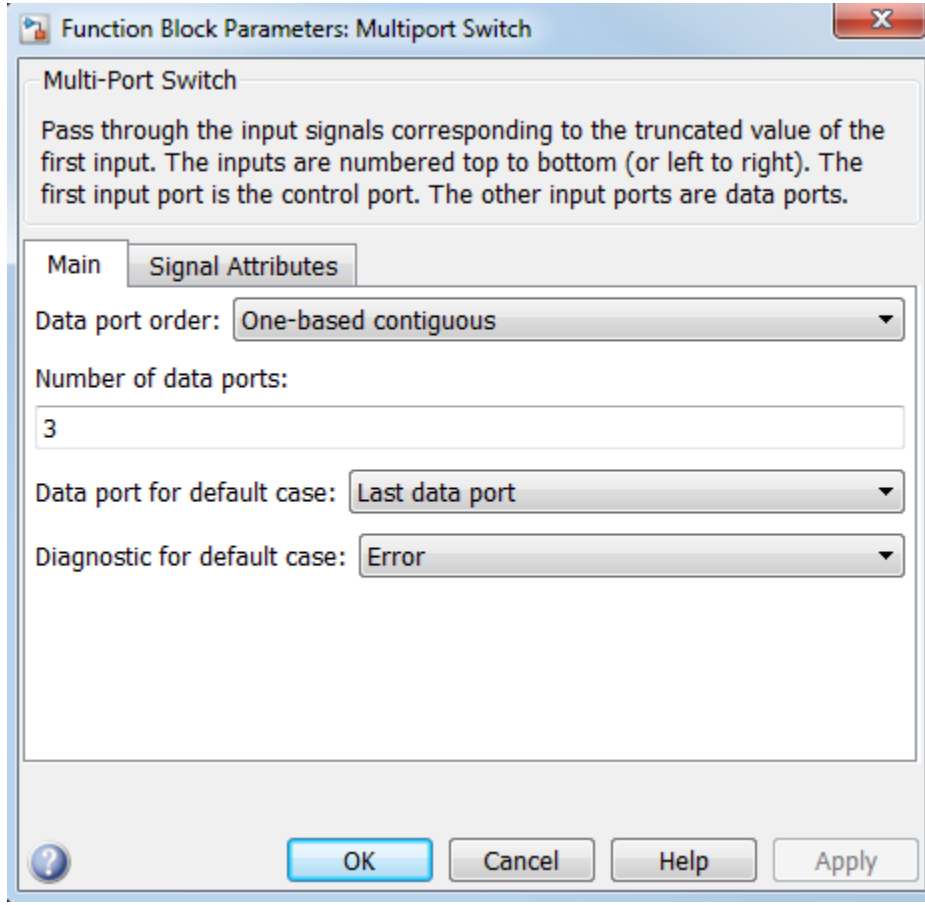
The control signal can be of any data type that Simulink supports, including fixed-point and enumerated types. If the control signal is numeric, it cannot be complex. If the control signal is an enumerated signal, the block uses the value of the underlying integer to select a data port. If the underlying 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.

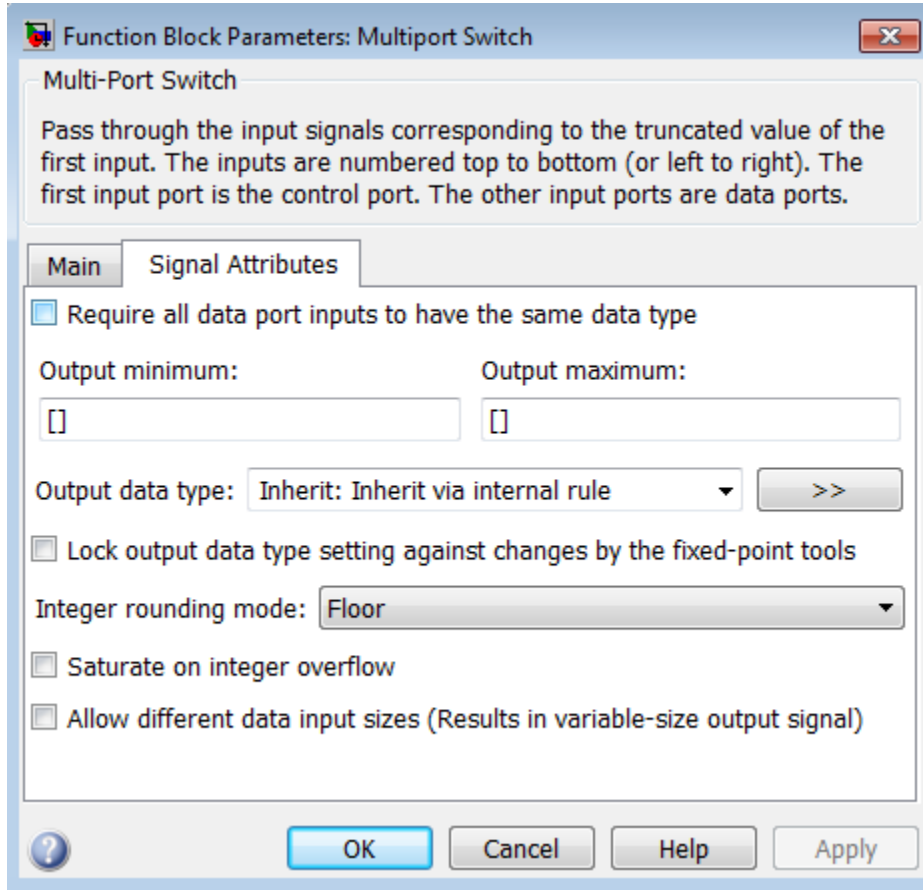
For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box

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



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



- "Data port order" on page 1-1190
- "Number of data ports" on page 1-1192
- "Data port indices" on page 1-1193
- "Data port for default case" on page 1-1195
- "Diagnostic for default case" on page 1-1196
- "Sample time" on page 1-298
- "Require all data port inputs to have the same data type" on page 1-1198
- "Lock output data type setting against changes by the fixed-point tools" on page 1-235
- "Integer rounding mode" on page 1-295

- "Saturate on integer overflow" on page 1-297
- "Allow different data input sizes" on page 1-1203
- "Output minimum" on page 1-1204
- "Output maximum" on page 1-1205
- "Output data type" on page 1-1206
- "Mode" on page 1-1208
- "Data type override" on page 1-230
- "Signedness" on page 1-1211
- "Word length" on page 1-1212
- "Scaling" on page 1-225
- "Fraction length" on page 1-1214
- "Slope" on page 1-1215
- "Bias" on page 1-1216

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 the Upgrade Advisor on your model to replace each `Multiport Switch` block of this configuration with a block that explicitly specifies data port indices. See "Model Upgrades".
- Avoid situations where the block contains unused data ports for simulation or code generation. When the control port is of fixed-point or built-in data type, verify that all data port indices are representable with that type. Otherwise, the following block behavior occurs:

If the block has unused data ports and data port order is...	You get...
`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 6-101 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 6-101 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 6-101 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 6-101 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`

Stop simulation and display an error in the Diagnostic Viewer. 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 1-1182.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Sample time

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

Require all data port inputs to have the same data type

Specify allowed data types.

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 6-101 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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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: `AllowDiffInputSizes`
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 6-101 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 6-101 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 6-101 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 6-101 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 6-101 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 6-101 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**

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 6-101 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 6-101 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 6-101 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.

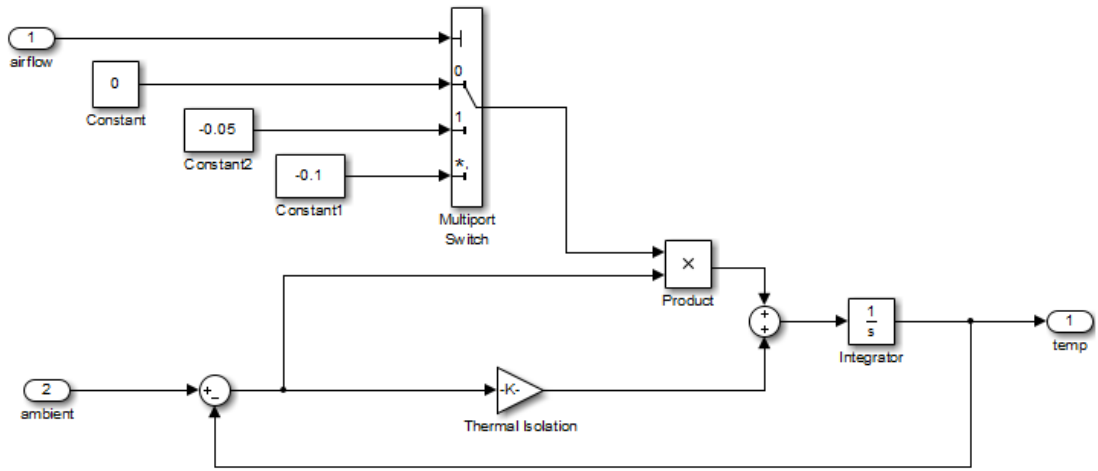
You can use a `Rate Transition` block to change the sample time of an individual signal, or of all signals in a bus. See "Composite Signals" and "Bus-Capable Blocks" for more information.

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.

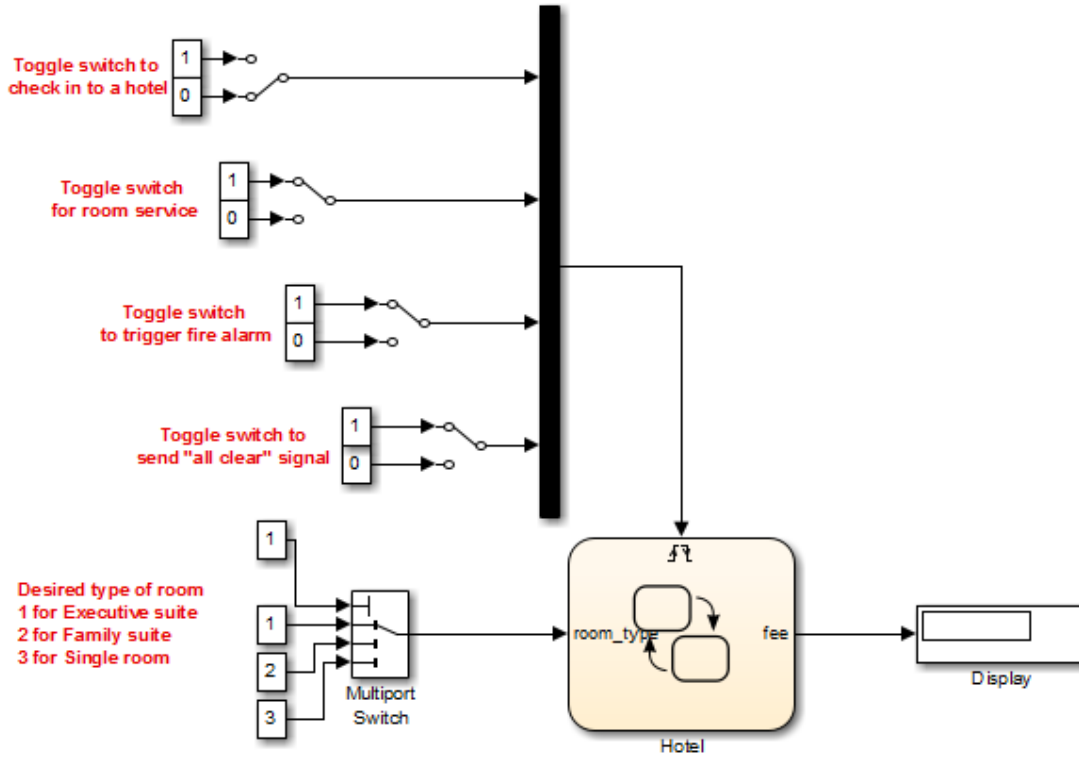


The indices are visible on the data port labels. You do not have to open the block dialog box to determine whether the data ports use zero-based or one-based indexing.

When you set **Data port for default case** to `Last data port`, the last data port includes a `*` on the label. The comma after the `*` indicates that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices 0, 1, or 2. In this case, the Multiport Switch block outputs a value of –0.1.

One-Based Indexing for Data Ports

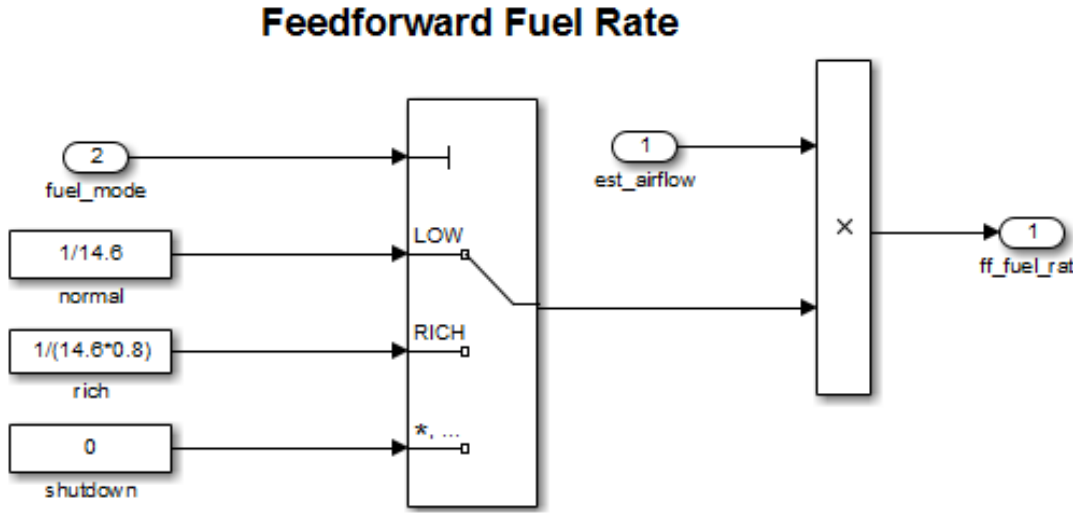
The sf_semantics_hotel_checkin model uses a Multiport Switch block. This block uses one-based indexing for contiguous ordering of three data ports.



If you increase the size of the block icon, the indices are visible on the data port labels. You do not have to open the block dialog box to determine whether the data ports use zero-based or one-based indexing.

Enumerated Names for Data Port Indices

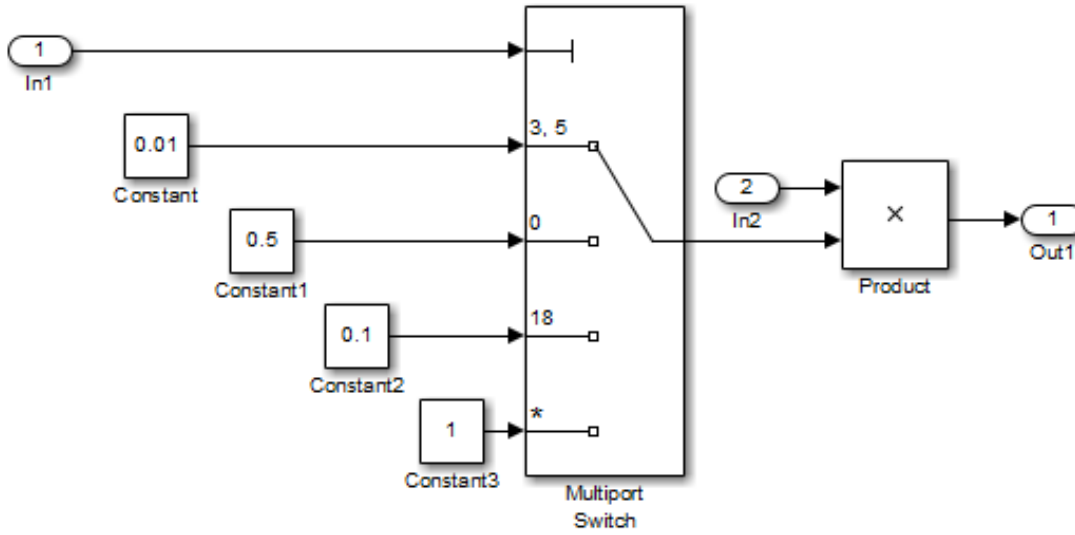
The sldemo_fuelsys model uses a Multiport Switch block in the `fuel_rate_control/fuel_calc/feedforward_fuel_rate` subsystem. This block uses the enumerated type `sld_FuelModes` to specify three data port indices: LOW, RICH, and DISABLED.



When you set **Data port for default case** to `Last data port`, the last data port includes a `*` on the label. The comma and ellipsis after the `*` indicate that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices LOW, RICH, or DISABLED. In this case, the Multiport Switch block outputs a value of 0.

Noncontiguous Values for Data Port Indices

The following model uses a Multiport Switch block that specifies noncontiguous integer values for data ports.



The values of the indices are visible on the data port labels. You do not have to open the block dialog box to determine which value maps to each data port.

When you set **Data port for default case** to `Additional data port`, an extra port with a `*` label appears. This port corresponds to the default case, which applies when the control input does not match the data port indices 3, 5, 0, or 18. In this case, the Multiport Switch block outputs a value of 1.

Characteristics

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

See Also

`Switch`

Introduced before R2006a

Mux

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 “Prevent Bus and Mux 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.

Format	Block Behavior
	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.
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`.

Simulink provides several techniques for combining signals into a composite signal. For a comparison of techniques, see "Techniques for Combining Signals". 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.

Tip

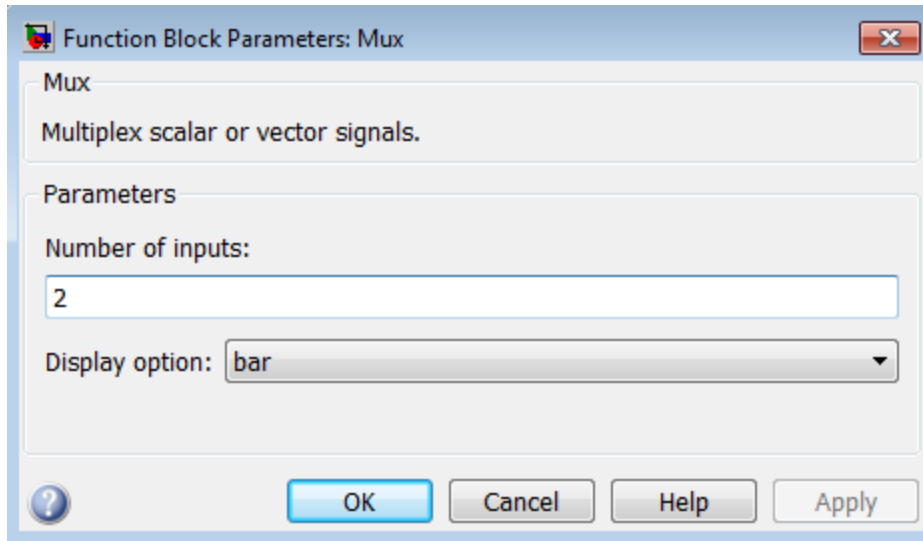
To create a composite signal, in which the constituent signals retain their identities and can have different data types, use a `Bus Creator` block rather than a Mux block. Although you can use a Mux block to create a composite signal, MathWorks discourages this practice. See “Prevent Bus and Mux Mixtures” 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.

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 6-101 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

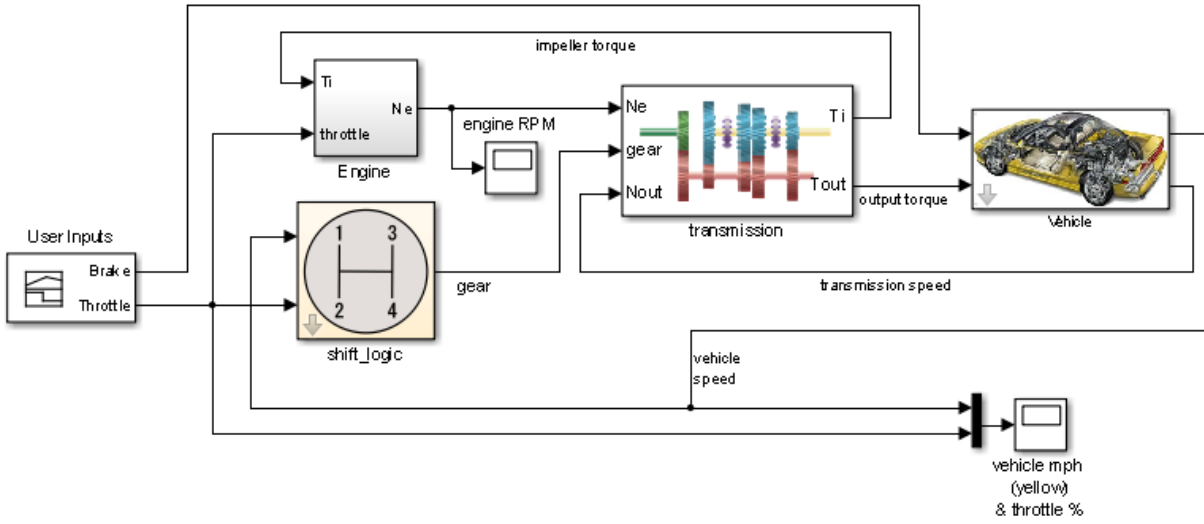
`signals`
: Displays signal names next to each port

Command-Line Information

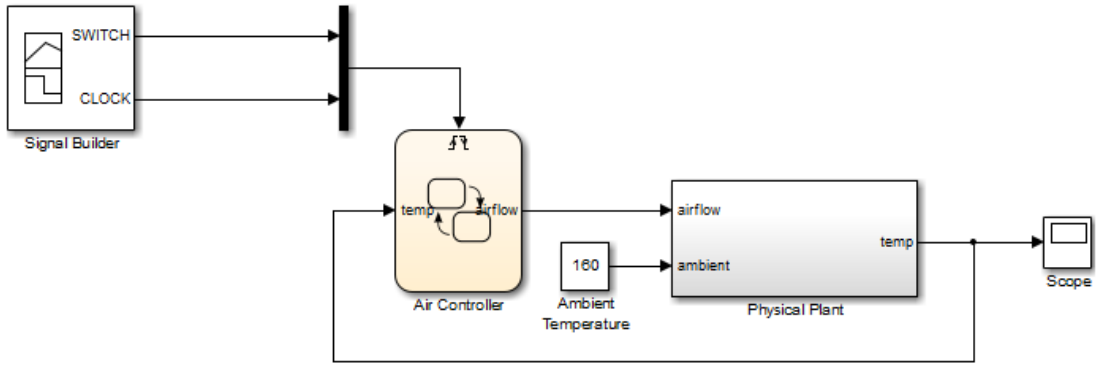
See "Block-Specific Parameters" on page 6-101 for the command-line information.

Examples

The sf_car model uses a `Mux` block to combine two signals for input to a Scope block:



The sf_aircontrol model uses a `Mux` block to combine two signals for input to a Stateflow chart:



The following models also show how to use the `Mux` block:

- `sldemo_auto_climatecontrol`
- `sldemo_suspn`
- `sldemo_zeroxing`
- `penddemo`

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

`Demux`

Introduced before R2006a

Outport

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 root-level system or subsystem sequentially, starting with 1.
- If you add an Outport block, it is assigned the next available number.
- If you delete an Outport block, other port numbers are automatically renumbered to ensure that the Outport blocks are in sequence and that no numbers are omitted.

Outport Blocks in a Subsystem

Outport blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Outport block in a subsystem flows out of the associated output port on that Subsystem block. The Outport block associated with an output port 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**.

Initializing Outport Blocks in Conditionally Executed Contexts

To set initial conditions for an Outport block in a conditionally executed subsystem, use one of these approaches.

- Inherit initial values from input signals for the subsystem.
- Explicitly specify initial values

For details, see "Specify or Inherit Conditional Subsystem Initial Values".

Note: If the conditional subsystem is driving a Merge block in the same model, you do not need to specify an Initial Condition (IC) for the subsystem's `Outport` block. For more information, see "Underspecified initialization detection" .

Top-level Outport Block in a Model Hierarchy

Outport blocks at the top-level of a model hierarchy have two uses: to supply external outputs to the base MATLAB workspace, which you can do by using either **Configuration Parameters** pane parameters or the `sim` command, and to provide a means for analysis functions to obtain output from the system.

- To supply external outputs to the workspace, use the **Configuration Parameters > Data Import/Export** pane (see Exporting Output Data to the MATLAB Workspace) or the `sim` command (see). 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

If you do not specify a bus object for 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`.

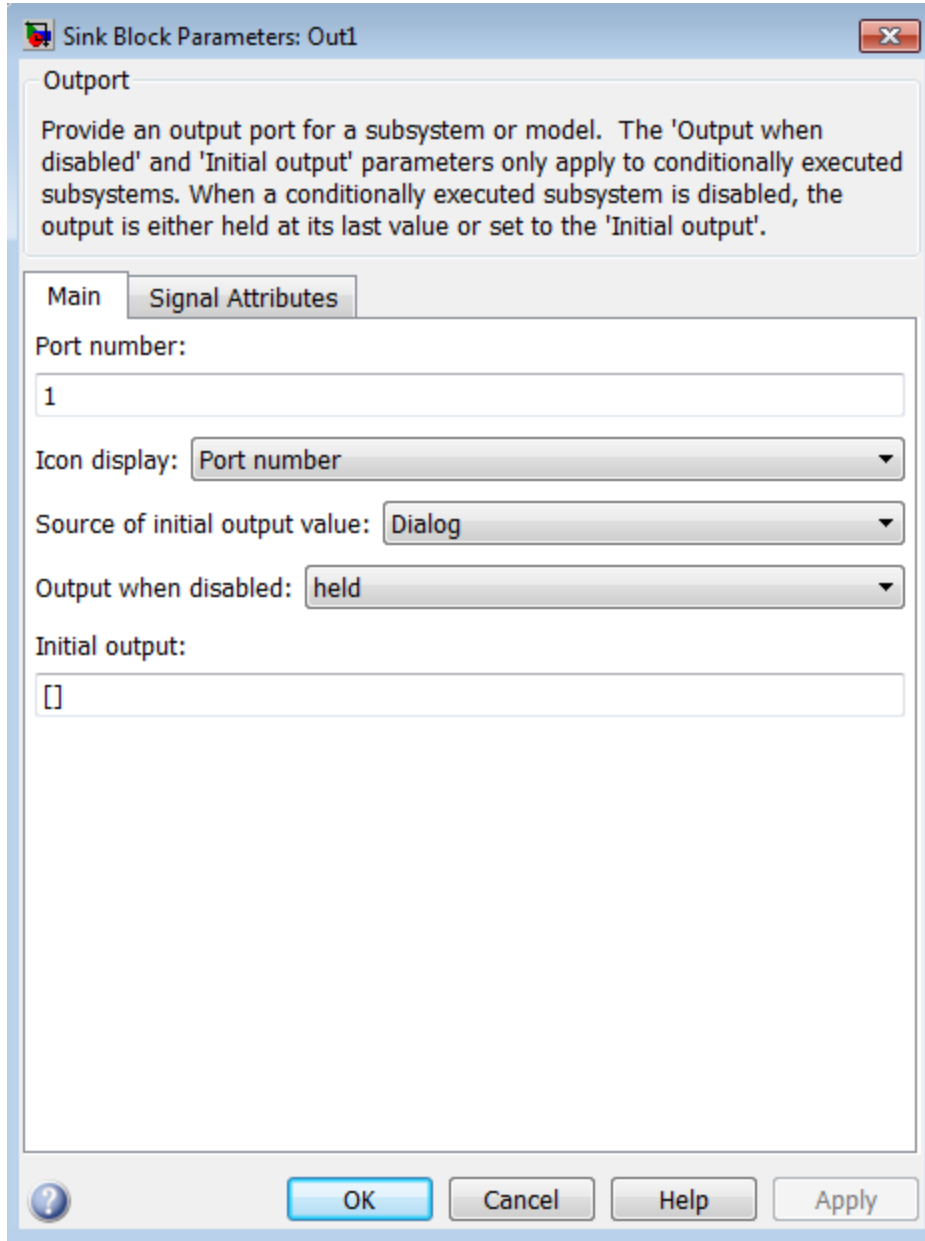
For more information, see "Data Types Supported by Simulink".

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, when present in a conditionally executed subsystem, 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'.
Main
Signal Attributes
Port number:
1
Icon display:
Port number
Source of initial output value:
Dialog
Output when disabled:
held
Initial output:
[]
OK
Cancel
Help
Apply

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 6-101 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 6-101 for the command-line information.

Source of initial output value

Select the source of the initial output value of the 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. See "Specify or Inherit Conditional Subsystem Initial Values".

Tips

- If you are using classic initialization mode, selecting `Input signal` will cause an error. To inherit the initial output value from the input signal, set this parameter to `Dialog` and specify `[]` (empty matrix) for the **Initial output** value. For more information, see "Specify or Inherit Conditional Subsystem Initial Values".

Dependencies

This parameter is enabled when the Outport resides in an Conditional Subsystem.

Selecting `Dialog` enables the following parameters:

- **Output when disabled**
- **Initial output**

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 to the value given by **Initial output** when the subsystem is disabled.

Tips

- When connecting the output of a conditional subsystem to a `Merge` block, set this parameter to `held`. Setting it to `reset` will return an error.

Dependencies

- Selecting `Dialog` in **Source of initial output value** enables this parameter.
- This parameter is enabled when the Outport resides in a conditional subsystem with valid enabling and disabling semantics. For example, this parameter is disabled when the Outport is placed inside a Triggered Subsystem but is enabled when the Outport is placed inside an Enabled Subsystem.
- If an Outport is placed inside a function-call subsystem, this parameter is meaningful only if the function-call subsystem is bound to a state in a Stateflow chart. For more information, see "Bind a Function-Call Subsystem to a State".

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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

- Specify `[ ]` (empty matrix) to inherit the initial output value from the input signal. For more information, see "Specify or Inherit Conditional Subsystem Initial Values".
- 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 is enabled when the Outport resides in an Conditional Subsystem.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 6-101 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 6-101 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 6-101 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 “Control Signal Data Types”.

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 6-101 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.

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.

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**

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.

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.

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.

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

- In a nonvirtual bus, all signals must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. For details, see "Connect Multi-Rate Buses to Referenced Models".
- For a virtual bus, to use a multirate signal, in the root-level Outport block, set the **Sample time** parameter to inherited (`-1`).
- For the top model in a model reference hierarchy, code generation creates a C structure to represent the bus signal output by this block.
- For referenced models, select this option to create a C structure. Otherwise, code generation creates an argument for each leaf element of the bus.

Dependency

Selecting **Data type** > `Bus: <object name>` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Unit (e.g., m, m/s^2, N*m)

Specify physical unit of the input signal to the block.

Settings

Default: `inherit`

To specify a unit, begin typing in the text box. As you type, the parameter displays potential unit string matches. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

- If a `Unit System Configuration` block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a `Unit System Configuration` block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 6-101 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 6-101 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 6-101 for the command-line information.

Characteristics

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

See Also

`Inport`

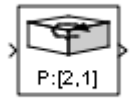
Introduced before R2006a

Permute Dimensions

Rearrange dimensions of multidimensional array dimensions

Library

Math Operations



Description

The block reorders the elements of the input signal so that they are in the order you specify in the **Order** parameter.

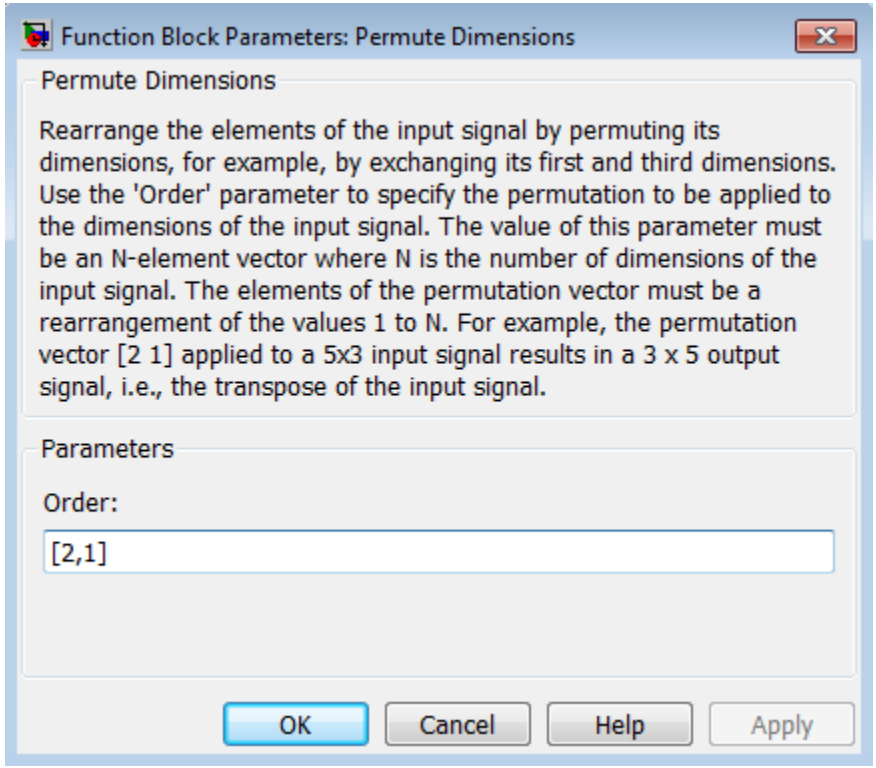
Data Type Support

This block accepts signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types. Output must be the same data type as the input.

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

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

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Inherited from driving block

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

See Also

`Math Function` (`transpose`), `permute` (in the MATLAB reference documentation)

Introduced in R2007a

PID ControllerDiscrete PID Controller

Simulate continuous- or discrete-time PID controllers

Library

Continuous, Discrete

Description

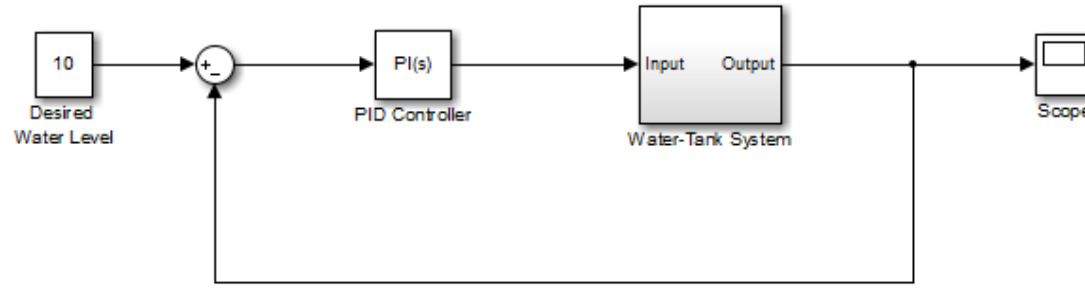
Implement a continuous- or discrete-time controller (PID, PI, PD, P, or I) in your Simulink model. PID controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design™ software (PID Tuner or SISO Design Tool).

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**

Task	Parameters
Choose discrete or continuous time.	• **Time-domain** • **Sample time**
Choose an integration method (discrete time).	• **Integrator method** • **Filter method**
Set and tune controller gains.	• **Controller Parameters Source** in Main tab • **Proportional (P)** in Main tab • **Integral (I)** in Main tab • **Derivative (D)** in Main tab • **Filter coefficient (N)** in Main tab • **Use filtered derivative** 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:

$$C_{par}(s) = \left[P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right) \right]$$

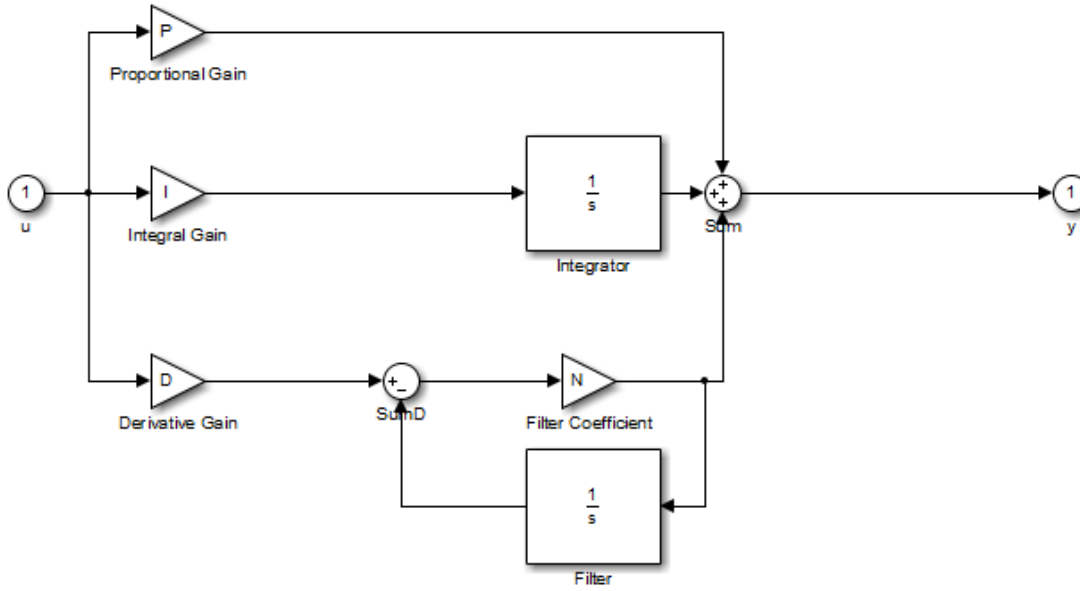
For a discrete-time parallel PID controller, the transfer function takes the form:

$$C_{par}(z) = P + Ia(z) + D\left[\frac{N}{1+Nb(z)}\right]$$

where the **Integrator method** determines `a(z)` and the **Filter method** determines `b(z)` (for sampling time T_s):

	Forward Euler method	**Backward Euler method**	**Trapezoidal method**
$a(z)$ (determined by Integrator method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2}\frac{z+1}{z-1}$
$b(z)$ (determined by Filter method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2}\frac{z+1}{z-1}$

The controller transfer function for the current settings is displayed in the block dialog box.



Parallel PID Controller

`Ideal`

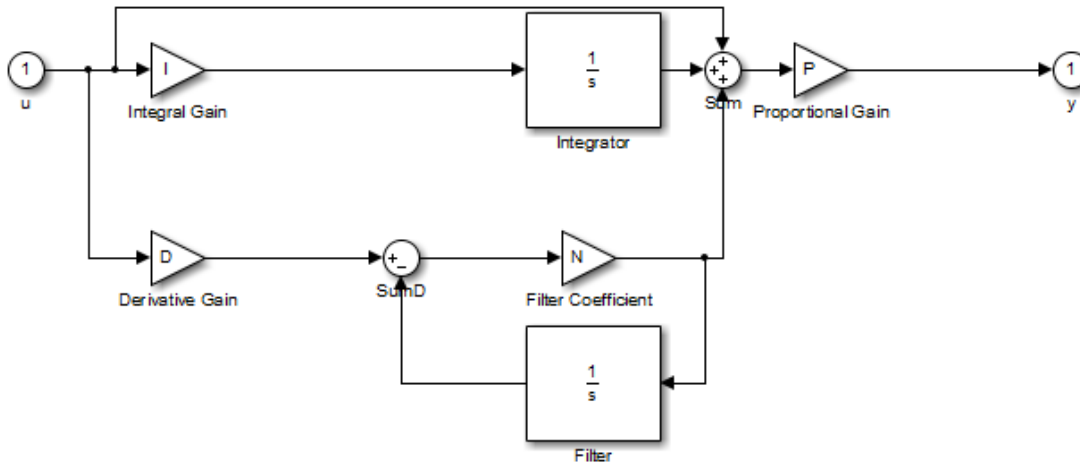
Selects a controller form in which the proportional gain `P` acts on the sum of all actions. The transfer functions are the same as for the parallel form, except that `P` multiplies all terms. For a continuous-time ideal PID controller, the transfer function is:

$$C_{id}(s) = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right)\right]$$

For a discrete-time ideal PID controller the transfer function is:

$$C_{id}(z) = P\left[1 + Ia(z) + D\frac{N}{1+Nb(z)}\right]$$

where the **Integrator method** determines `a(z)` and the **Filter method** determines `b(z)` as described previously for the parallel controller form.



Ideal PID Controller

Controller

Specify the controller type.

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.

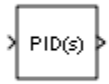
`I`

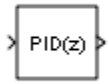
Implements a controller with integral action.

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.

When the `PID Controller` block is in a model with synchronous state control (see the `State Control` block), you cannot select `Continuous-time`.

`Discrete-time`

Selects the discrete-time representation. Selecting `Discrete-time` also allows you to specify the:

- **Sample time**, which is the discrete interval between samples.
- Discrete integration methods for the integrator and the derivative filter using the **Integrator method** and **Filter method** menus.

Integrator method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the integrator output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

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.

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.

If you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

`Trapezoidal`

Selects the Bilinear approximation.

An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

If you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a

model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

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.

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.

This filter method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

`Trapezoidal`

Selects the Bilinear approximation.

An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method. Of all available filter methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

This filter method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code

using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

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.

Controller Parameters Source

Select the source of the controller gains and filter coefficient. You can provide these parameters explicitly in the block dialog box, or enable external inputs for them on the block. Enabling external inputs for the parameters allows you to compute PID gains and filter coefficients externally to the block and provide them to the block as signal inputs.

External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control, in which controller gains are determined by logic or other calculation in the Simulink model and passed to the block.

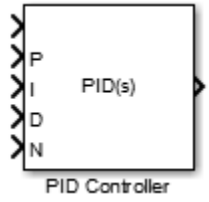
Settings

`internal` (Default)

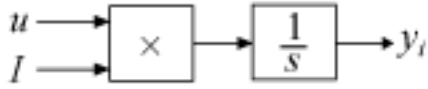
Specify the PID gains and filter coefficient explicitly using the **P**, **I**, **D**, and **N** parameters.

`external`

Specify the PID gains and filter coefficient externally. An additional input port appears under the block input for each parameter that is required for the current controller type:



When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because of the way the PID gains are implemented within the block. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.



Within the block, the block's input signal is multiplied by the externally-supplied integrator gain, *I*, before integration. This implementation yields:

$$y_i = \int u I \, dt.$$

Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain *D* to be differentiated.

Proportional (P)

(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 1-1270 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 Control 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 Control Design Approach”.

Derivative (D)

(Available for PID and PD controllers.) Specify the derivative gain `D`.

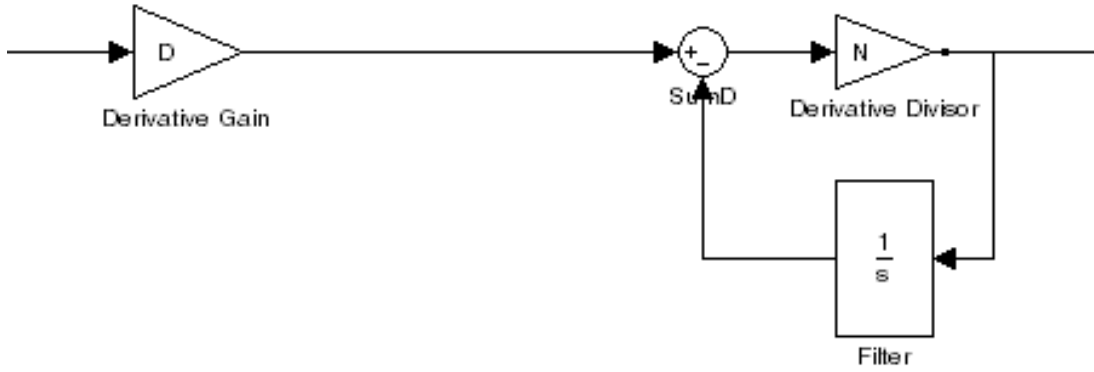
Default: 0

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 Control Design Approach”.

Filter coefficient (N)

(Available for PID and PD controllers, when **Use filtered derivative** is checked.) Specify the filter coefficient `N`, which determines the pole location of the filter in the derivative action:



The filter pole falls at `s = -N` in the `Continuous-time` **Time-domain**. For `Discrete-time`, the location of the pole depends on which **Filter method** you select (for sampling time T_s):

- `Forward Euler`:

  $$z_{pole} = 1 - NT_s$$

- `Backward Euler`:

  $$z_{pole} = \frac{1}{1 + NT_s}$$

- `Trapezoidal`:

  $$z_{pole} = \frac{1 - NT_s / 2}{1 + NT_s / 2}$$

Default: 100.

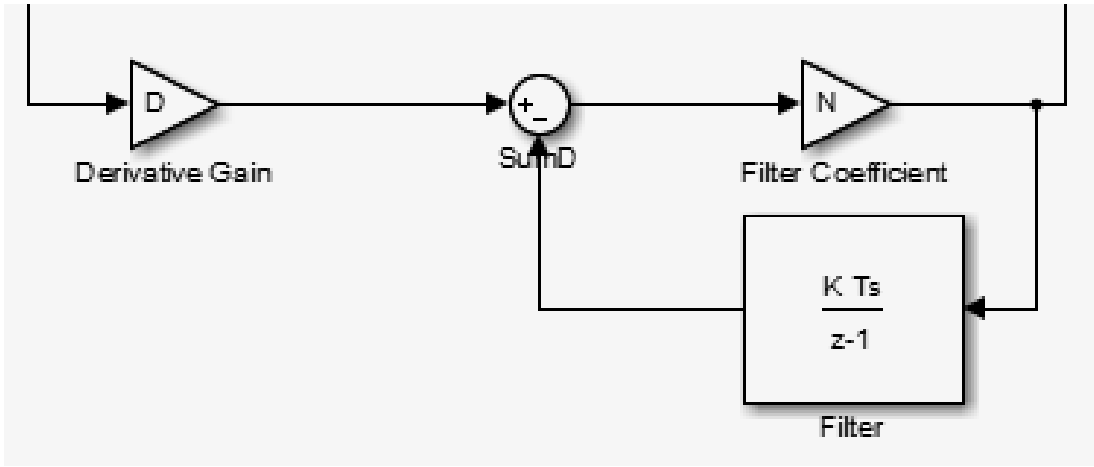
Enter a finite, real gain value into the **Filter Coefficient (N)** field. Use either scalar or vector gain values. Note that the `PID controller` block does not support `N = inf` (ideal unfiltered derivative).

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 Control Design Approach". Automatic tuning requires `N > 0`.

Use Filtered Derivative

Specify whether derivative term is filtered (finite `N`) or unfiltered. Unfiltered derivative is available only for discrete-time controllers.

Unchecking this option replaces the filtered derivative with a discrete differentiator. For example, if **Filter Method** is `Forward Euler`, then the filtered derivative term is represented by:



When you uncheck **Use filtered derivative**, the derivative term becomes:

Settings

☑ On (Default)

Use derivative filter (finite N).

`Off`

Derivative is unfiltered.

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 1-1292). The integrator and filter initial conditions in turn determine the initial block output.

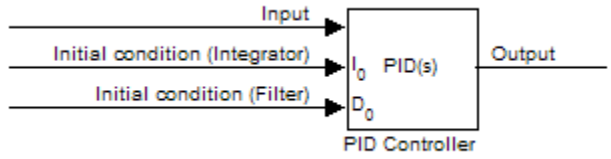
Settings

`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: I_0 for the integrator and D_0 for the filter:



Integrator Initial condition

(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the integrator initial value. Simulink uses the initial condition to initialize the integrator output at the start of a simulation or at a specified trigger event (see "External reset" on page 1-1292). The integrator initial condition, together with the filter initial condition, determines the initial output of the `PID controller` block.

Default: 0

Simulink does not permit the integrator initial condition to be `inf` or `NaN`.

Filter Initial condition

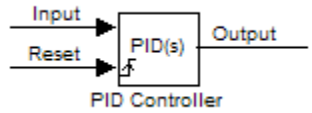
(Available only when **Initial conditions Source** is `internal`, the controller includes derivative action, and **Use filtered derivative** is checked.) Specify the filter initial value. Simulink uses the initial condition to initialize the filter output at the start of a simulation or at a specified trigger event (see "External reset" on page 1-1292). The filter initial condition, together with the integrator initial condition, determines the initial output of the `PID controller` block.

Default: 0

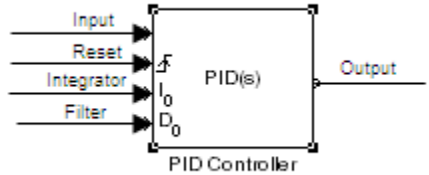
Simulink does not permit the filter initial condition to be `inf` or `NaN`.

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:



Or, if the **Initial conditions Source** is `External`,



The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Note: To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the `PID controller` block.

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.

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 1-1299).

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.

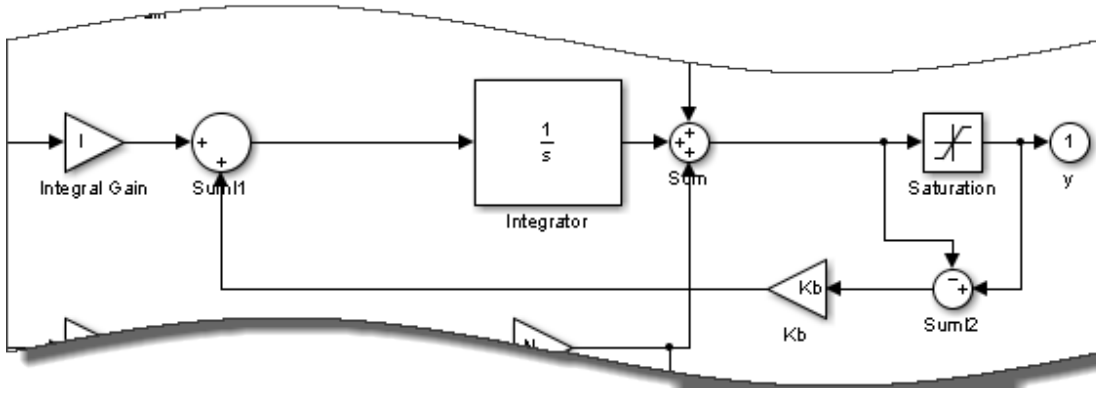
Settings

`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`

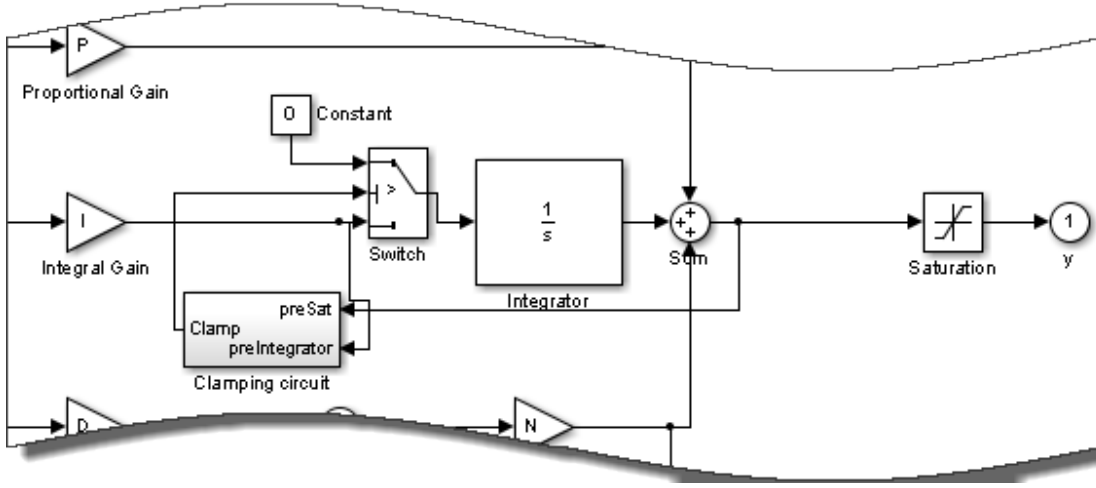
Discharges the integrator when the block output saturates using the integral-gain feedback loop:



You can also specify a value for the **Back-calculation coefficient (Kb)**.

`clamping`

Stops integration when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Resumes integration when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. The integrator portion of the block is:



The clamping circuit implements the logic necessary to determine whether integration continues.

Back-calculation gain (Kb)

(Available only when the `back-calculation` **Anti-windup method** is active.) Specify the gain coefficient of the anti-windup feedback loop.

The `back-calculation` anti-windup method discharges the integrator on block saturation using a feedback loop having gain coefficient `Kb`.

Default: 1

Ignore saturation when linearizing

Force Simulink linearization commands ignore `PID Controller` block output limits. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the `PID Controller` block to exceed the output limits.

Settings

 `On` (Default)

Simulink linearization commands ignore states corresponding to saturation.

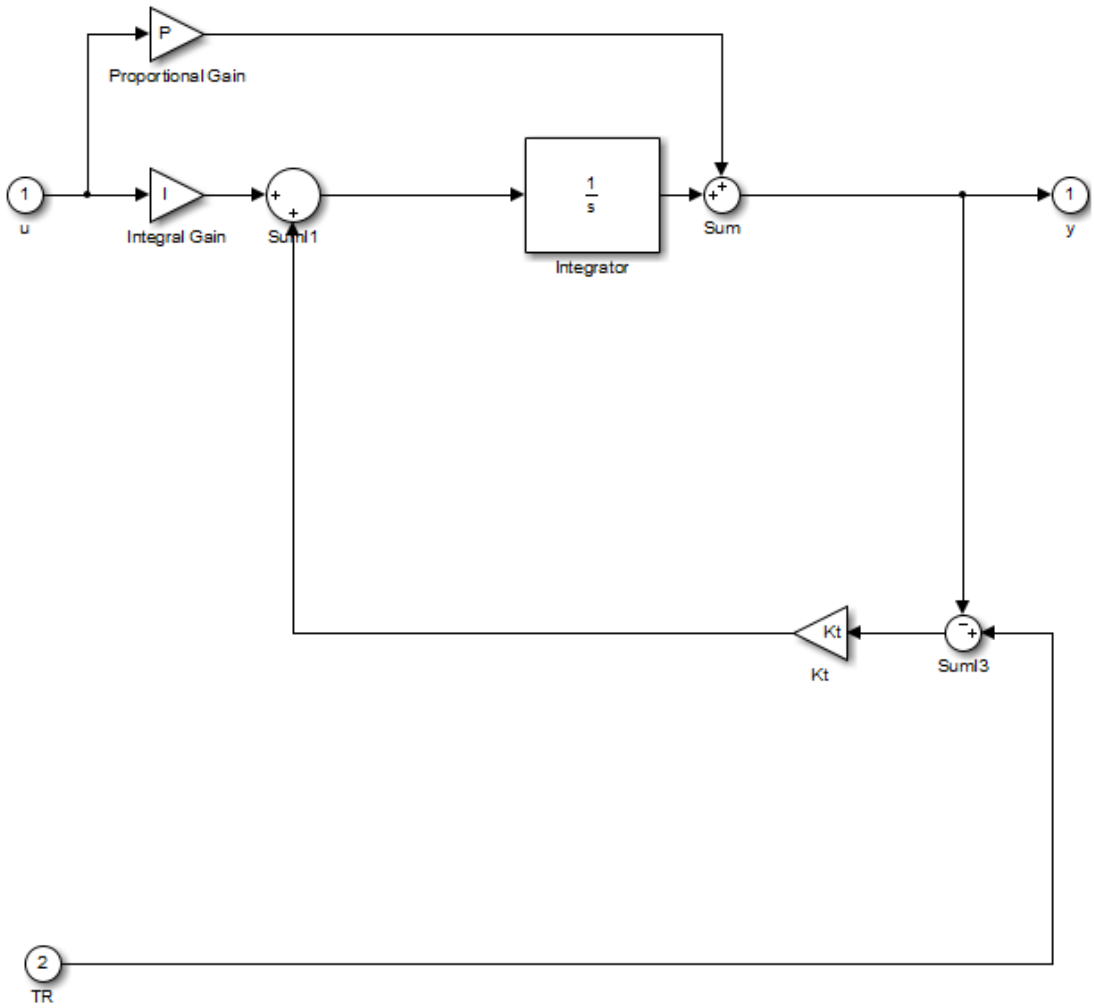
 `Off`

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**.

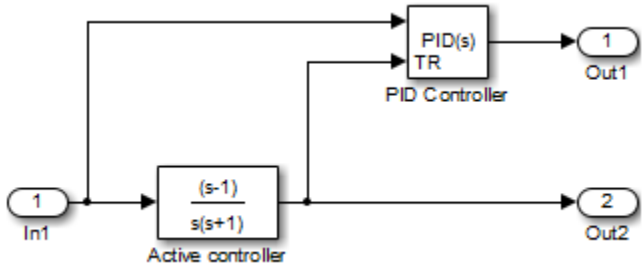
When signal tracking is active, the difference between the tracked signal and the block output is fed back to the integrator input with a gain `Kt`. The structure is illustrated for a PI controller:



You can also specify the **Tracking coefficient (Kt)**.

Bumpless control transfer

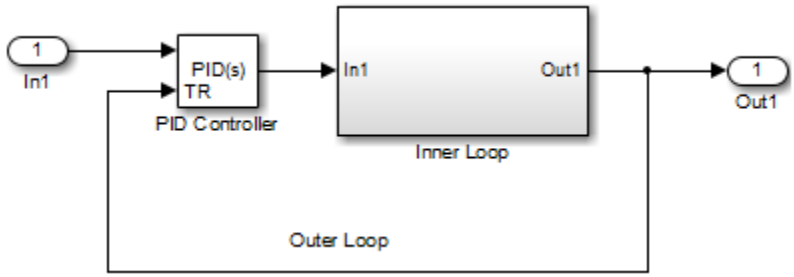
Use signal tracking, for example, to achieve bumpless control transfer in systems that switch between two controllers. You can make one controller track the output of the other controller by connecting the TR port to the signal you want to track. For example:



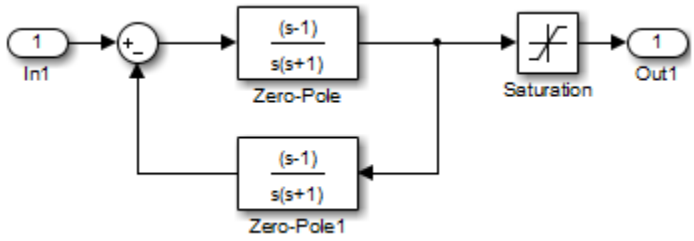
In this example, the outputs `Out1` and `Out2` can drive a controlled system (not shown) through a switch that transfers control between the "Active controller" block and the `PID Controller` block. The signal tracking feature of the `PID Controller` block provides smooth operation upon transfer of control 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:



The inner-loop subsystem contains the following blocks:



In this example, the inner loop has an effective gain of 1 when it does not saturate. Without signal tracking, the inner loop winds up in saturation. Signal tracking ensures that the `PID Controller` output does not exceed the saturated output of the inner loop.

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 6-101 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 6-101 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 6-101 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 6-101 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.

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 6-101 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 6-101 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 6-101 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.

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 6-101 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.

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 6-101 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.

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 6-101 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.

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 6-101 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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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

Parameter: `StateMustResolveToSignalObject`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Code generation storage class

Select state storage class for code generation.

Settings

Default: `Auto`

`Auto`

> `Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

> Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".
>
> Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

State name enables this parameter.

Command-Line Information

Command-Line Information
Parameter: `StateStorageClass`
Type: string
Value: `'Auto' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'SimulinkGlobal' | 'Custom'`
Default: `'Auto'`

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Settings

- **Default:** `' '` (empty string)
- `const`
- `volatile`

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Command-Line Information

Parameter Name: `RTWStateStorageTypeQualifier`
Value Type: string
Default: `' '` (empty string)

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`.

Introduced in R2009b

PID Controller (2 DOF)Discrete PID Controller (2 DOF)

Simulate continuous- or discrete-time two-degree-of-freedom PID controllers

Library

Continuous, Discrete

Description

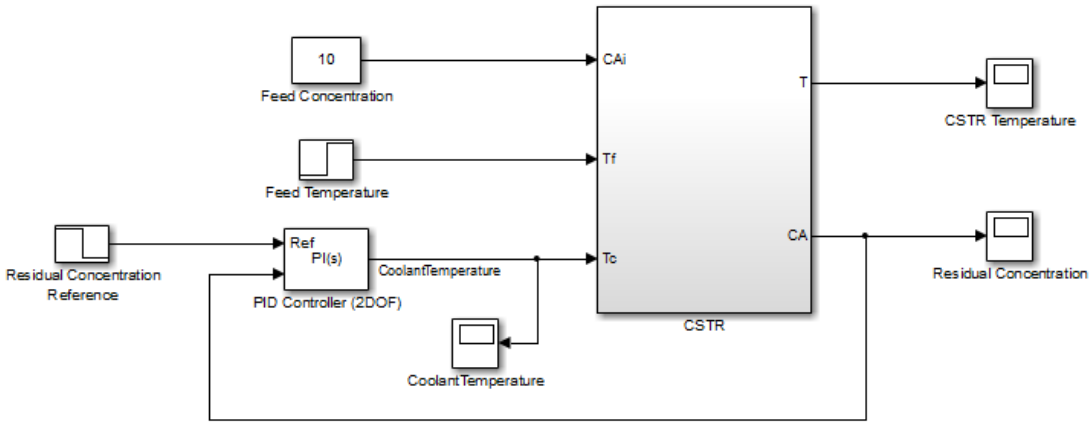
Implement a continuous- or discrete-time two-degree-of-freedom controller (PID, PI, or PD) in your Simulink model. The `PID Controller (2DOF)` block allows you to implement setpoint weighting in your controller to achieve both smooth setpoint tracking and good disturbance rejection.

The `PID Controller (2DOF)` block generates an output signal based on the difference between a reference signal and a measured system output. The block computes a weighted difference signal for each of the proportional, integral, and derivative actions according to the setpoint weights you specify. The block output is the sum of the proportional, integral, and derivative actions on the respective difference signals, where each action is weighted according to the gain parameters. A first-order pole filters the derivative action. Controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software (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:



For a single-input block that accepts an error signal (a difference between a setpoint and a system output), see the PID Controller block reference page.

You can generate code to implement your controller using any Simulink data type, including fixed-point data types. (Code generation requires Simulink Coder software; fixed-point implementation requires the Fixed-Point Designer product.)

For an example illustrating an application of the `PID Controller (2 DOF)` block, see the Simulink example Two Degree-of-Freedom PID Control for Setpoint Tracking.

Data Type Support

The `PID Controller (2DOF)` block accepts real signals of any numeric data type that Simulink software supports, including fixed-point data types. See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Parameters

The following table summarizes the `PID Controller (2DOF)` block parameters, accessible via the block parameter dialog box.

Task	Parameters
Choose controller form and type.	• **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.	• **Controller Parameters Source** in Main tab • **Proportional (P)** in Main tab • **Integral (I)** in Main tab • **Derivative (D)** in Main tab • **Filter coefficient (N)** in Main tab • **Use filtered derivative** 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
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

Task	Parameters
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 • **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

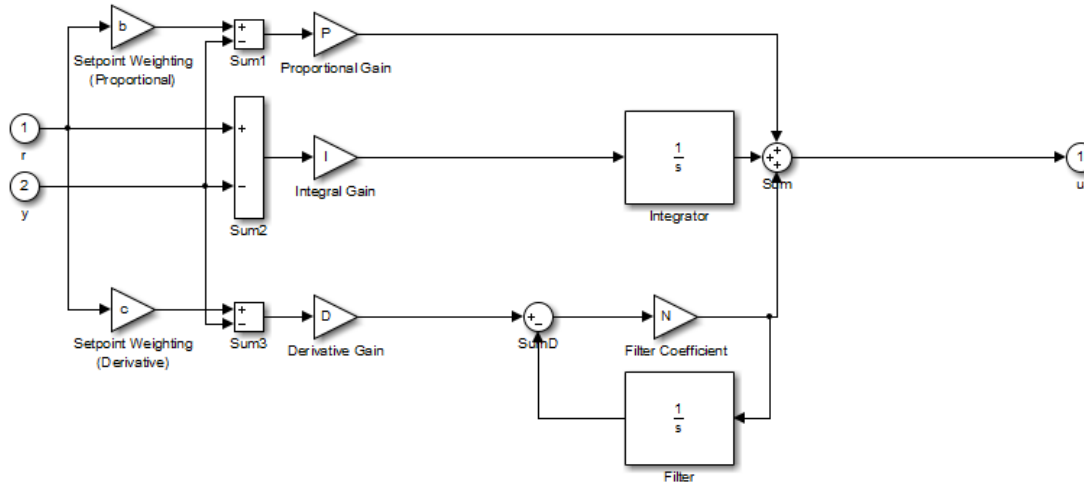
Select the controller form.

Settings

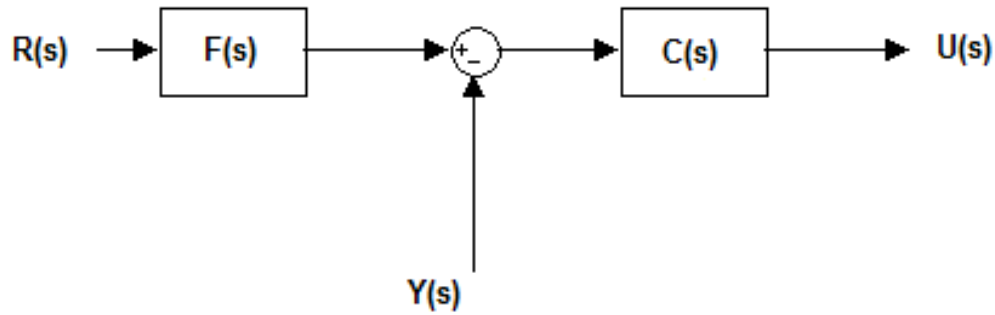
`Parallel` (Default)

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:



The parallel two-degree-of-freedom PID controller can be equivalently modeled by the following block diagram:



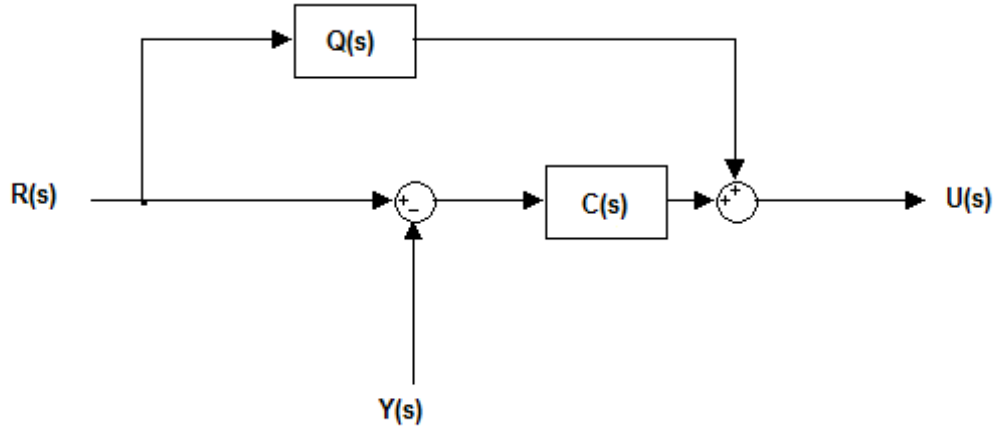
`R(s)` represents the reference signal and `Y(s)` represents the feedback from measured system output. In this model, `C(s)` is a single degree-of-freedom controller, and `F(s)` acts as a prefilter on the reference signal. For a parallel two-degree-of-freedom PID controller in the `Continuous-time` **Time-domain**, 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:



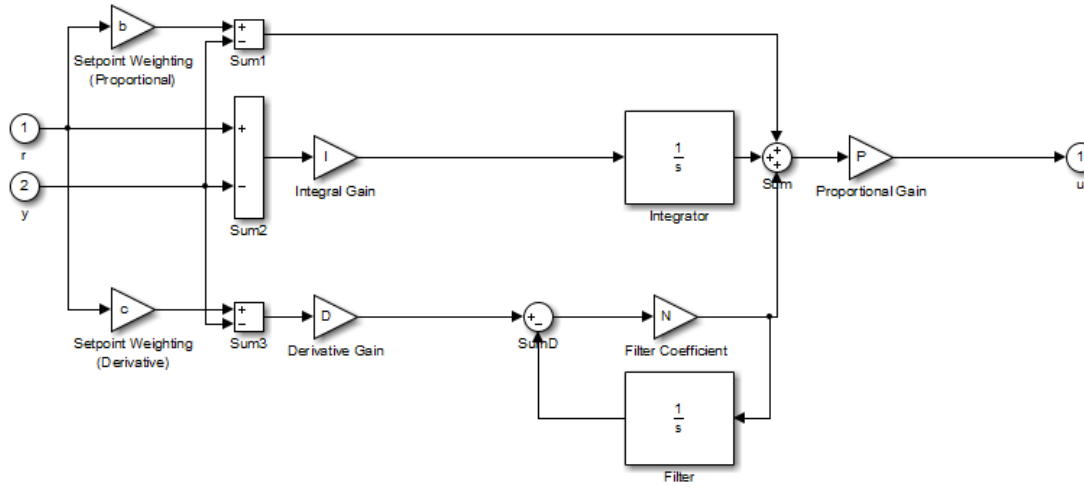
`R(s)`, `Y(s)`, and `C(s)` are as discussed previously. In this realization, `Q(s)` acts as feed-forward conditioning on the reference signal `R(s)`. For a parallel PID controller in the `Continuous-time` **Time-domain**, the transfer function `Q(s)` is:

$$Q_{par}(s) = \frac{\left((b-1)P + (c-1)DN\right)s + (b-1)PN}{s+N}$$

`Ideal`

Selects a controller form in which the proportional gain `P` acts on the sum of all actions.

Ideal two-degree-of-freedom PID controller, where input 1 receives a reference signal and input 2 receives feedback from the measured system output:



Similarly to the parallel controller form discussed previously, the ideal two-degree-of-freedom PID controller can be modeled as a single degree-of-freedom controller `C(s)` with a prefilter `F(s)`. For an ideal two-degree-of-freedom PID controller in the `Continuous-time` **Time-domain**, the transfer functions `F(s)` and `C(s)` are:

$$F_{id}(s) = \frac{(b + cDN)s^2 + (bN + I)s + IN}{(1 + DN)s^2 + (N + I)s + IN}$$

$$C_{id}(s) = P\frac{(1 + DN)s^2 + (N + I)s + IN}{s(s + N)}$$

where `b` and `c` are the **Setpoint weight** parameters.

Alternatively, modeling the ideal two-degree-of-freedom PID controller as a one-degree-of-freedom controller `C(s)` with feed-forward conditioning `Q(s)` on the reference signal gives, in continuous-time:

$$Q_{id}(s) = P\frac{\left((b-1) + (c-1)DN\right)s + (b-1)N}{s + N}$$

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.

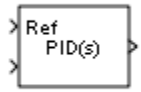
`PD`

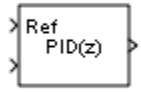
Implements a controller with proportional and derivative action.

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.

When the `PID Controller (2DOF)` block is in a model with synchronous state control (see the `State Control` block), you cannot select `Continuous-time`.

`Discrete-time`

Selects the discrete-time representation. Selecting `Discrete-time` also allows you to specify the:

- **Sample time**, which is the discrete interval between samples.
- Discrete integration methods for the integrator and the derivative filter using the **Integrator method** and **Filter method** menus.

Integrator method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the integrator output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

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.

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.

If you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

`Trapezoidal`

Selects the Bilinear approximation.

An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

If you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a

model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

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.

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.

This filter method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

`Trapezoidal`

Selects the Bilinear approximation.

An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method. Of all available filter methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

This filter method can cause algebraic loops in your controller. Algebraic loops can slow down simulation of the model. In addition, if you want to generate code

using Simulink Coder software or the Fixed-Point Designer product, you cannot generate code for a model that contains an algebraic loop. For more information about algebraic loops in Simulink models, see “Algebraic Loops” in the Simulink documentation.

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.

Controller Parameters Source

Select the source of the controller gains, filter coefficient, and setpoint weights. You can provide these parameters explicitly in the block dialog box, or enable external inputs for them on the block. Enabling external inputs for the parameters allows you to compute PID gains and filter coefficients externally to the block and provide them to the block as signal inputs.

External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control, in which controller gains are determined by logic or other calculation in the Simulink model and passed to the block.

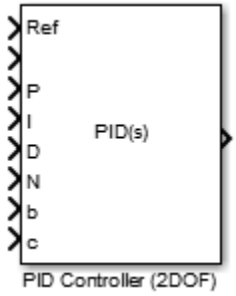
Settings

`internal` (Default)

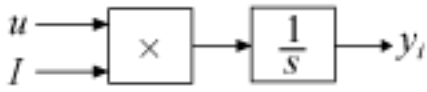
Specify the PID gains and filter coefficient explicitly using the **P**, **I**, **D**, **N**, **b**, and **c** parameters.

`external`

Specify the PID gains and filter coefficient externally. An additional input port appears under the block input for each parameter that is required for the current controller type:



When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because of the way the PID gains are implemented within the block. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.



Within the block, the signal to be integrated is multiplied by the externally-supplied integrator gain, *I*, before integration. This implementation yields:

$$y_i = \int uI\,dt.$$

Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain *D* to be differentiated.

Proportional (P)

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 1-1354 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 Control 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 Control Design Approach”.

Derivative (D)

(Available for PID and PD controllers.) Specify the derivative gain `D`.

Default: 0

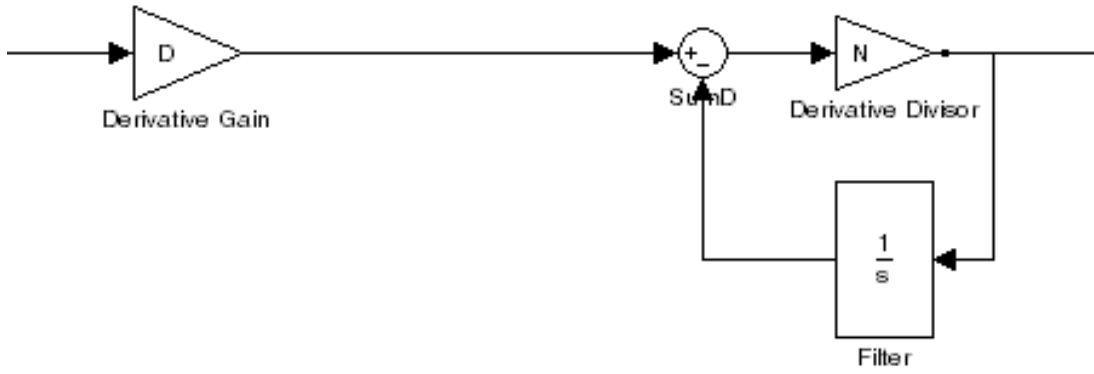
Enter a finite, real gain value into the **Derivative (D)** field. Use either scalar or vector gain values.

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 Control Design Approach”.

Filter coefficient (N)

Specifies the filter coefficient of the controller.

(Available for PID and PD controllers, when **Use filtered derivative** is checked.) Specify the filter coefficient `N`, which determines the pole location of the filter in the derivative action:



The filter pole falls at `s = -N` in the `Continuous-time` **Time-domain**. For `Discrete-time`, the location of the pole depends on which **Filter method** you select (for sampling time T_s):

- `Forward Euler`:

  $$z_{pole} = 1 - NT_s$$

- `Backward Euler`:

  $$z_{pole} = \frac{1}{1 + NT_s}$$

- `Trapezoidal`:

  $$z_{pole} = \frac{1 - NT_s / 2}{1 + NT_s / 2}$$

Default: 100.

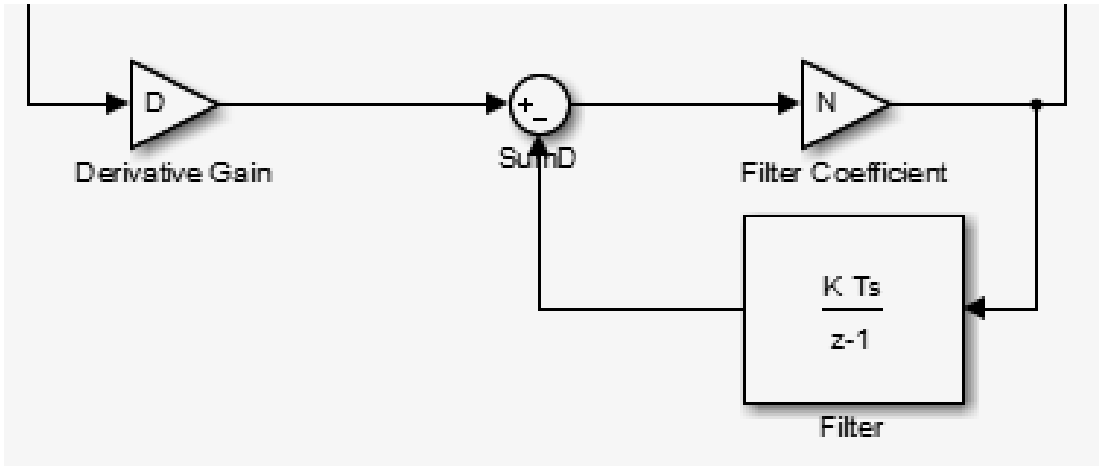
Enter a finite, real gain value into the **Filter Coefficient (N)** field. Use either scalar or vector gain values. Note that the `PID controller (2DOF)` block does not support `N = inf` (ideal unfiltered derivative).

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 Control Design Approach". Automatic tuning requires `N > 0`.

Use Filtered Derivative

Specify whether derivative term is filtered (finite N) or unfiltered. Unfiltered derivative is available only for discrete-time controllers.

Unchecking this option replaces the filtered derivative with a discrete differentiator. For example, if **Filter Method** is `Forward Euler`, then the filtered derivative term is represented by:



When you uncheck **Use filtered derivative**, the derivative term becomes:

Settings

☑ On (Default)

Use derivative filter (finite N).

Off

Derivative is unfiltered.

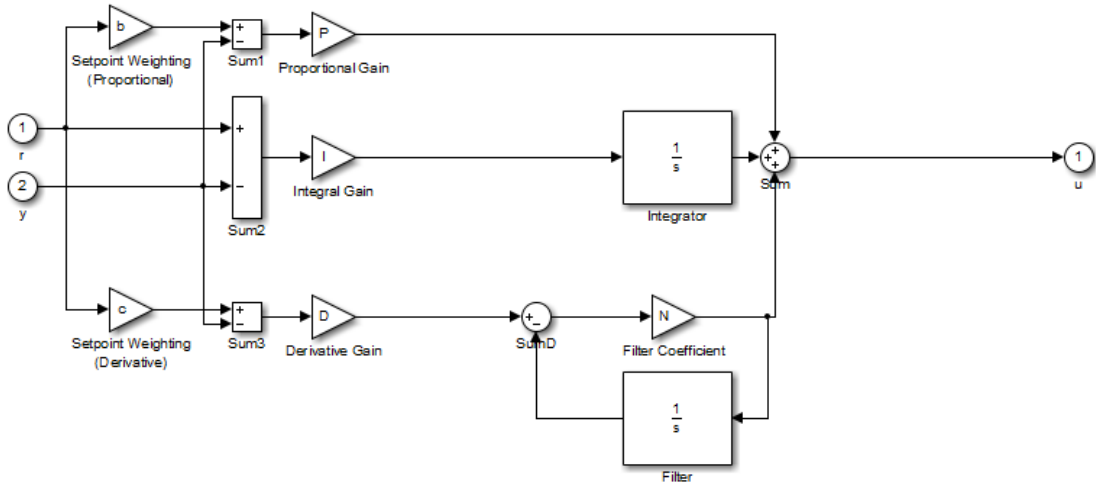
Setpoint weight (b)

Specify the proportional setpoint weight `b`.

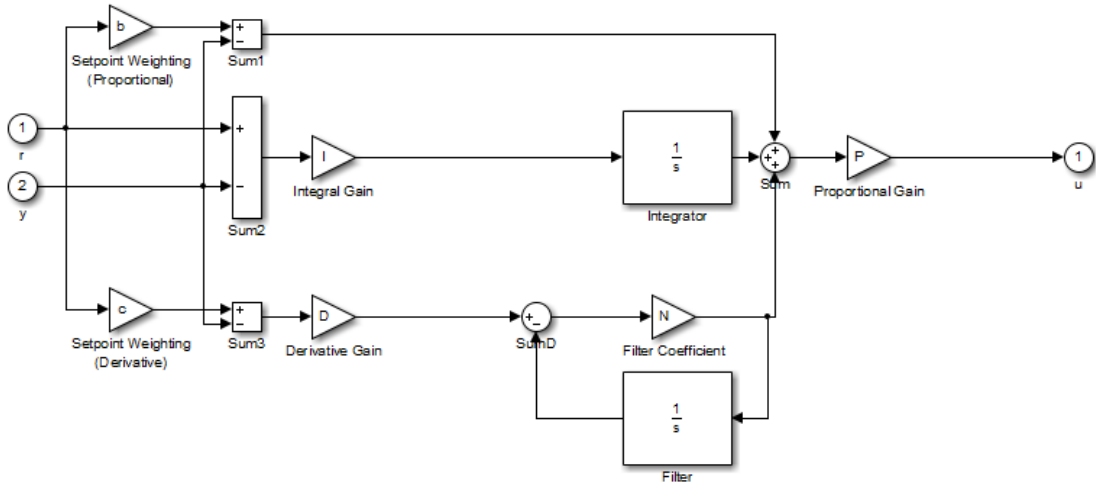
Default: 1

Enter the proportional setpoint weight value into the **Setpoint weight (b)** field. Setting `b = 0` eliminates the proportional action on the reference signal, which can reduce overshoot in the system response to step changes in the setpoint.

The following diagrams show the role of **Setpoint weight (b)** in PID controllers of `Parallel` and `Ideal` form. See "Controller form" on page 1-1354 for a discussion of the corresponding transfer functions.



Parallel Two-Degree-of-Freedom PID Controller



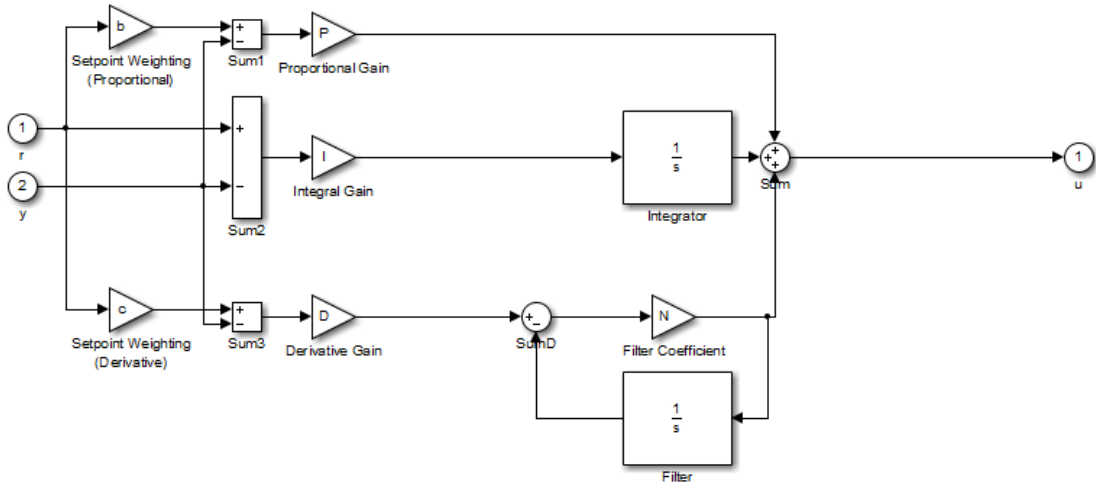
Ideal Two-Degree-of-Freedom PID Controller

Setpoint weight (c)

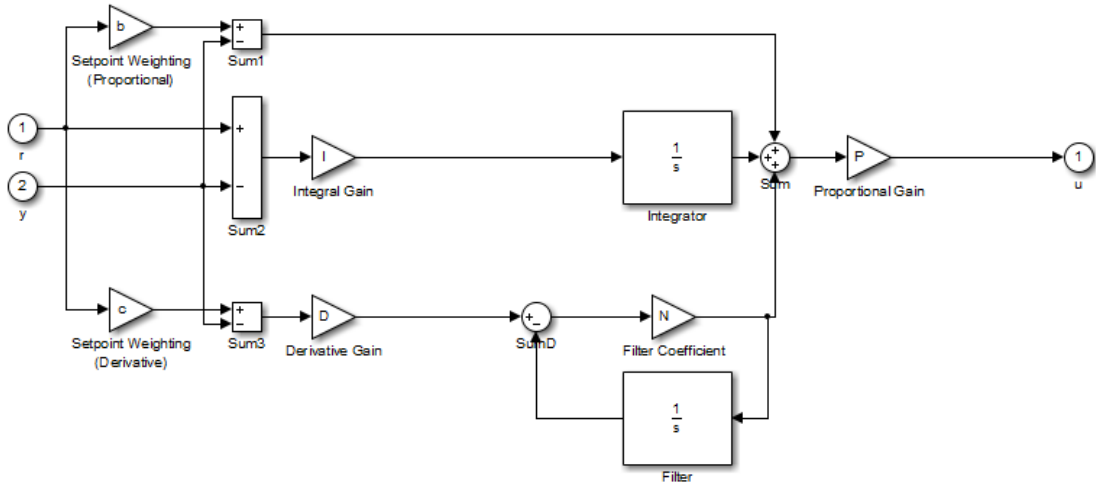
(Available for PID and PD controllers.) Specify the derivative setpoint weight `c`.

Enter the derivative setpoint weight value into the **Setpoint weight (c)** field. To implement a controller that achieves both effective disturbance rejection and smooth setpoint tracking without excessive transient response, set `c = 0`. Setting `c = 0` yields a controller with derivative action on the measured system response but not on the reference input.

The following diagrams show the role of **Setpoint weight (c)** in `Parallel` and `Ideal` PID controllers. See "Controller form" on page 1-1354 for a discussion of the corresponding transfer functions.



Parallel Two-Degree-of-Freedom PID Controller



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 1-1381). The integrator and filter initial conditions in turn determine the initial block output.

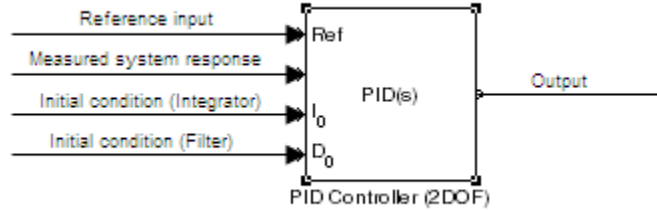
Settings

`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:



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 1-1381). The integrator initial condition, together with the filter initial condition, determines the initial output of the `PID Controller (2DOF)` block.

Default: 0

Simulink does not permit the integrator initial condition to be `inf` or `NaN`.

Filter Initial condition

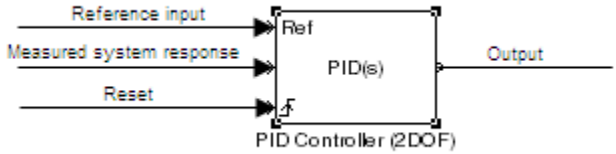
(Available only when **Initial conditions Source** is `internal`, the controller includes derivative action, and **Use filtered derivative** is checked.) Specify the filter initial value. Simulink uses the initial condition to initialize the filter output at the start of a simulation or at a specified trigger event (see "External reset" on page 1-1381). The filter initial condition, together with the integrator initial condition, determines the initial output of the `PID Controller (2DOF)` block.

Default: 0

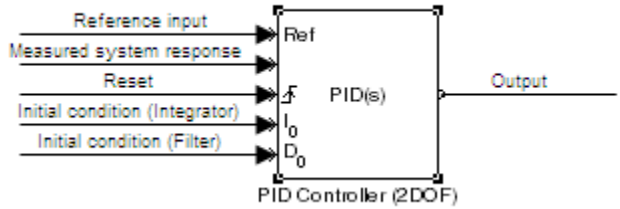
Simulink does not permit the filter initial condition to be `inf` or `NaN`.

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:



Or, if the **Initial conditions Source** is `External`:



The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Note: To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the `PID controller (2DOF)` block.

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.

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 1-1388).

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.

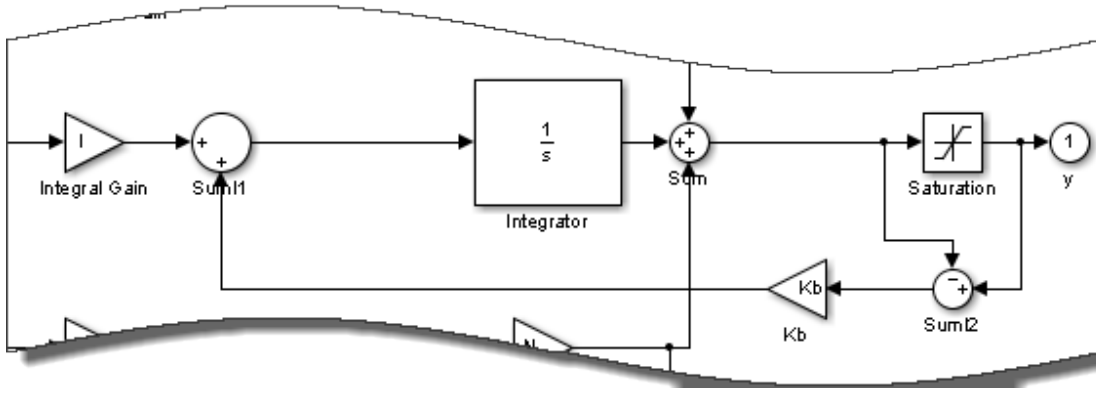
Settings

`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`

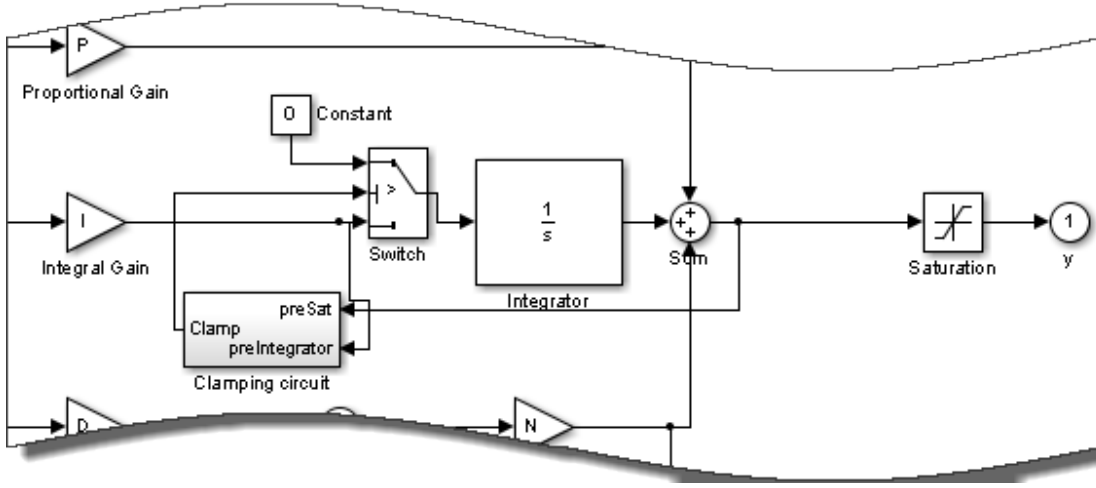
Discharges the integrator when the block output saturates using the integral-gain feedback loop:



You can also specify a value for the **Back-calculation coefficient (Kb)**.

`clamping`

Stops integration when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Resumes integration when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. The integrator portion of the block is:



The clamping circuit implements the logic necessary to determine whether integration continues.

Back-calculation gain (Kb)

(Available only when the `back-calculation` **Anti-windup method** is active.) Specify the gain coefficient of the anti-windup feedback loop.

The `back-calculation` anti-windup method discharges the integrator on block saturation using a feedback loop having gain coefficient `Kb`.

Default: 1

Ignore saturation when linearizing

Force Simulink linearization commands ignore `PID Controller (2DOF)` block output limits. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the `PID Controller (2DOF)` block to exceed the output limits.

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 1-1303 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 6-101 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 6-101 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 6-101 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 6-101 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.

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 6-101 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 6-101 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 6-101 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.

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 6-101 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.

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 6-101 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.

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 6-101 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.

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 6-101 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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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

Parameter: `StateMustResolveToSignalObject`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Code generation storage class

Select state storage class for code generation.

Settings

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

State name enables this parameter.

Command-Line Information

Command-Line Information
Parameter: `StateStorageClass`
Type: string
Value: `'Auto' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'SimulinkGlobal' | 'Custom'`
Default: `'Auto'`

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Settings

- **Default:** `' '` (empty string)
- `const`
- `volatile`

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Command-Line Information

Parameter Name: `RTWStateStorageTypeQualifier`
Value Type: string
Default: `' '` (empty string)

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**, 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`.

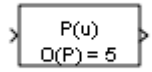
Introduced in R2009b

Polynomial

Perform evaluation of polynomial coefficients on input values

Library

Math Operations



Description

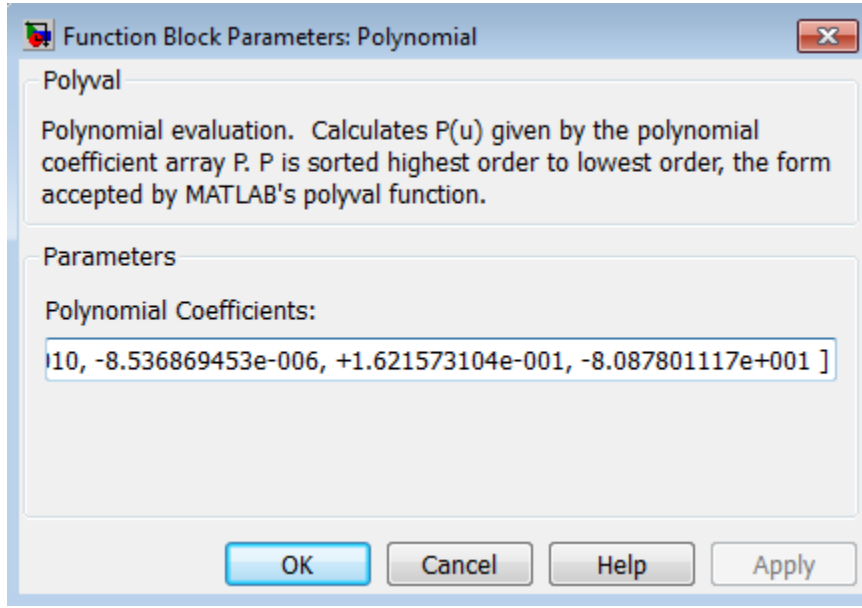
You define a set of polynomial coefficients in the form that the MATLAB `polyval` command accepts. The block evaluates *`P(u)`* at each time step for the input u. The inputs and coefficients must be real.

Data Type Support

The Polynomial block accepts real signals of type `double` or `single`. The **Polynomial coefficients** parameter must be of the same type as the inputs. The output data type is the same as the input data type.

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

Parameters and Dialog Box



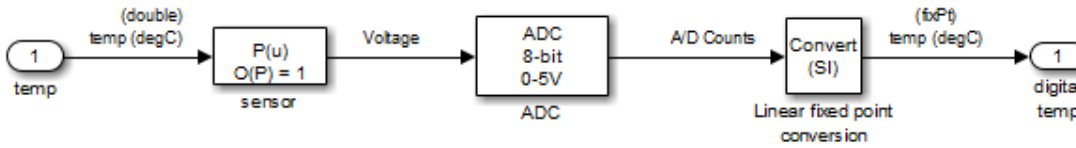
Polynomial coefficients

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]`:



This subsystem models a digital thermometer composed of a simple temperature sensor and an ADC. The transfer function of the sensor is:
V = .05 * T + 0.75
for T in degrees C.

The conversion block inverts the combined transfer function of the sensor and ADC so that the output is an sfix(8) code representing T in degrees C.

Characteristics

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

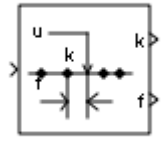
Introduced before R2006a

Prelookup

Compute index and fraction for Interpolation Using Prelookup block

Library

Lookup Tables



Description

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. These two blocks have distributed algorithms. When combined together, they perform the same operation as the integrated algorithm in the n-D Lookup Table block. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility that can provide more efficient simulation and code generation. For more information, see "Efficiency of Performance" in the *Simulink 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 $0 \le f < 1$, 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.

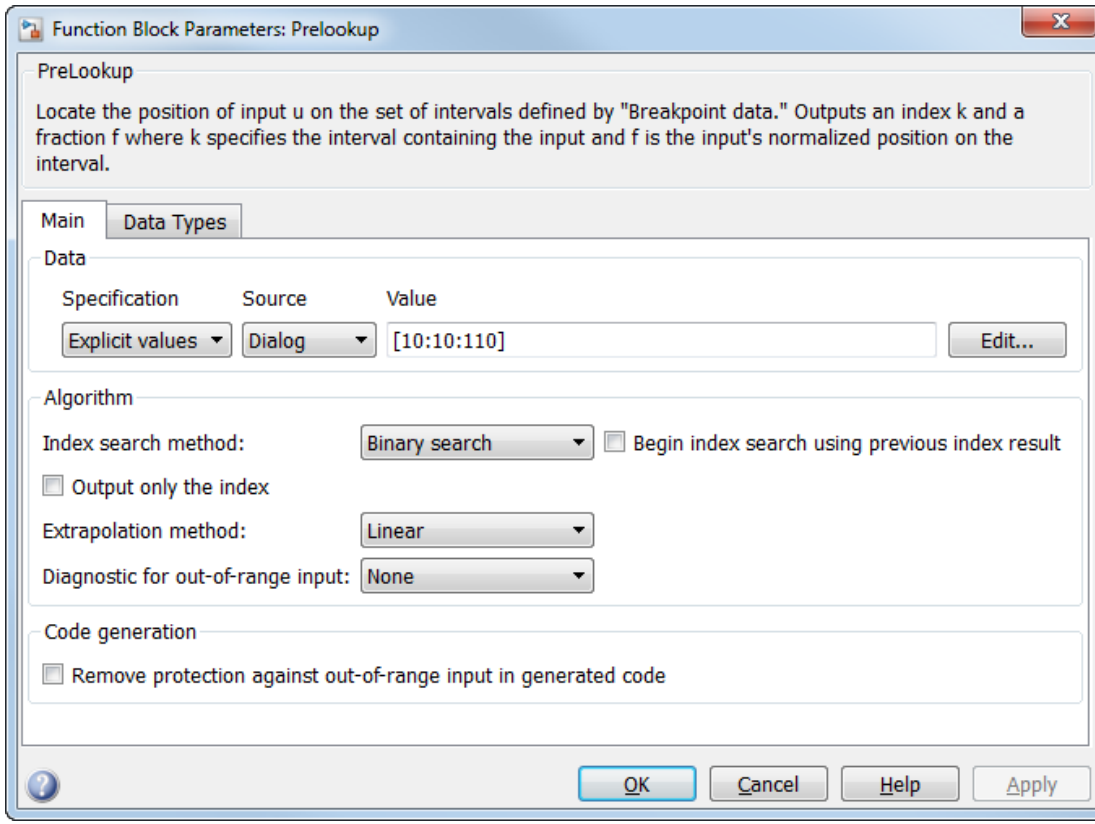
The interval fraction can be negative or greater than `1` for out-of-range input. See the documentation for the **Extrapolation method** block parameter for more information.

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



- "Main Tab" on page 1-1441
- "Data Types tab" on page 1-1446

Main Tab

Specification

Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- If you set this parameter to `Explicit values`, the **Source** and **Value** parameters are visible on the dialog box.

- If you set this parameter to `Even spacing`, the **First point**, **Spacing**, and **Number of points** parameters are visible on the dialog box.

Source

Specify whether to enter breakpoint data in the dialog box or to inherit the data from an input port. This parameter is available when **Specification** is set to `Explicit values`.

- If you set **Source** to `Dialog`, enter breakpoint data in the text box under **Value**.
- 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 **Edit** to open the Lookup Table Editor (see "Edit Lookup Tables" in the Simulink documentation).

Value

Explicitly specify the breakpoint data. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, you specify breakpoint attributes on the **Data Types** pane. This parameter is available when **Source** is set to `Dialog`.

First point

Specify the first point in your evenly spaced breakpoint data. This parameter is available when **Specification** is set to `Even spacing`.

Spacing

Specify the spacing between points in your evenly spaced breakpoint data. This parameter is available when **Specification** is set to `Even spacing`.

Number of points

Specify the number of evenly spaced points in your breakpoint data. This parameter is available when **Specification** is set to `Even spacing`.

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 Diagnostic 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 "Run Model Checks" in the Simulink documentation.

Sample time

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

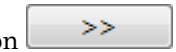
Data Types tab

Note: The parameters for breakpoint attributes (data type, minimum, and maximum) are not available when you set **Source** to `Input port`. In this case, the block inherits all breakpoint attributes from the `bp` input port.

Breakpoint > Data Type

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

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

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the breakpoint data type.

See "Specify Data Types Using Data Type Assistant" in the Simulink User's Guide for more information.

Tip Specify a breakpoint data type different from the data type of input `u` for these cases:

- Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal `u`
- Sharing of prescaled breakpoint data between two `Prelookup` blocks with different data types for input `u`
- Sharing of custom storage breakpoint data in the generated code for blocks with different data types for input `u`

Breakpoint > Minimum

Specify the minimum value that the breakpoint data can have. The default value is `[ ]` (unspecified).

Breakpoint > Maximum

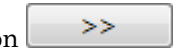
Specify the maximum value that the breakpoint data can have. The default value is `[ ]` (unspecified).

Index > Data Type

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.

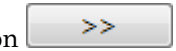
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the index data type.

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

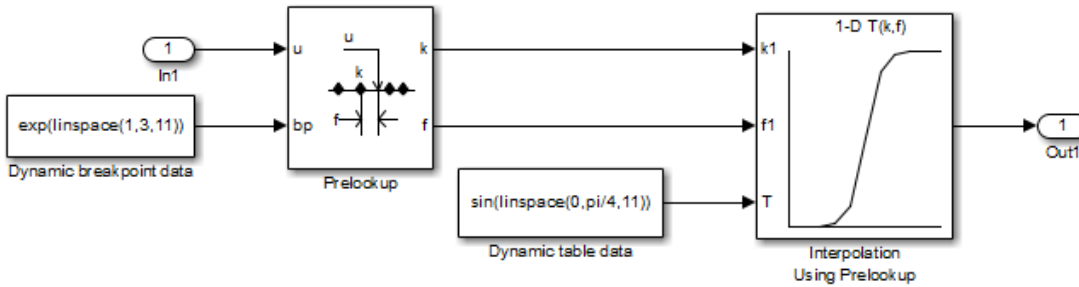
Specify the rounding mode for fixed-point operations. For more information, see “Rounding” in the Simulink Fixed Point documentation.

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

Prelookup with External Breakpoint Specification

In the following model, a `Constant` block feeds the breakpoint data set to the `bp` input port of the `Prelookup` block.



The Prelookup block inherits the following breakpoint attributes from the `bp` input port:

Breakpoint Attribute	Value
Minimum	`-Inf`
Maximum	`Inf`
Data type	`single`

Similarly, a Constant block feeds the table data values to the `T` input port of the `Interpolation Using Prelookup` block, which inherits the following table attributes:

Table Attribute	Value
Minimum	`-Inf`
Maximum	`Inf`
Data type	`single`

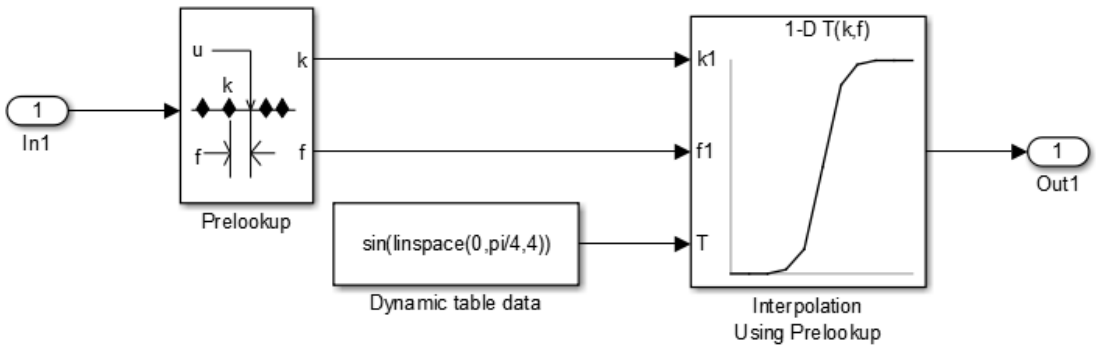
Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision, floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.

Prelookup with Even Spaced Breakpoint Specification

In this model, you specify evenly spaced breakpoint data to the `Prelookup` block.

The **Breakpoint Specification** parameter is set to `Even Spacing`. The parameters **First point**, **Spacing**, and **Number of points** are set to `25`, `12`, and `4` respectively. Specifying these parameters creates four evenly spaced breakpoints: `[25, 37, 49, 61]`.

Specifying even spacing is an alternative way to specify breakpoints that are evenly spaced. You can also set **Breakpoint Specification** to `Explicit values` and set **Value** to `[25:12:61]`.



Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoints and table data as double.

For other examples, see “Prelookup and Interpolation Blocks” in the Simulink documentation.

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Inherited from driving block

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

See Also

`Interpolation Using Prelookup`

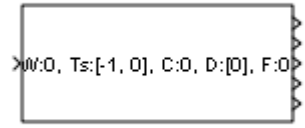
Introduced in R2006b

Probe

Output signal attributes, including width, dimensionality, sample time, and complex signal flag

Library

Signal Attributes



Description

The Probe block outputs selected information about the signal on its input. The block can output the input signal's width, dimensionality, sample time, and a flag indicating whether the input is a complex-valued signal. The block has one input port. The number of output ports depends on the information that you select for probing, that is, signal dimensionality, sample time, and/or complex signal flag. Each probed value is output as a separate signal on a separate output port. The block accepts real or complex-valued signals of any built-in data type. It outputs signals of type `double`. During simulation, the block icon displays the probed data.

Data Type Support

The Probe block accepts signals of the following data types:

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

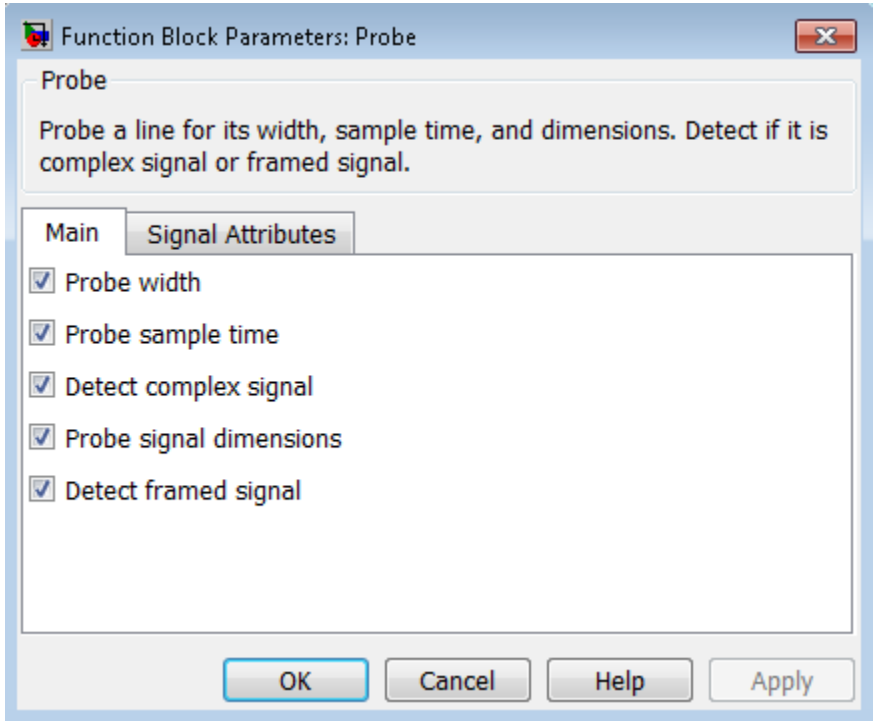
- Enumerated (input only)
- Bus object

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

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

Parameters and Dialog Box

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



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.

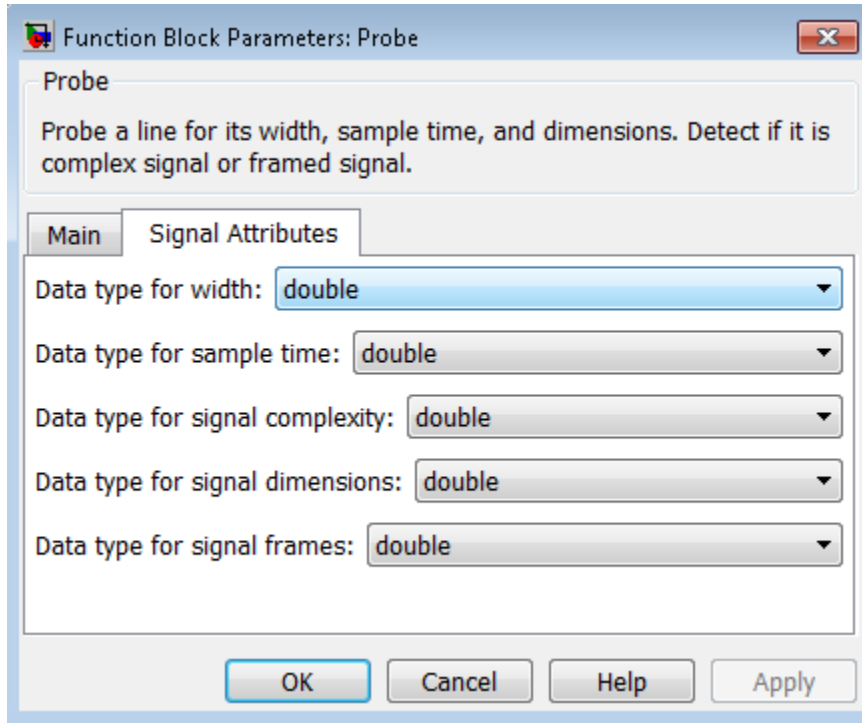
Probe signal dimensions

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:



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

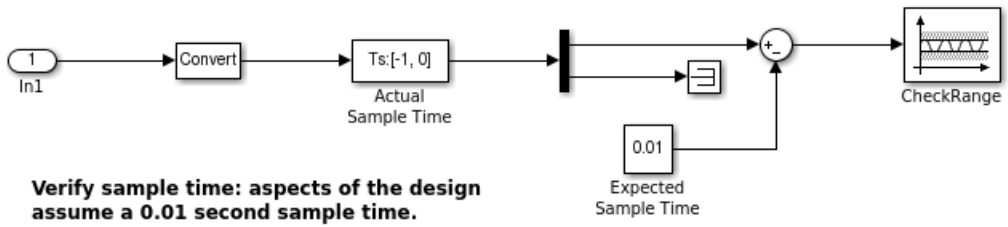
Select the output data type for the frames information.

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

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

Introduced before R2006a

Product

Multiply and divide scalars and nonscalars or multiply and invert matrices

Library

Math Operations

Description

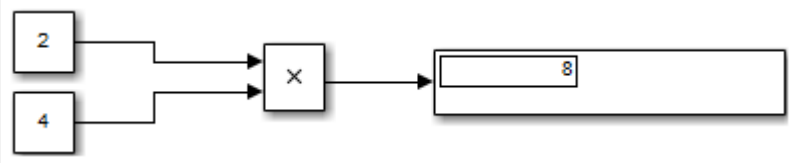
The Divide and Product of Elements blocks are variants of the Product block.

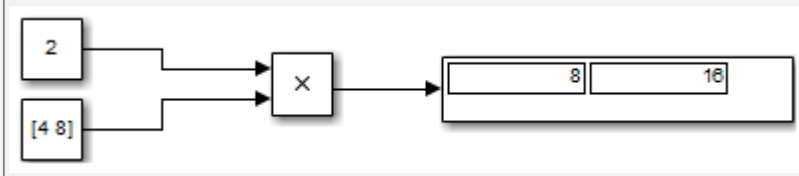
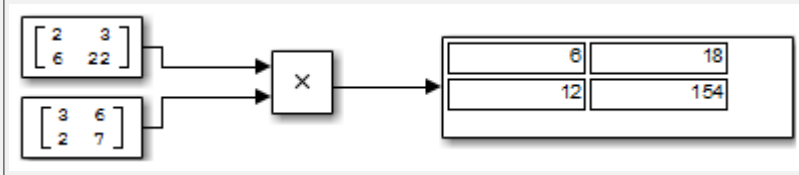
- For information on the Divide block, see `Divide`.
- For information on the Product of Elements block, see `Product of Elements`.

The Product block outputs the result of multiplying two inputs: two scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The default parameter values that specify this behavior are:

- **Multiplication:** `Element-wise(.*)`
- **Number of inputs:** `2`

The following table shows the output of the Product block for example inputs using default block parameter values.

Inputs and Behavior	Example
Scalar X Scalar Output the product of the two inputs.	

Inputs and Behavior	Example
Scalar X Nonscalar Output a nonscalar having the same dimensions as the input nonscalar. Each element of the output nonscalar is the product of the input scalar and the corresponding element of the input nonscalar.	
Nonscalar X Nonscalar Output a nonscalar having the same dimensions as the inputs. Each element of the output is the product of corresponding elements of the inputs.	

The 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.

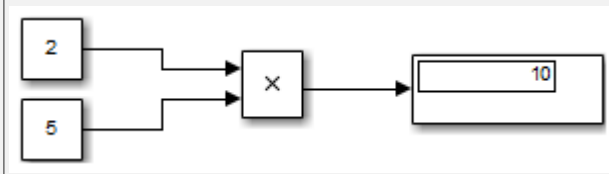
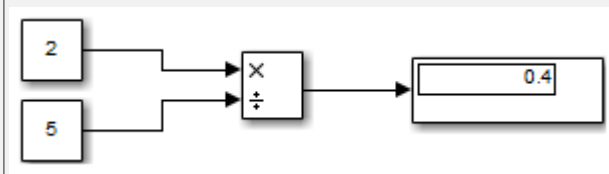
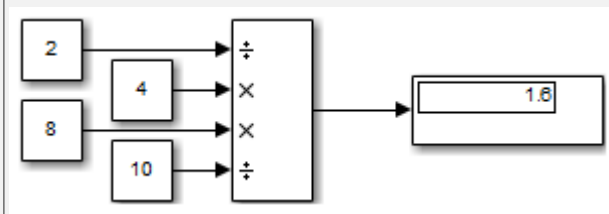
The Product block has two modes: *Element-wise mode*, which processes nonscalar inputs element by element, and *Matrix mode*, which processes nonscalar inputs as matrices. The next two sections describe these two modes.

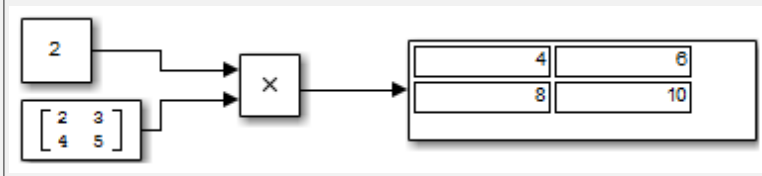
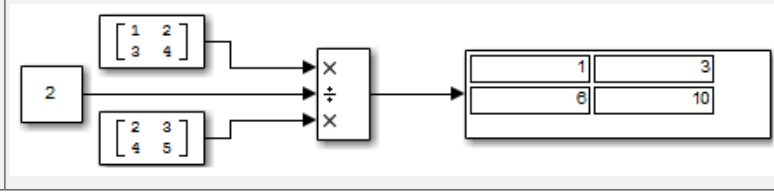
Element-wise Mode

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: */*	

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.

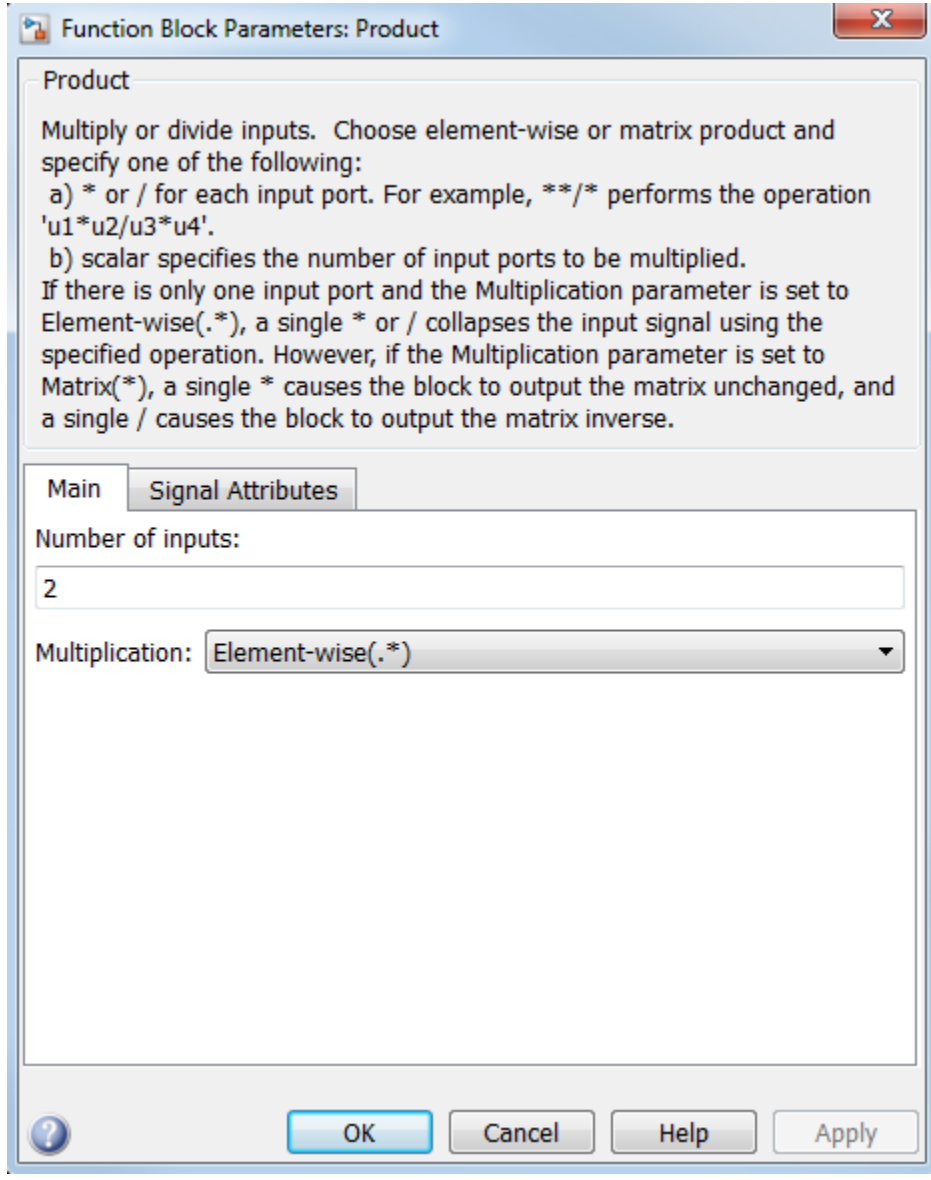
Data Type Support

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 1-1459, and "Matrix Mode" on page 1-1460 for more information.

Parameters and Dialog Box



Number of inputs

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

Settings

Default: `2` for Product block, `*/` for `Divide` block, and `*` for `Product of Elements` block

- **`1` or `*` or `/`**

  Has one input. In element-wise mode, processes the input as described for the `Product of Elements` block. In matrix mode, if the parameter value is `1` or `*`, the block outputs the input value. If the value is `/`, the input must be a square matrix (including a scalar as a degenerate case) and the block outputs the matrix inverse. See "Element-wise Mode" on page 1-1459 and "Matrix Mode" on page 1-1460 for more information.

- **Integer value > 1**

  Has the number of inputs given by the integer value. The inputs are multiplied together in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See "Element-wise Mode" on page 1-1459 and "Matrix Mode" on page 1-1460 for more information.

- **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 1-1459 and "Matrix Mode" on page 1-1460 for more information.

Dependency

Setting **Number of inputs** to `*` and selecting `Element-wise(.*)` for **Multiplication** enables the **Multiply over** parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 6-101 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 6-101 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 6-101 for the command-line information.

Sample time

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

Require all inputs to have the same data type

Require that all inputs have the same data type.

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 6-101 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.

Simulink uses the minimum to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges") 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 6-101 for the command-line information.

See Also

For more information, see "Control Signal 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 "Control Signal 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 6-101 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.

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.

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**

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.

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.

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.

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Characteristics

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

See Also

- `Divide`

- Dot Product
- Product of Elements

Introduced before R2006a

Product of Elements

Copy or invert one scalar input, or collapse one nonscalar input

Library

Math Operations

Description

The Divide and Product blocks are variants of the Product of Elements block.

- For information on the Divide block, see `Divide`.
- For information on the Product block, see `Product`.

The Product of Elements block inputs one scalar, vector, or matrix. You can use the block to:

- Copy a scalar input unchanged
- Invert a scalar input (divide 1 by it)
- Collapse a vector or matrix to a scalar by multiplying together all elements or taking successive inverses of the elements
- Collapse a matrix to a vector by multiplying together the elements of each row or column or taking successive inverses of the elements of each row or column

The Product of Elements block is functionally a `Product` block that has two preset parameter values:

- **Multiplication**: `Element-wise(.*)`
- **Number of inputs**: `*`

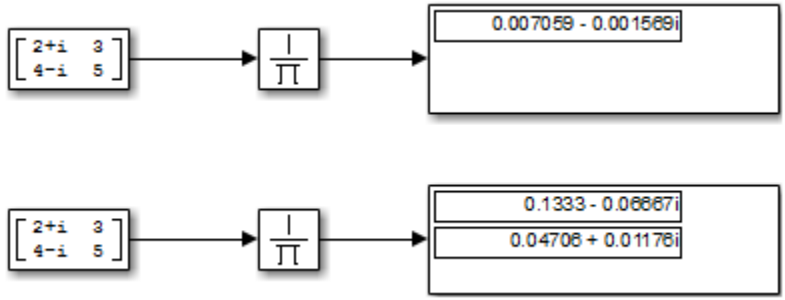
Setting non-default values for either of those parameters can change a Product of Elements block to be functionally equivalent to a `Product` block or a `Divide` block. See the documentation of those two blocks for more information.

Algorithm

The Product of Elements block uses the following algorithms to perform element-wise operations on inputs of floating-point, built-in integer, and fixed-point types:

Input	Element-Wise Operation	Algorithm
Real scalar, `u`	Multiplication	`y = u`
	Division	`y = 1/u`
Real vector or matrix, with elements `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:



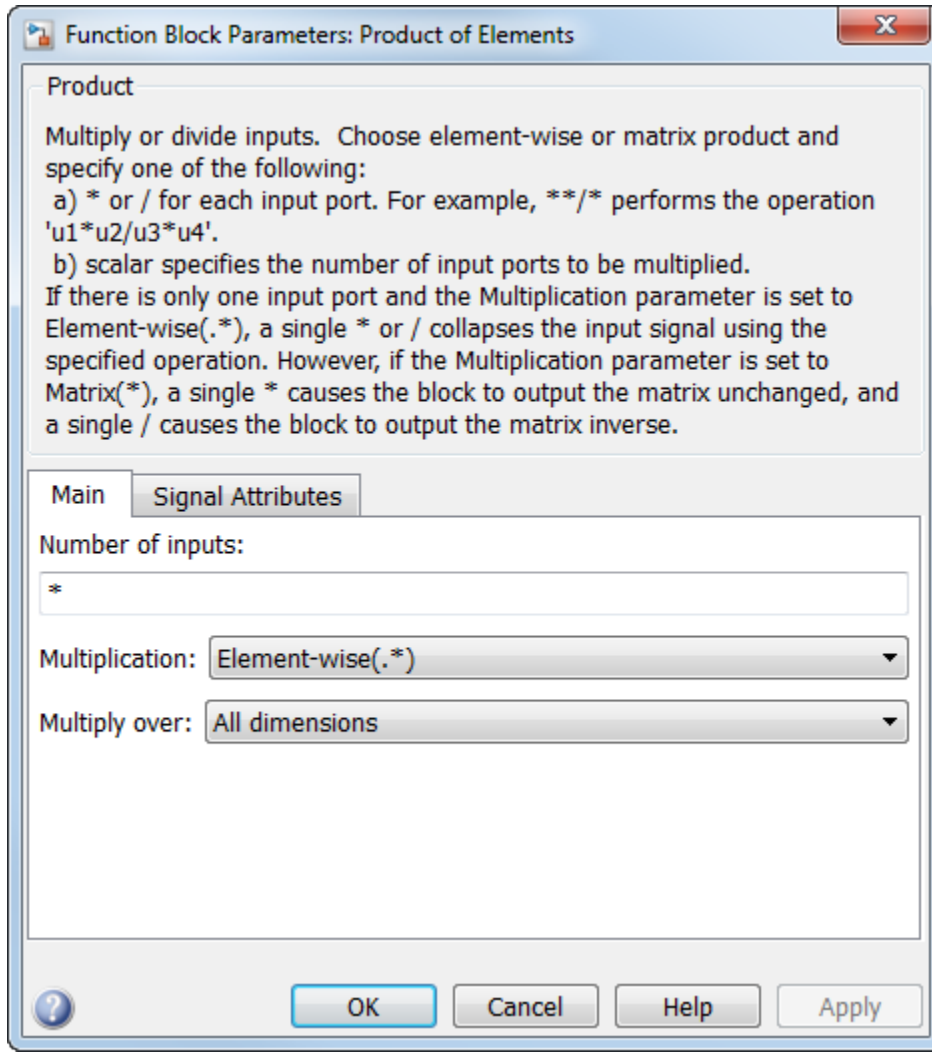
The top Product of Elements block collapses the matrix input to a scalar by taking successive inverses of the four elements:

- `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



Number of inputs

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

Settings

Default: *

- **1 or ***

  - Copies a scalar input unchanged
  - Collapses a vector input to a scalar by multiplying all elements together
  - Collapses a matrix input to a scalar or vector by multiplying elements together based on the **Multiply over** parameter

  For more information, see “Algorithm” on page 1-1481.

- **/**

  - 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 1-1481.

- **Integer value > 1**

  Has the number of inputs given by the integer value. The block becomes a product block and the input are multiplied together in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1459 and “Matrix Mode” on page 1-1460 in the Product documentation for more information.

- **Unquoted string of two or more * and / characters**

  Has the number of inputs given by the length of the string. The block becomes a product or divide block and multiplies each input that corresponds to a * character into the output. Each input that corresponds to a / character is divided into the output. The operations occur in Element-wise mode or Matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1459 and “Matrix Mode” on page 1-1460 in the Product block reference page for more information.

Dependency

Setting **Number of inputs** to `*` and selecting `Element-wise(.*)` for **Multiplication** enables the following **Multiply over** parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 6-101 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 6-101 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 6-101 for the command-line information.

Sample time

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

Require all inputs to have the same data type

Require that all inputs have the same data type.

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 6-101 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.

Simulink uses the minimum to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see "Control Block Parameter Value Ranges") 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 6-101 for the command-line information.

See Also

For more information, see "Control Signal 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 "Control Signal 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 6-101 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.

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.

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**

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.

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.

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.

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

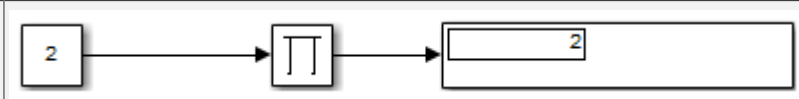
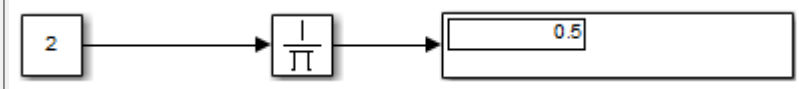
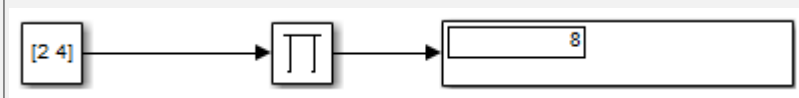
  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".
- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

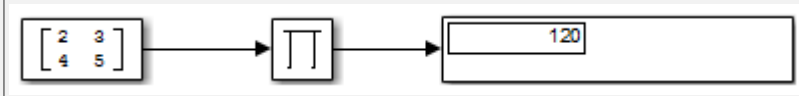
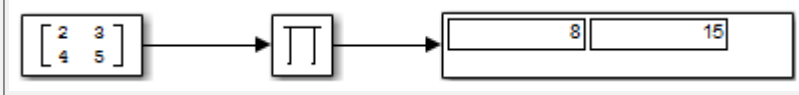
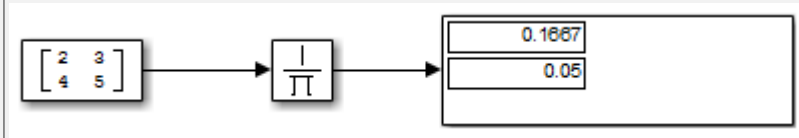
Command-Line Information

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Examples

This table shows the output of the Product of Elements block for example inputs using default block parameter values, except as shown in the table.

Parameter Values	Examples
Multiplication: `Element-wise(.*)` **Number of inputs**: *	
Multiplication: `Element-wise(.*)` **Number of inputs**: /	
Multiplication: `Element-wise(.*)` **Number of inputs**: *	

Parameter Values	Examples
Multiplication: `Element-wise(.*)` **Number of inputs**: * **Multiply over**: `All dimensions`	
Multiplication: `Element-wise(.*)` **Number of inputs**: * **Multiply over:** `Specified dimension` **Dimension**: 1	
Multiplication: `Element-wise(.*)` **Number of inputs**: / **Multiply over**: `Specified dimension` **Dimension**: 2	

Characteristics

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

Introduced before R2006a

Pulse Generator

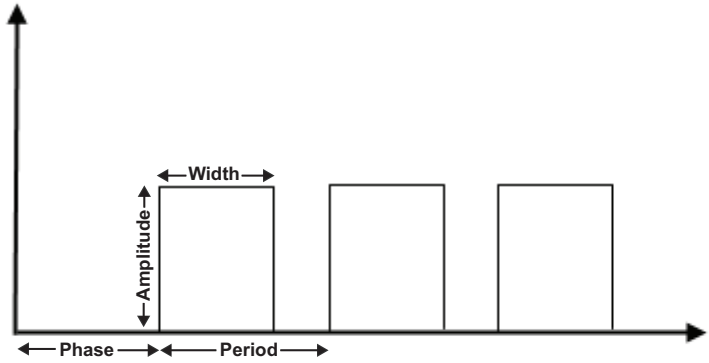
Generate square wave pulses at regular intervals

Library

Sources

Description

The Pulse Generator block generates square wave pulses at regular intervals. The block waveform parameters, **Amplitude**, **Pulse Width**, **Period**, and **Phase delay**, determine the shape of the output waveform. The following diagram shows how each parameter affects the waveform.



The Pulse Generator can emit scalar, vector, or matrix signals of any real data type. To cause the block to emit a scalar signal, use scalars to specify the waveform parameters.

To cause the block to emit a vector or matrix signal, use vectors or matrices, respectively, to specify the waveform parameters. Each element of the waveform parameters affects the corresponding element of the output signal. For example, the first element of a vector amplitude parameter determines the amplitude of the first element of a vector output pulse. All the waveform parameters must have the same dimensions after scalar expansion. The data type of the output is the same as the data type of the **Amplitude** parameter.

This block output can be generated in time-based or sample-based modes, determined by the **Pulse type** parameter.

Time-Based Mode

In time-based mode, Simulink computes the block output only at times when the output actually changes. This approach results in fewer computations for the block output over the simulation time period. Activate this mode by setting the **Pulse type** parameter to `Time based`.

The block does not support a time-based configuration that results in a constant output signal. Simulink returns an error if the parameters **Pulse Width** and **Period** satisfy either of these conditions:

$$Period * \frac{PulseWidth}{100} = 0$$

$$Period * \frac{PulseWidth}{100} = Period$$

Depending on the pulse waveform characteristics, the intervals between changes in the block output can vary. For this reason, a time-based Pulse Generator block has a variable sample time. The sample time color of such blocks is brown (see “View Sample Time Information” for more information).

Simulink cannot use a fixed-step solver to compute the output of a time-based pulse generator. If you specify a fixed-step solver for models that contain time-based pulse generators, Simulink computes a fixed sample time for the time-based pulse generators. Then the time-based pulse generators simulate as sample based.

If you use a fixed-step solver and the **Pulse type** is `Time based`, choose the step size such that the period, phase delay, and pulse width (in seconds) are integer multiples of

the solver step size. For example, suppose that the period is `4` seconds, the pulse width is `75%` (that is, `3` s), and the phase delay is `1` s. In this case, the computed sample time is `1` s. Therefore, choose a fixed-step size of `1` or a number that divides `1` exactly (e.g., `0.25`). You can guarantee this by setting the fixed-step solver step size to `auto` on the **Solver** pane of the Configuration Parameters dialog box.

Sample-Based Mode

In sample-based mode, the block computes its outputs at fixed intervals that you specify. Activate this mode by setting the **Pulse type** parameter to `Sample based`.

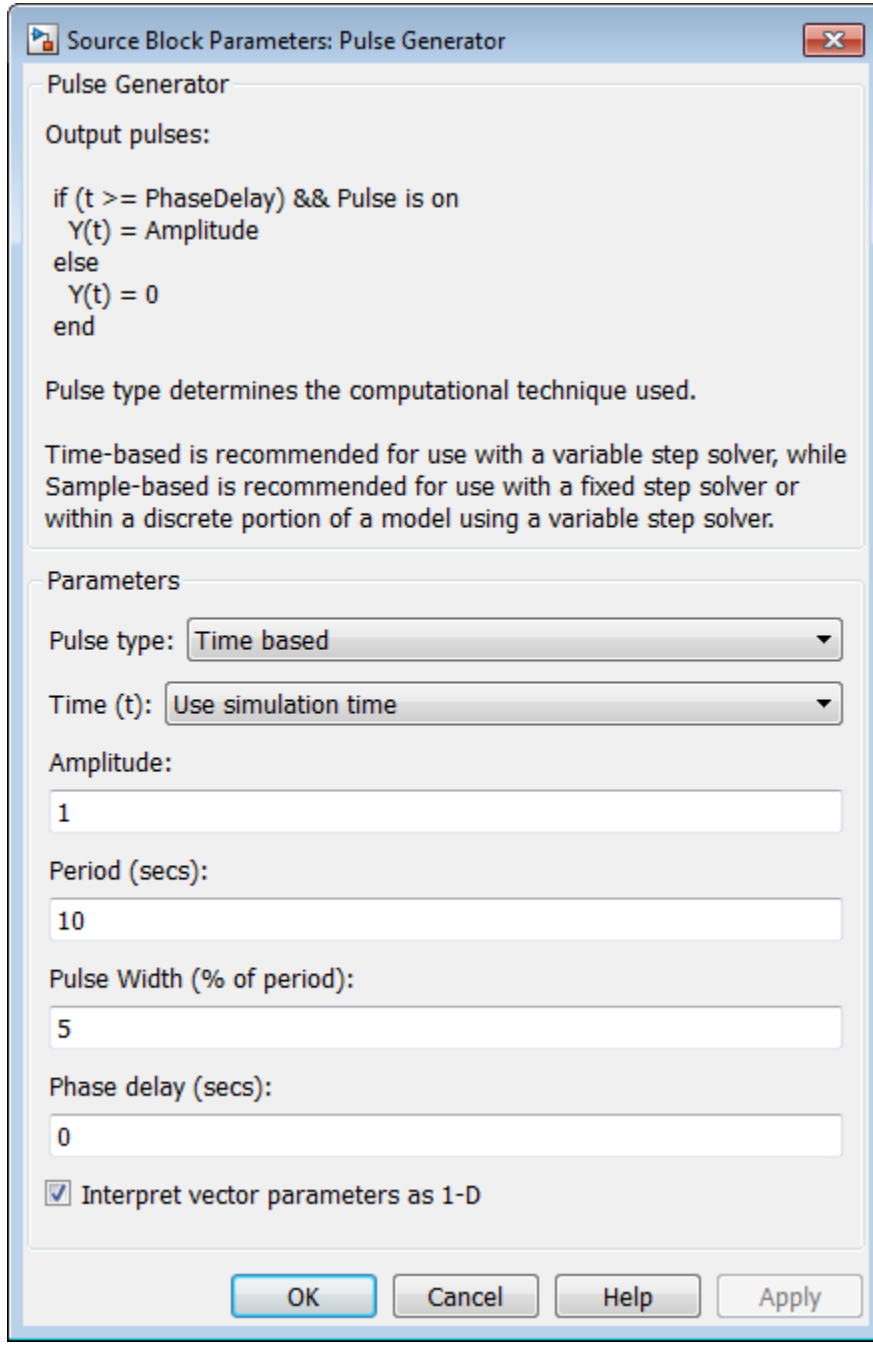
An important difference between the time-based and sample-based modes is that in time-based mode, the block output is based on simulation time, and in sample-based mode, the block output depends only on the simulation start, regardless of elapsed simulation time. For more information, see the example "Difference Between Time-Based and Sample-Based Pulse Generation Modes" on page 1-1506.

This block supports reset semantics in sample-based mode. For example, if a Pulse Generator is in a resettable subsystem that hits a reset trigger, the block output resets to its initial condition.

Data Type Support

The Pulse Generator block outputs real signals of any numeric data type that Simulink supports, including fixed-point data types. The data type of the output signal is the same as that of the **Amplitude** parameter.

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


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
OK
Cancel
Help
Apply

Pulse type

The pulse type for this block: `Time based` or `Sample based`. The default is `Time based`.

Time (t)

Specifies whether to use simulation time or an external signal as the source of values for the output pulse's time variable. If you specify an external source, the block displays an input port for connecting the source. The output pulse differs as follows:

- `Use simulation time`: The block generates an output pulse where the time variable equals the simulation time.
- `Use external signal`: The block generates an output pulse where the time variable equals the value from the input port, which can differ from the simulation time.

Amplitude

The pulse amplitude. The default is `1`.

Period

The pulse period specified in seconds if the pulse type is time-based or as number of sample times if the pulse type is sample-based. The default is `10` seconds.

Pulse Width

The duty cycle specified as the percentage of the pulse period that the signal is on if time-based or as number of sample times if sample-based. The default is `5` percent.

Phase delay

The delay before the pulse is generated specified in seconds if the pulse type is time-based or as number of sample times if the pulse type is sample-based. The default is `0` seconds.

Sample time

The length of the sample time for this block in seconds. This parameter appears only if the block's pulse type is sample-based. See “Specify Sample Time” in the Simulink User's Guide for more information.

Interpret vector parameters as 1-D

If you select this check box and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise the output dimensionality is the same as that of the other parameters. See “Determining the Output Dimensions of Source Blocks” in the “Working with Signals” chapter of the Simulink User's Guide.

Examples

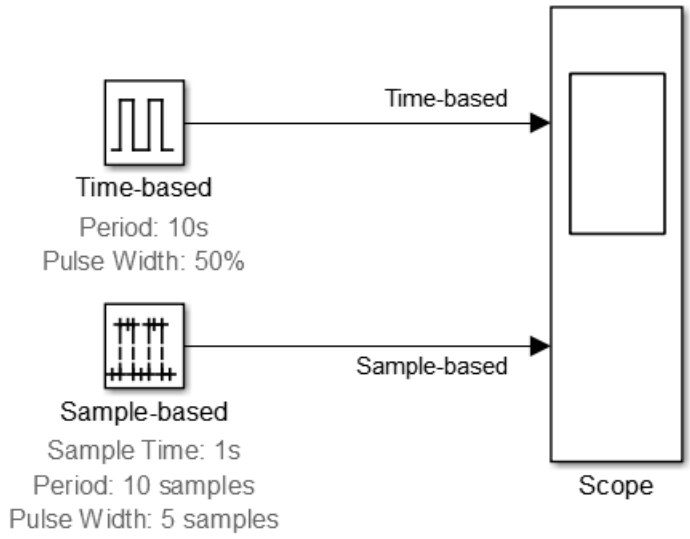
The following Simulink examples show how to use the Pulse Generator block:

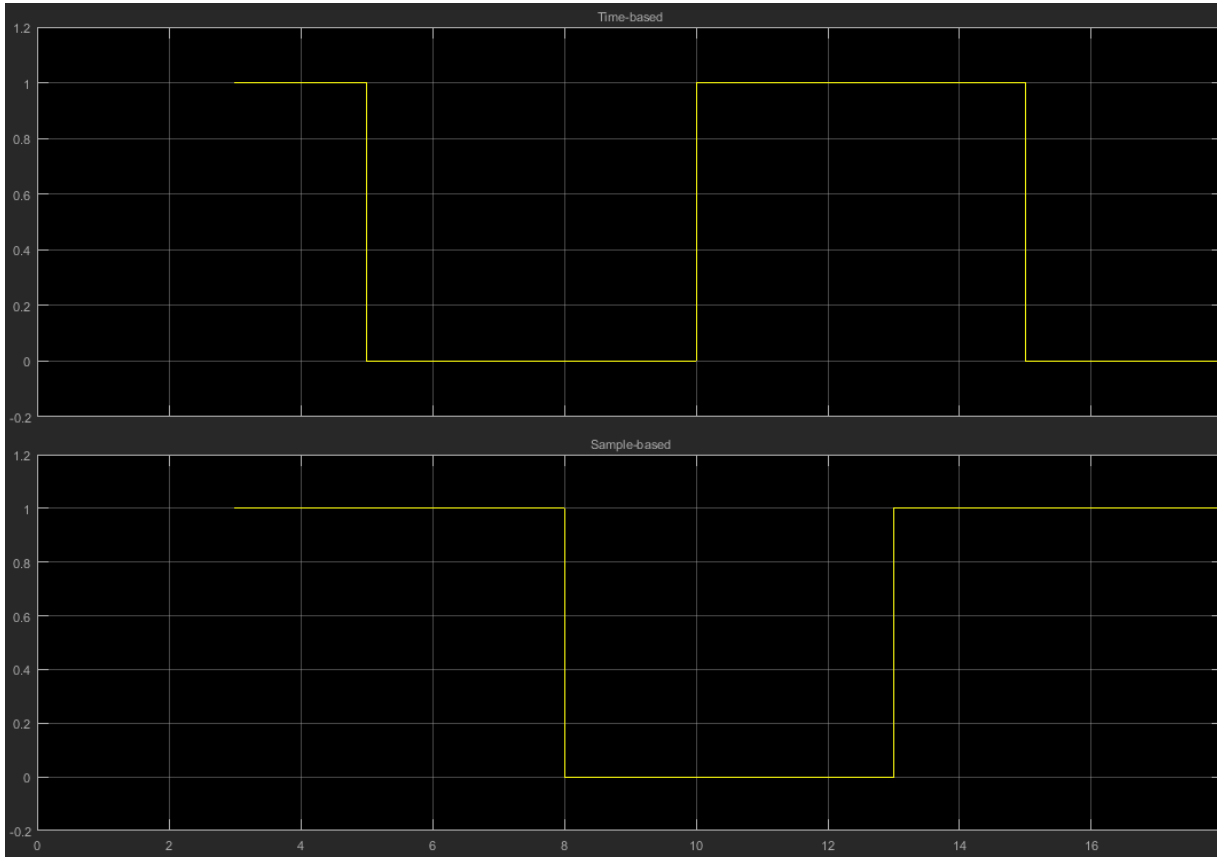
- `sldemo_auto_climatecontrol`
- `sldemo_boiler`

Difference Between Time-Based and Sample-Based Pulse Generation Modes

This example shows the difference in the behavior of the Pulse Generator block in time-based and sample-based modes.

Consider a model with two Pulse Generator blocks. In one block, the **Pulse type** parameter is set to `Time based`. In the other block, it is set to `Sample based`. Both blocks are set up to output a Boolean pulse of 10 seconds: 5 seconds `on` followed by 5 seconds `off`. The simulation runs for 15 seconds from a start time of 3 seconds to a stop time of 18 seconds, specified in the Model Configuration Parameters dialog box. The figure shows the block diagram for this model and the simulation output in the Scope block.





You can see that simulation output starts at 3 seconds, as expected. Notice that the time-based Pulse Generator produces a logical `on` for only 2 seconds, after which its output changes to `off` at t = 5 seconds. This is because this block starts computing its output from t = 0 seconds, even though it doesn't output it until the simulation starts at t = 3 seconds. The time-based block depends on simulation time for its output.

The sample-based block outputs a pulse of 5 seconds `on` followed by 5 seconds `off`. In this case, the blocks output does not depend on simulation time, and starts only when the simulation starts.

Characteristics

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

Introduced before R2006a

Push Button

Set value of tunable parameter or variable by holding button

Library

Dashboard

Description 

The `Push Button` block enables you to control tunable parameters and variables in your model during simulation. The block sets a value of a tunable parameter or variable by selecting and holding the button. When the push button is not selected, the value is set back to the default value.

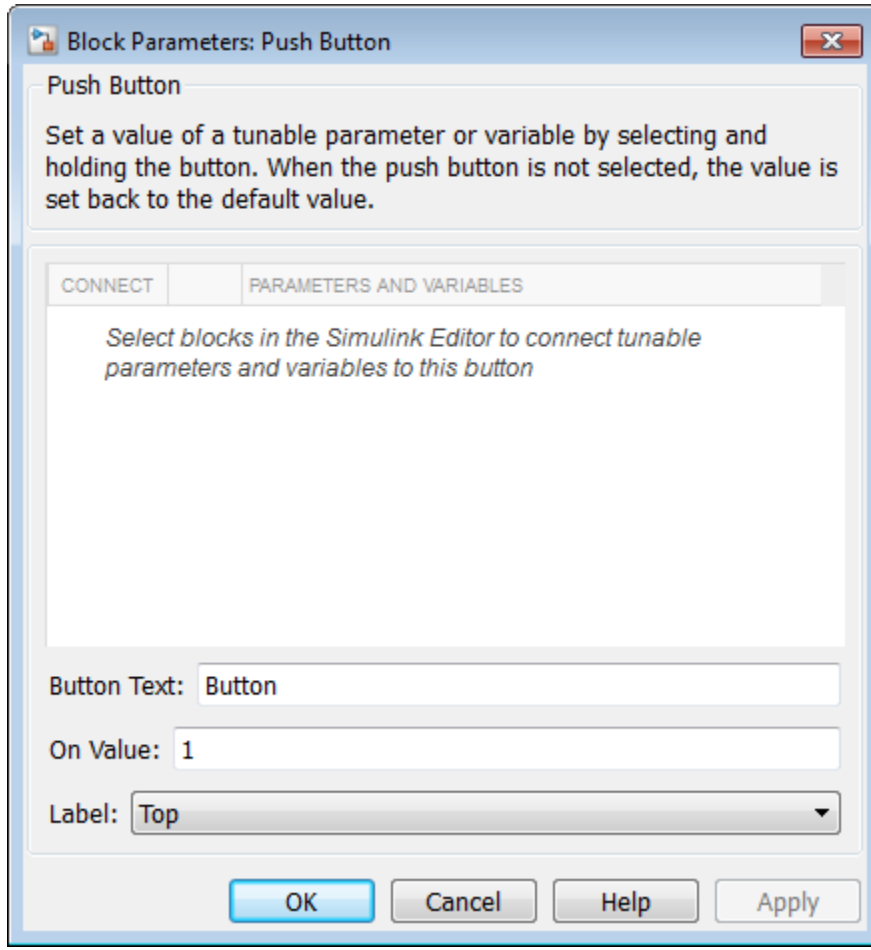
To control a tunable parameter or variable using the `Push Button` block, double-click the `Push Button` block to open the dialog box. Select a block in the model canvas. The tunable parameter or variable appears in the dialog box **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable to the block.

Limitations

The `Push Button` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.
Parameters that index a variable array do not appear in the **Connection** table.	For example, a block parameter specified using the variable `engine(1)` will not appear in the table because the parameter uses an index of the variable `engine`, which is not a scalar variable. To make the parameter appear in the **Connection** table, change the block parameter field to a scalar variable, such as `engine_1`.

Parameters and Dialog Box



Connection

Select a block to connect and control a tunable parameter or variable.

To control a tunable parameter or variable, select a block in the model. The tunable parameter or variable appears in the **Connection** table. Select the option button next

to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable.

Settings

The table has a row for the tunable parameter or variable connected to the block. If there are no tunable parameters or variables selected in the model or the block is not connected to any tunable parameters or variables, then the table is empty.

Button Text

The push button text label.

Settings

Default: Button

Specify this label as a string.

On Value

The value when the button is pushed and held.

Settings

Default: 1

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

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom

Show the label at the bottom of the block.

Hide

Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

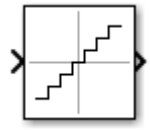
Introduced in R2015a

Quantizer

Discretize input at specified interval

Library

Discontinuities

Description 

The Quantizer block passes its input signal through a stair-step function so that many neighboring points on the input axis are mapped to one point on the output axis. The effect is to quantize a smooth signal into a stair-step output. The output is computed using the round-to-nearest method, which produces an output that is symmetric about zero.

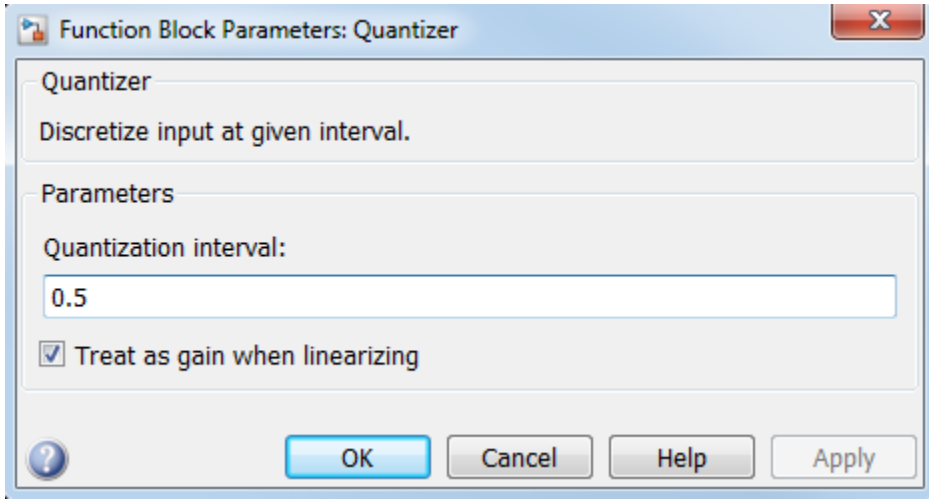
```
y = q * round(u/q)
```

where `y` is the output, `u` the input, and `q` the **Quantization interval** parameter.

Data Type Support

The Quantizer block accepts and outputs real or complex signals of type `single` or `double`. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters 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.

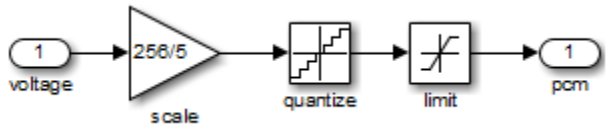
Sample time

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

Examples

The `sldemo_boiler` model shows how you can use the `Quantizer` block.

The `Quantizer` block appears in the `Boiler Plant model/digital thermometer/ADC` subsystem.



The ADC subsystem digitizes the input analog voltage by:

- Multiplying the analog voltage by 256/5 with the `Gain` block
- Rounding the value to integer floor with the `Quantizer` block
- Limiting the output to a maximum of 255 (the largest unsigned 8-bit integer value) with the `Saturation` block

For more information, see “Explore the Fixed-Point "Bang-Bang Control" Model” in the Stateflow documentation.

Characteristics

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

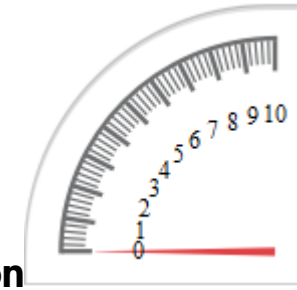
Introduced before R2006a

Quarter Gauge

Display input value on ninety degree scale

Library

Dashboard



Description

The `Quarter Gauge` block displays connected signals during simulation on a circular ninety degree gauge.

To view data from a signal on the `Quarter Gauge` block, double-click the `Quarter Gauge` block to open the dialog box. Select a signal in the model canvas. The signal appears in the dialog box **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal to the block.

You can modify the tick range by modifying the **Minimum**, **Maximum**, and **Tick Interval** values.

You can also add scale colors that appear on the outside of the `Quarter Gauge` block scale using the **Scale Colors** table.

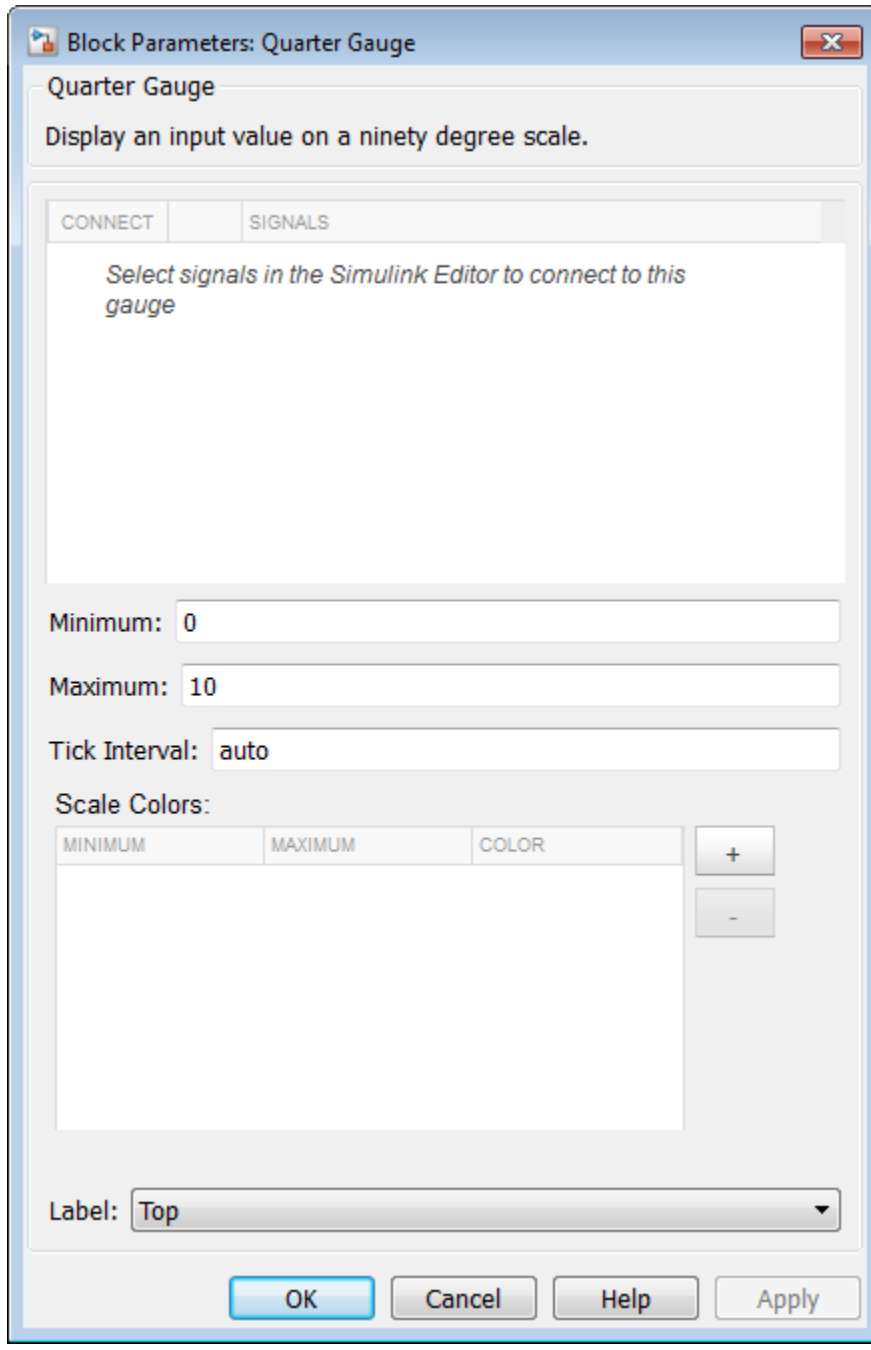
Limitations

The `Quarter Gauge` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.

If you turn off streaming for a signal connected to any dashboard gauge, then the connection shows as broken. Signal data does not stream to the block. To view signal data again, double-click the gauge and reconnect the signal.

The External simulation mode is not supported for the `Quarter Gauge` block.


Block Parameters: Quarter Gauge
Quarter Gauge
Display an input value on a ninety degree scale.
CONNECT
SIGNALS
Select signals in the Simulink Editor to connect to this gauge
Minimum: 0
Maximum: 10
Tick Interval: auto
Scale Colors:
MINIMUM
MAXIMUM
COLOR
+
-
Label: Top
OK
Cancel
Help
Apply

Connection

Select a signal to connect and display.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

Settings

The table has a row for the signal connected to the block. If there are no signals selected in the model or the block is not connected to any signals, then the table is empty.

Minimum

Minimum tick mark value.

Settings

Default: 0

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

Dependencies

The **Minimum** tick value must be less than the **Maximum** tick value.

Maximum

Maximum tick mark value.

Settings

Default: 100

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

Dependencies

The **Maximum** tick value must be greater than the **Minimum** tick value.

Tick Interval

Interval between major tick marks.

Settings

Default: `auto`

Specify this number as a finite, real, positive, integer, scalar value. Specify as `auto` for the block to adjust the tick interval automatically.

Scale Colors

Specify ranges of color bands on the outside of the scale. Specify the minimum and maximum color range to display on the gauge.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

Introduced in R2015a

Ramp

Generate constantly increasing or decreasing signal

Library

Sources

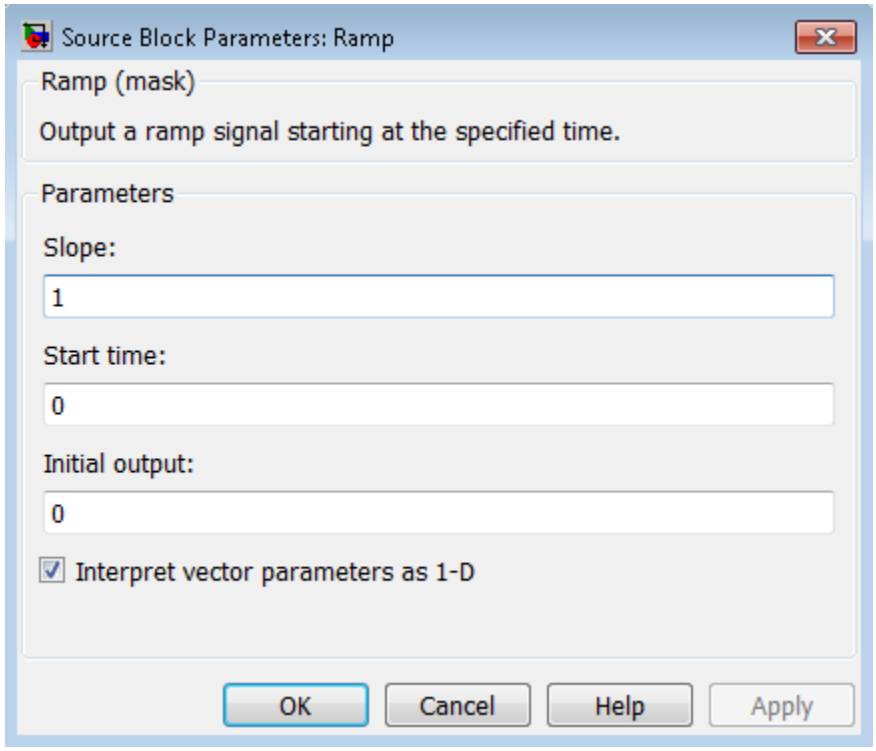
Description

The Ramp block generates a signal that starts at a specified time and value and changes by a specified rate. The block's **Slope**, **Start time**, and **Initial output** parameters determine the characteristics of the output signal. All must have the same dimensions after scalar expansion.

Data Type Support

The Ramp block outputs signals of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Slope

Specify the rate of change of the generated signal. The default is `1`.

Start time

Specify the time at which the block begins generating the signal. The default is `0`.

Initial output

Specify the initial value of the output signal. The default is `0`.

Interpret vector parameters as 1-D

If you select this option and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise, the output dimensionality is the same as that of the other parameters.

See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Examples

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

- `sldemo_VariableTransportDelay_pipe`

Characteristics

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

Introduced before R2006a

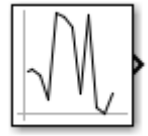
Random Number

Generate normally distributed random numbers

Library

Sources

Description



The Random Number block generates normally distributed random numbers. To generate uniformly distributed random numbers, use the `Uniform Random Number` block.

You can generate a repeatable sequence using any Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts. By default, the block produces a sequence that has a mean of 0 and a variance of 1. To generate a vector of random numbers with the same mean and variance, specify the **Seed** parameter as a vector.

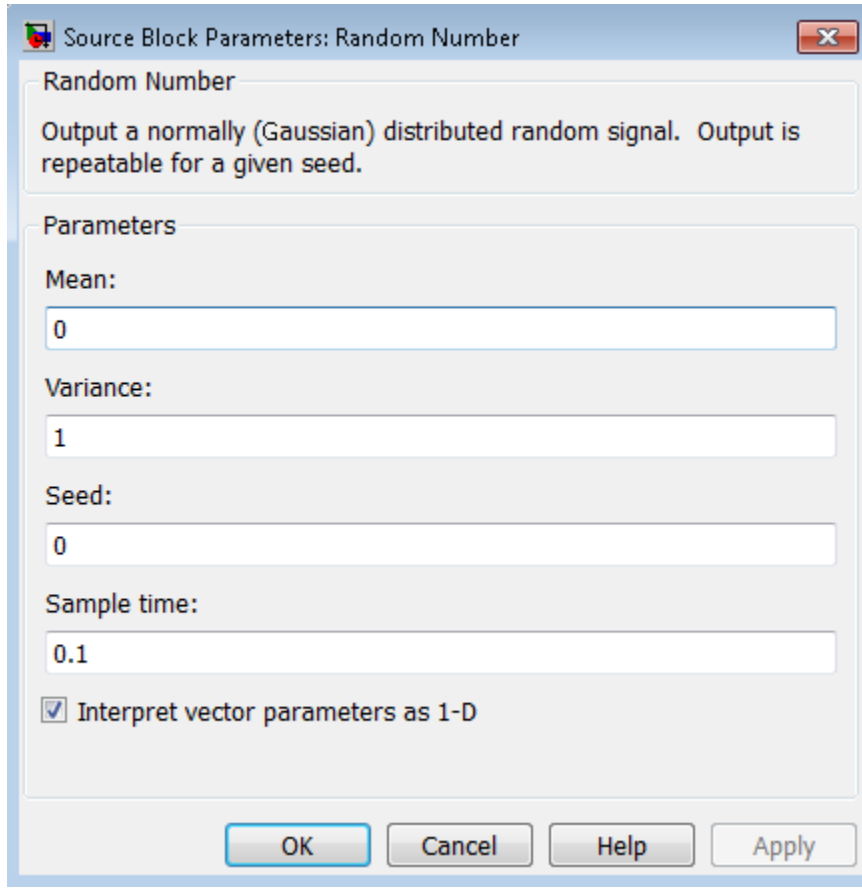
Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the `Band-Limited White Noise` block.

The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the **Interpret vector parameters as 1-D** check box and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensionality as the parameters.

Data Type Support

The Random Number block outputs a real signal of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters 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

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

The generator algorithm is identical to the one used in MATLAB Version 4.0 by the `rand` and `randn` functions. For details on the `mcg16807` algorithm, see "Choosing a Random Number Generator" in the MATLAB documentation.

To use other algorithms supported by MATLAB in a Simulink model, generate a stream of random numbers in MATLAB, and store the output as a .mat file. Use this .mat file as the random number input for your simulation. For more information, see "Creating and Controlling a Random Number Stream". To create multiple independent streams using MATLAB, see "Multiple streams".

Note: Using multiple seeds to generate multiple parallel independent streams for a generator algorithm is not recommended for the `mcg16807` algorithm. Instead, use the method described above.

See Also

`Uniform Random Number`

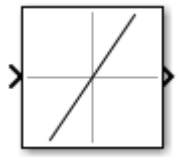
Introduced before R2006a

Rate Limiter

Limit rate of change of signal

Library

Discontinuities



Description

The Rate Limiter block limits the first derivative of the signal passing through it. The output changes no faster than the specified limit. The derivative is calculated using this equation:`

$$rate = \frac{u(i) - y(i-1)}{t(i) - t(i-1)}$$

u(i) and *t(i)* are the current block input and time, and *y(i–1)* and *t(i–1)* are the output and time at the previous step. The output is determined by comparing *rate* to the **Rising slew rate** and **Falling slew rate** parameters:

- If *rate* is greater than the **Rising slew rate** parameter (*R*), the output is calculated as

  $$y(i) = \Delta t \cdot R + y(i-1).$$

- If *rate* is less than the **Falling slew rate** parameter (*F*), the output is calculated as

  $$y(i) = \Delta t \cdot F + y(i-1).$$

- If *rate* is between the bounds of *R* and *F*, the change in output is equal to the change in input:

$$y(i) = u(i)$$

When the block is running in continuous mode (for example, **Sample time mode** is `inherited` and **Sample time** of the driving block is zero), the **Initial condition** is ignored. The block output at `t = 0` is equal to the initial input:

$$y(0) = u(0)$$

When the block is running in discrete mode (for example, **Sample time mode** is `inherited` and **Sample time** of the driving block is nonzero), the **Initial condition** is preserved:

$$y(-1) = Ic$$

where *Ic* is the initial condition. The block output at `t = 0` is calculated as if *rate* is outside the bounds of *R* and *F*. For `t = 0`, *rate* is calculated as follows:

$$rate = \frac{u(0) - y(-1)}{sampletime}$$

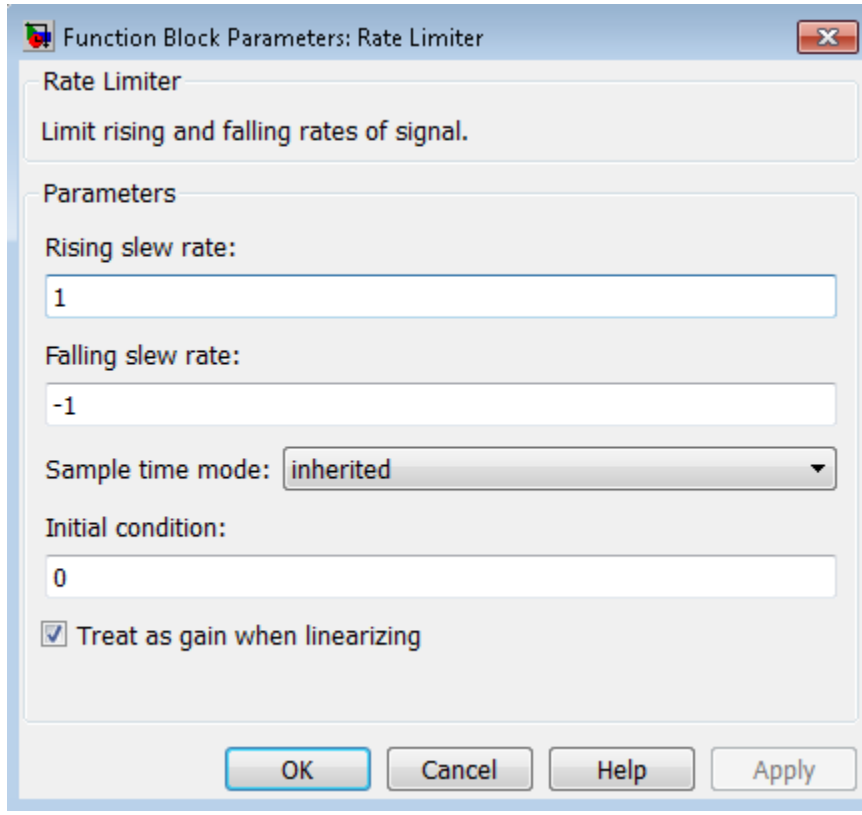
Note: You cannot use a Rate Limiter block inside a Triggered Subsystem. Use the `Rate Limiter Dynamic` block instead.

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



Rising slew rate

Specify the limit of the derivative of an increasing input signal. This parameter is tunable for fixed-point inputs.

Falling slew rate

Specify the limit of the derivative of a decreasing input signal. This parameter is tunable for fixed-point inputs.

Sample time mode

Specify the sample time mode, `continuous` or `inherited` from the driving block.

Initial condition

Set the initial output of the simulation. Simulink software does not allow you to set the initial condition of this block to `inf` or `NaN`.

Treat as gain when linearizing

Linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the linearization commands to treat the gain as 1; otherwise, the commands treat the gain as 0.

Characteristics

Data Types	Double | Single| Base Integer | Fixed-Point
Sample Time	Continuous or inherited (specified in the **Sample time mode** parameter)
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Rate Limiter Dynamic`

Introduced before R2006a

Rate Limiter Dynamic

Limit rising and falling rates of signal

Library

Discontinuities

Description

The Rate Limiter Dynamic block limits the rising and falling rates of the signal.

- The external signal `up` sets the upper limit on the rising rate of the signal.
- The external signal `lo` sets the lower limit on the falling rate of the signal.

Follow these guidelines when using the Rate Limiter Dynamic block:

- Ensure that the data types of `up` and `lo` are the same as the data type of the input signal `u`.

  When the lower limit uses a signed type and the input signal uses an unsigned type, the output signal keeps increasing regardless of the input and the limits.
- Use a fixed-step solver to simulate models that contain this block.

  Because the Rate Limiter Dynamic block does not support continuous sample time, simulation with a variable-step solver causes an error.

Data Type Support

The Rate Limiter Dynamic block accepts input signals of the following data types:

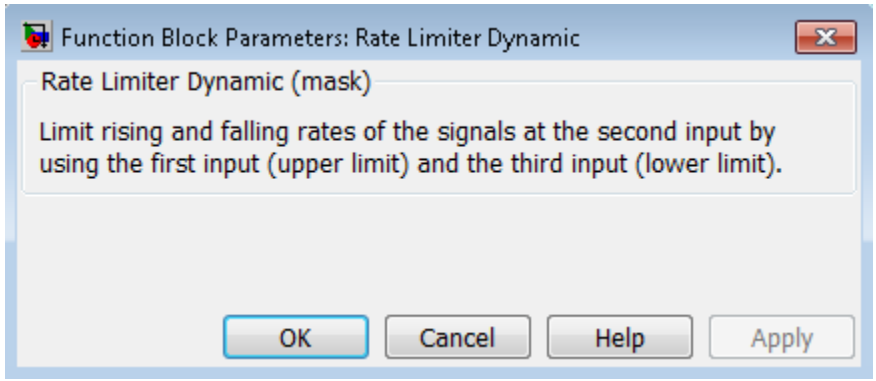
- Floating point

- Built-in integer
- Fixed point

The data type of the output signal `Y` matches the data type of the input signal `u`.

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

Parameters and Dialog Box



Characteristics

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

See Also

`Rate Limiter`

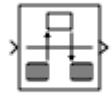
Introduced before R2006a

Rate Transition

Handle transfer of data between blocks operating at different rates

Library

Signal Attributes



Description

Transition Handling Options

The Rate Transition block transfers data from the output of a block operating at one rate to the input of a block operating at a different rate. Use the block parameters to trade data integrity and deterministic transfer for faster response or lower memory requirements. To learn about data integrity and deterministic data transfer, see "Data Transfer Problems" 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**

Transition Handling Options	Block Parameter Settings
	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 1-1537 and "Effects of Asynchronous Sample Times" on page 1-1539)
- 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)

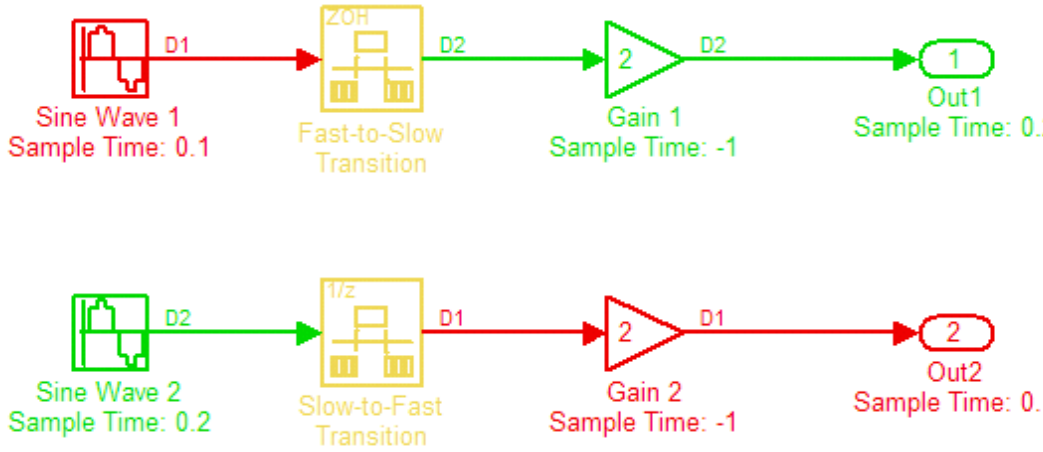
Block Labels

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

Label	Block Behavior
`NoOp`	Does nothing
`Mixed`	Expands to multiple blocks with different behaviors

The block behavior label shows the method that ensures safe transfer of data between tasks operating at different rates. You can use the sample-time colors feature (see "View Sample Time Information" in the Simulink documentation) to display the relative rates that the block bridges. Consider, for example, the following model:



Sample-time colors and the block behavior label show that the Rate Transition block at the top of the diagram acts as a zero-order hold in a fast-to-slow transition and the bottom Rate Transition block acts as a unit delay in a slow-to-fast transition.

For more information, see "Handle Rate Transitions" 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)		

Block Settings		Block Label		
Rate Transition	Conditions for Rate Transition Block	With Data Integrity and Determinism	With Only Data Integrity	Without Data Integrity or Determinism
KEY • `inTs, outTs`: Sample times of input and output ports, respectively • `inTsOffset, outTsOffset`: Sample time offsets of input and output ports, respectively • `N`: Integer value > 1				

When you select the **Block reduction** parameter on the **All Parameters** tab 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			

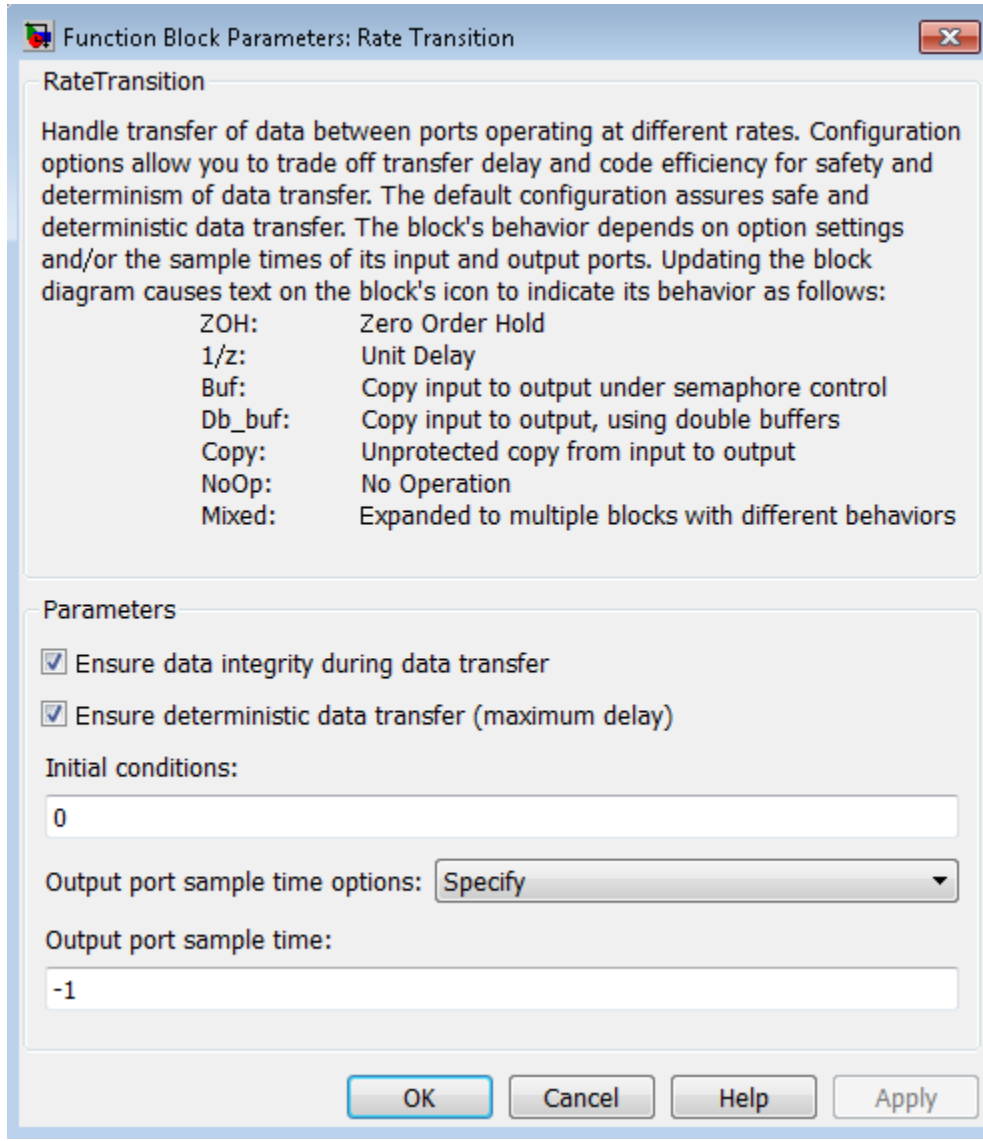
Data Type Support

The Rate Transition block accepts most signals that Simulink supports, including fixed-point and enumerated data types. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

However, do not use the Rate Transition block with frame-based signals. For rate transitions with such signals, use one of these blocks from the DSP System Toolbox instead:

- Buffer
- Unbuffer
- CIC Decimation
- CIC Interpolation
- FIR Decimation
- FIR Interpolation
- Downsample
- Upsample

Parameters 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), depending on the priority of input rate and output rate, the generated code uses a proper algorithm using single or multiple buffers to protect data integrity during data transfer.

Otherwise, the Rate Transition block is either reduced or generates code using a copy operation to effect the data transfer. This unprotected mode consumes less memory. But the copy operation is also interruptible, which can lead to loss of data integrity during data transfers. Select this check box if you want the generated code to operate with maximum responsiveness (i.e., nondeterministically) and data integrity. For more information, see "Rate Transition Block Options" 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 "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

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

Sample Time	This block supports discrete-to-discrete transitions
Direct Feedthrough	No, for slow-to-fast transitions for which you select the **Ensure data integrity during data transfer** check box. Yes, otherwise.
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Real-Imag to Complex

Convert real and/or imaginary inputs to complex signal

Library

Math Operations

Description

The Real-Imag to Complex block converts real and/or imaginary inputs to a complex-valued output signal.

The inputs can both be arrays (vectors or matrices) of equal dimensions, or one input can be an array and the other a scalar. If the block has an array input, the output is a complex array of the same dimensions. The elements of the real input map to the real parts of the corresponding complex output elements. The imaginary input similarly maps to the imaginary parts of the complex output signals. If one input is a scalar, it maps to the corresponding component (real or imaginary) of all the complex output signals.

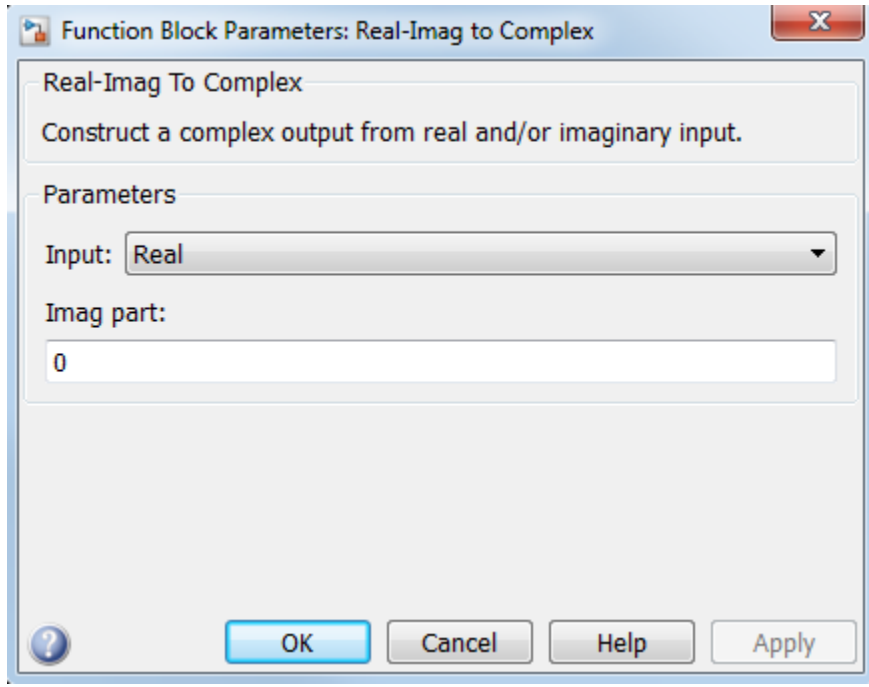
Data Type Support

The block accepts input signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

Complex fixed-point signals must have trivial slope and zero bias. For more information about support for fixed-point data types, see “Scaling” in the Fixed-Point Designer documentation.

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

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

Characteristics

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

See Also

`Complex to Real-Imag`

Introduced before R2006a

Relational Operator

Perform specified relational operation on inputs

Library

Logic and Bit Operations

Description

Two-Input Mode

By default, the Relational Operator block compares two inputs using the **Relational operator** parameter that you specify. The first input corresponds to the top input port and the second input to the bottom input port. (See "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	`isInf`	FALSE
	`isNaN`	FALSE
	`isFinite`	TRUE

Data Type	Operation	Block Output
• Built-in integer		

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

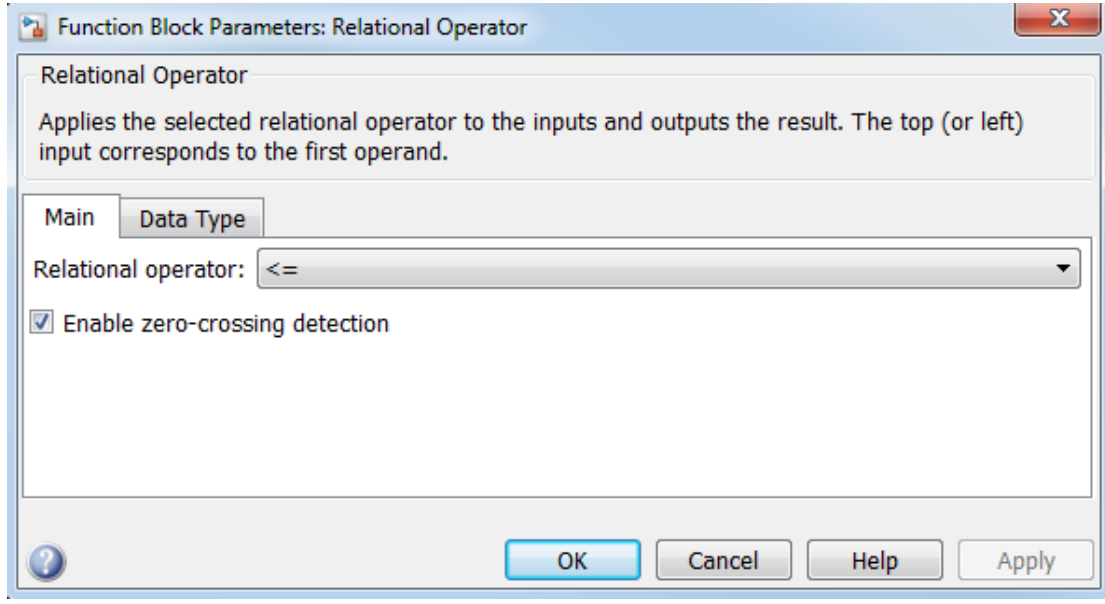
Data Type Support

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`.

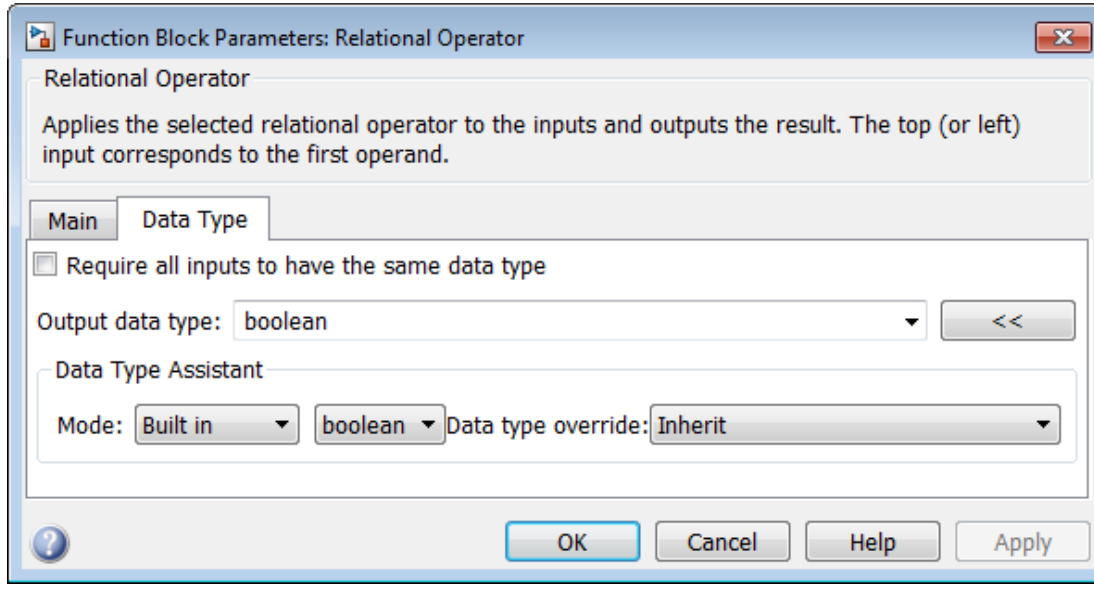
For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

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



The **Data Type** pane of the Relational Operator block dialog box appears as follows:



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 “Control Signal Data Types”.

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 6-101 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

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Sample time

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

Require all inputs to have the same data type

Require that all inputs have the same data type.

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 6-101 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 6-101 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 6-101 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.

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.

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

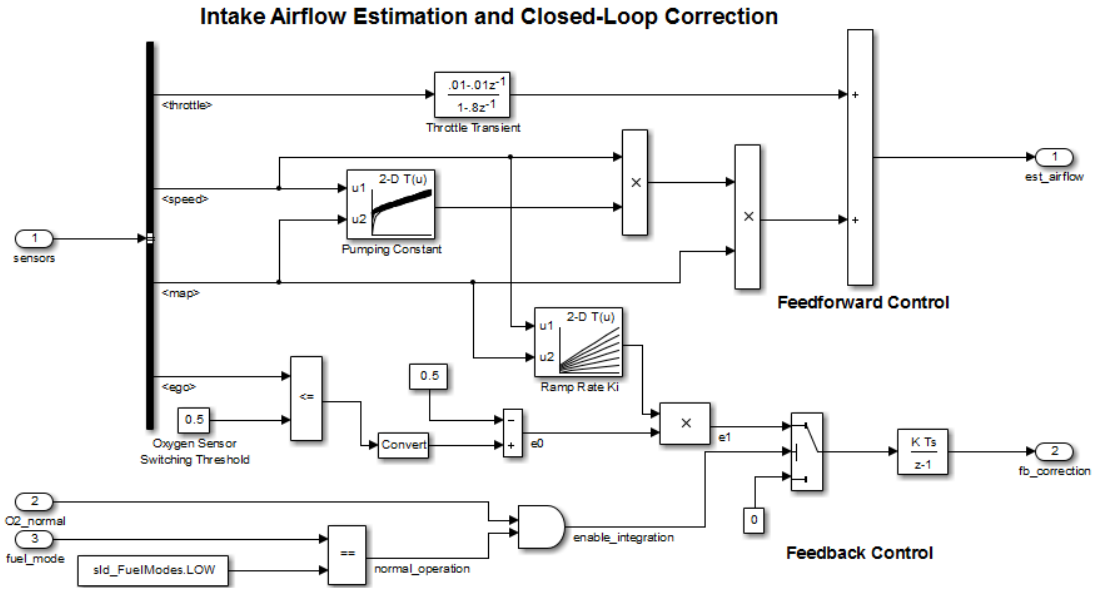
See “Block-Specific Parameters” on page 6-101 for the command-line information.

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:



Both Relational Operator blocks operate in two-input mode.

The block that uses this operator...	Compares...
<=	The value of the oxygen sensor to the threshold value, 0.5
==	The value of the fuel mode to the ideal value, `sld_FuelModes.LOW`

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes

Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

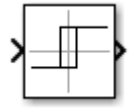
Introduced before R2006a

Relay

Switch output between two constants

Library

Discontinuities

Description 

The Relay block allows its output to switch between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.

The **Switch on point** value must be greater than or equal to the **Switch off point**. Specifying a **Switch on point** value greater than the **Switch off point** models hysteresis, whereas specifying equal values models a switch with a threshold at that value.

Note: When the initial input falls *between* the **Switch off point** and **Switch on point** values, the initial output is the value when the relay is off.

Data Type Support

The Relay block accepts signals of the following data types:

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

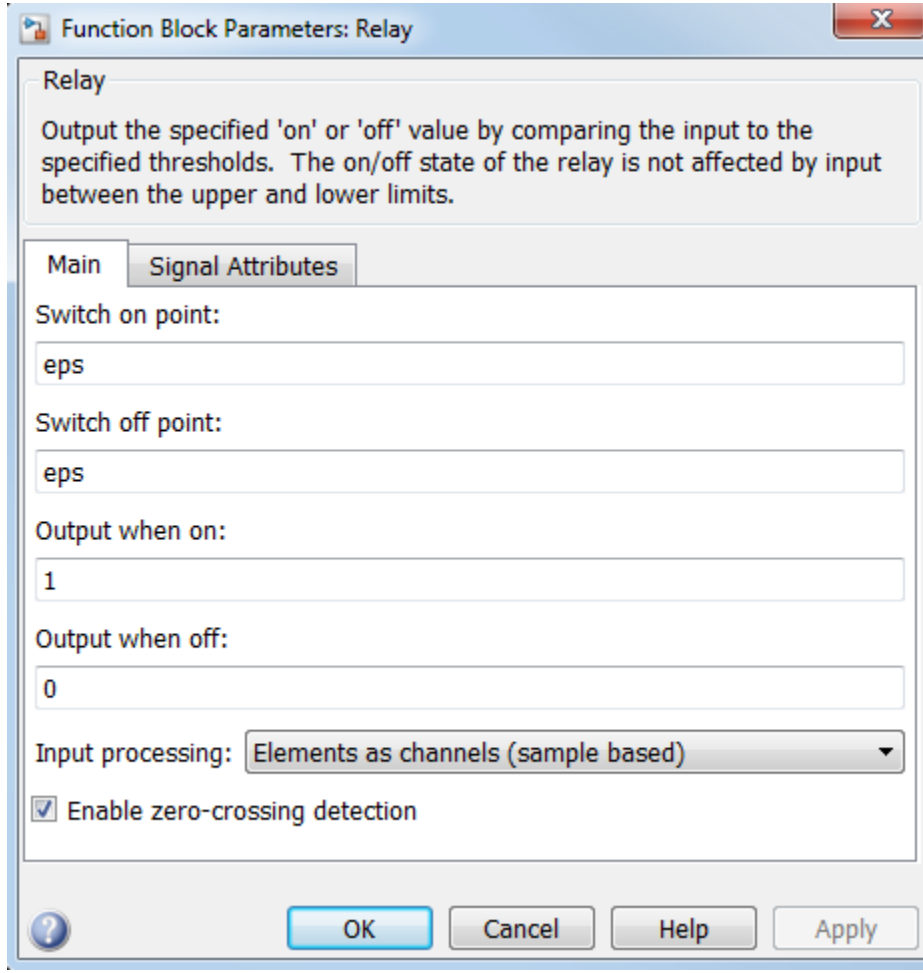
- Enumerated (output only)

If **Output when on** or **Output when off** is an enumerated value, both must be of the same enumerated type as the output.

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

Parameters and Dialog Box

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



Switch on point

The “on” threshold for the relay. The **Switch on point** parameter is converted to the input data type offline using round-to-nearest and saturation.

Switch off point

The “off” threshold for the relay. The **Switch off point** parameter is converted to the input data type offline using round-to-nearest and saturation.

Output when on

The output when the relay is on.

Output when off

The output when the relay is off.

Input processing

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

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

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

  For more information, see "Sample- and Frame-Based Concepts" 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.

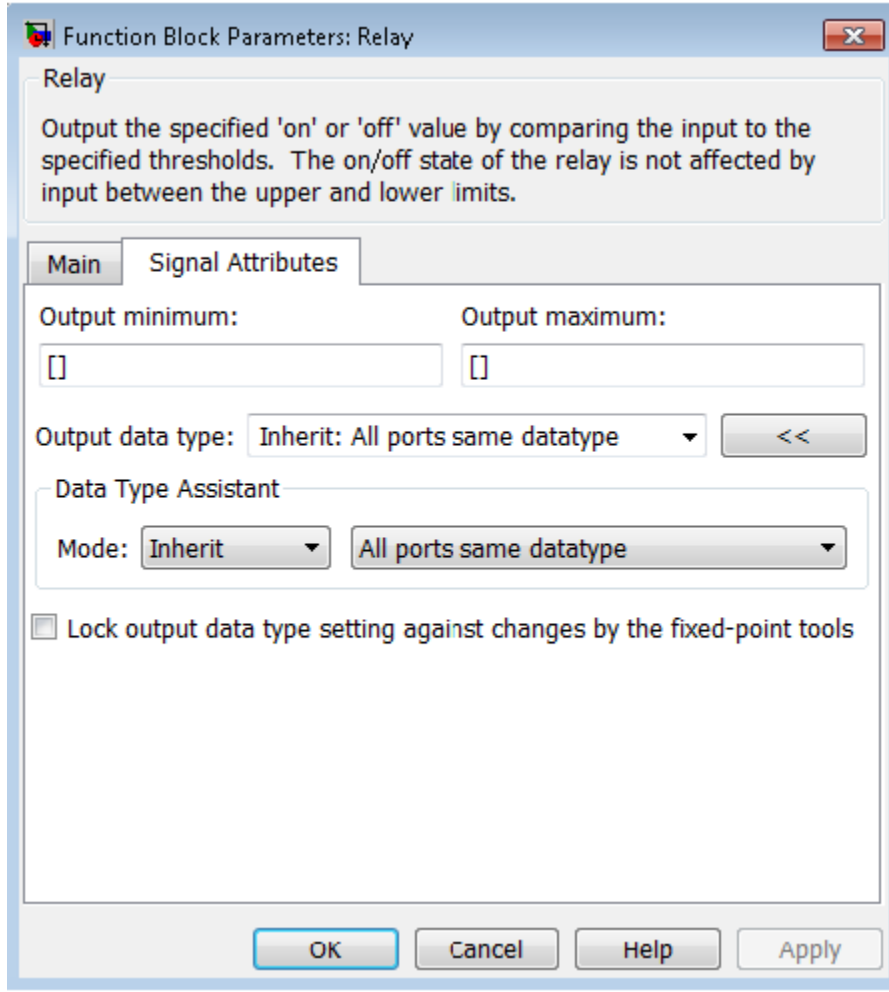
Enable zero-crossing detection

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

Sample time

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

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



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 “Control Block Parameter Value Ranges”)
- 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:

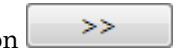
- Parameter range checking (see "Control Block Parameter Value Ranges")
- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

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 "Control Signal Data Types" in Simulink User's Guide for more information.

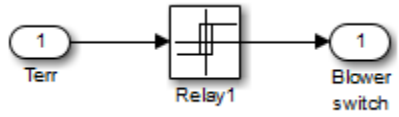
Lock output data type setting against changes by the fixed-point tools

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

Examples

The `sldemo_househeat` model shows how you can use the `Relay` block.

The `Relay` block appears in the Thermostat subsystem.



The thermostat allows fluctuations of 5 degrees Fahrenheit above or below the desired room temperature. If air temperature drops below 65 degrees Fahrenheit, the thermostat turns on the heater. The Relay block outputs a value of 1 to turn on the heater and a value of 0 to turn off the heater.

Characteristics

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

Introduced before R2006a

Repeating Sequence

Generate arbitrarily shaped periodic signal

Library

Sources

Description

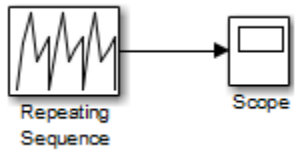
The Repeating Sequence block outputs a periodic scalar signal having a waveform that you specify using the **Time values** and **Output values** parameters. The **Time values** parameter specifies a vector of output times. The **Output values** parameter specifies a vector of signal amplitudes at the corresponding output times. Together, the two parameters specify a sampling of the output waveform at points measured from the beginning of the interval over which the waveform repeats (the period of the signal).

By default, both parameters are `[0 2]`. These default settings specify a sawtooth waveform that repeats every `2` seconds from the start of the simulation and has a maximum amplitude of `2`.

Algorithm

The block sets the input period as the difference between the first and last value of the **Time values** parameter. The output at any time `t` is the output at time `t = t-n*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.

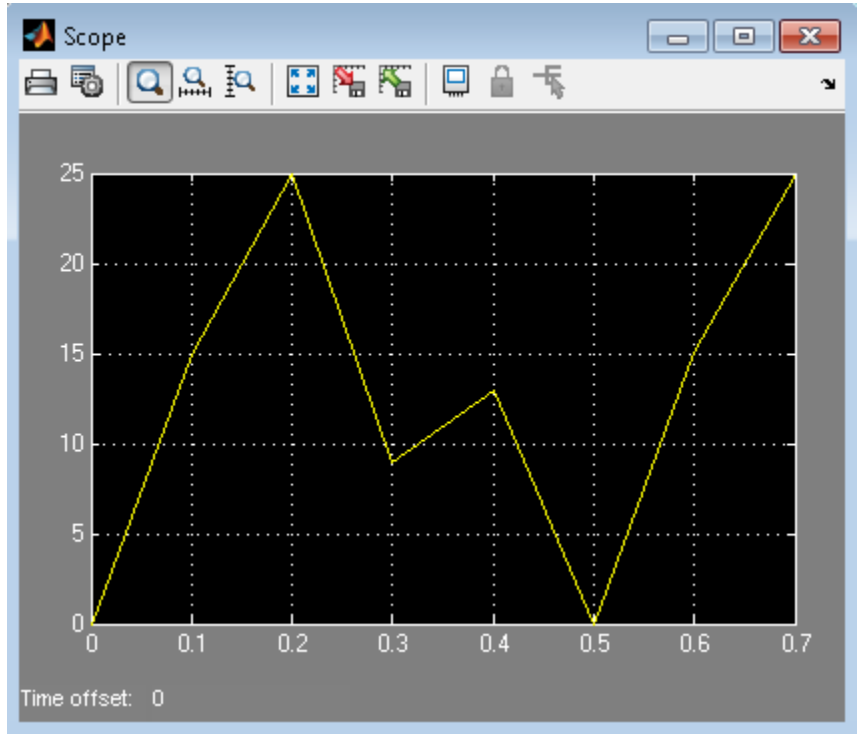
In the following model, the Repeating Sequence block defines **Time values** as [0:0.1:0.5] and **Output values** as [0 15 25 09 13 17]. The stop time of the simulation is 0.7 second.



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:



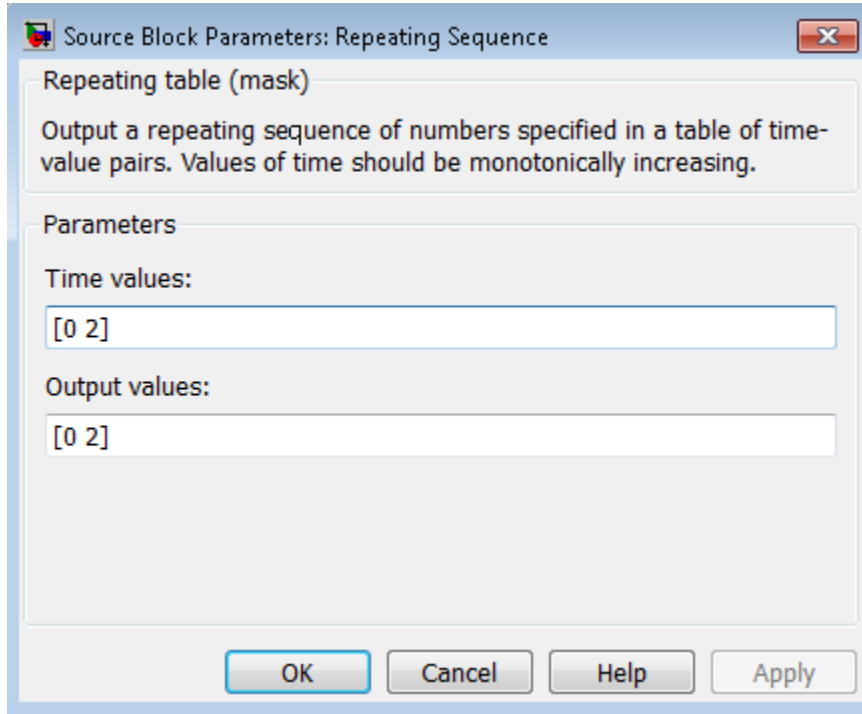
At t = 0.5, the expected output is equal to the output at t = 0, which is 0. Therefore, the last value in the **Output values** vector [0 15 25 09 13 17] does not appear.

Data Type Support

The Repeating Sequence block outputs real signals of type `double`.

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

Parameters and Dialog Box



Time values

Specify a vector of strictly monotonically increasing time values. The default is `[0 2]`.

Output values

Specify a vector of output values. Each element corresponds to the time value in the same column. The default is `[0 2]`.

Examples

The following Simulink examples show how to use the Repeating Sequence block:

- `sldemo_fuelsys`
- `sldemo_hydrod`
- `sldemo_VariableTransportDelay`

Characteristics

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

See Also

`Repeating Sequence Interpolated`, `Repeating Sequence Stair`

Introduced before R2006a

Repeating Sequence Interpolated

Output discrete-time sequence and repeat, interpolating between data points

Library

Sources

Description

The Repeating Sequence Interpolated block outputs a discrete-time sequence and then repeats it. Between data points, the block uses the method you specify for the **Lookup Method** parameter to determine the output.

Data Type Support

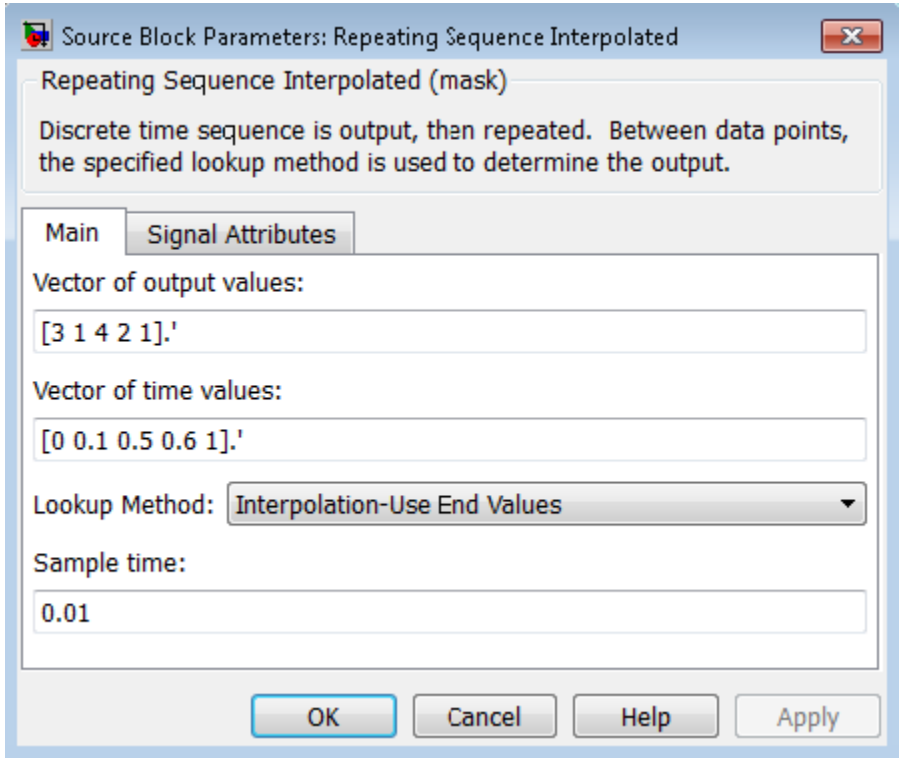
The Repeating Sequence Interpolated block outputs signals of the following data types:

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

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

Parameters and Dialog Box

The **Main** pane of the Repeating Sequence Interpolated block dialog box appears as follows:



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.

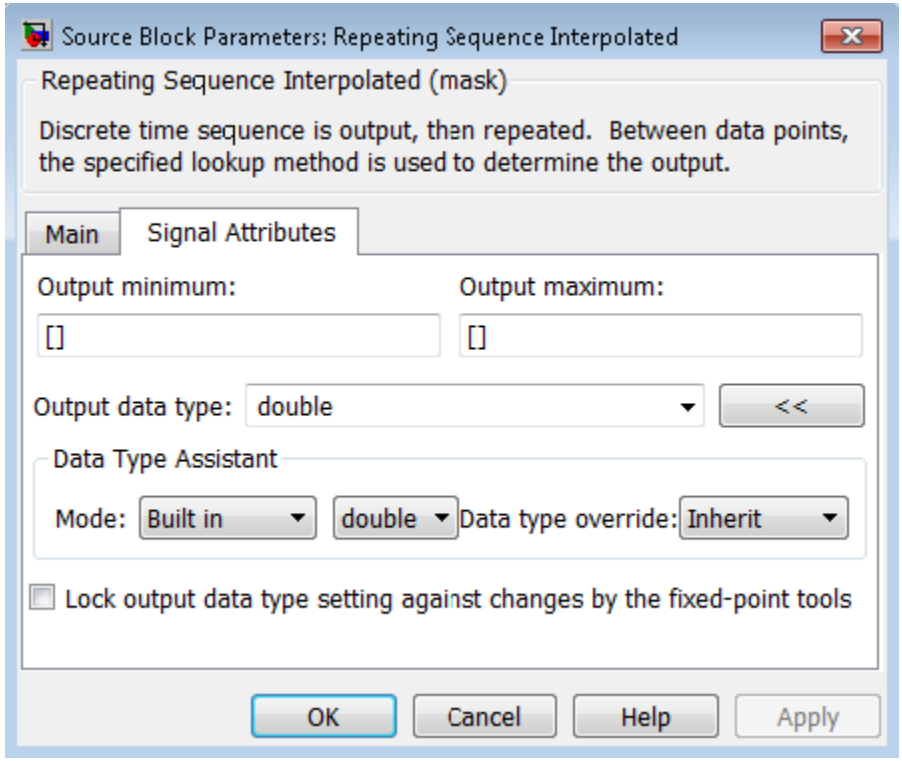
Lookup Method

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:



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 “Control Block Parameter Value Ranges”)
- 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:

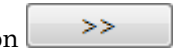
- Parameter range checking (see “Control Block Parameter Value Ranges”)
- Simulation range checking (see “Signal Ranges”)

- Automatic scaling of fixed-point data types

Output data type

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

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

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

See "Control Signal Data Types" 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Repeating Sequence`, `Repeating Sequence Stair`

Introduced before R2006a

Repeating Sequence Stair

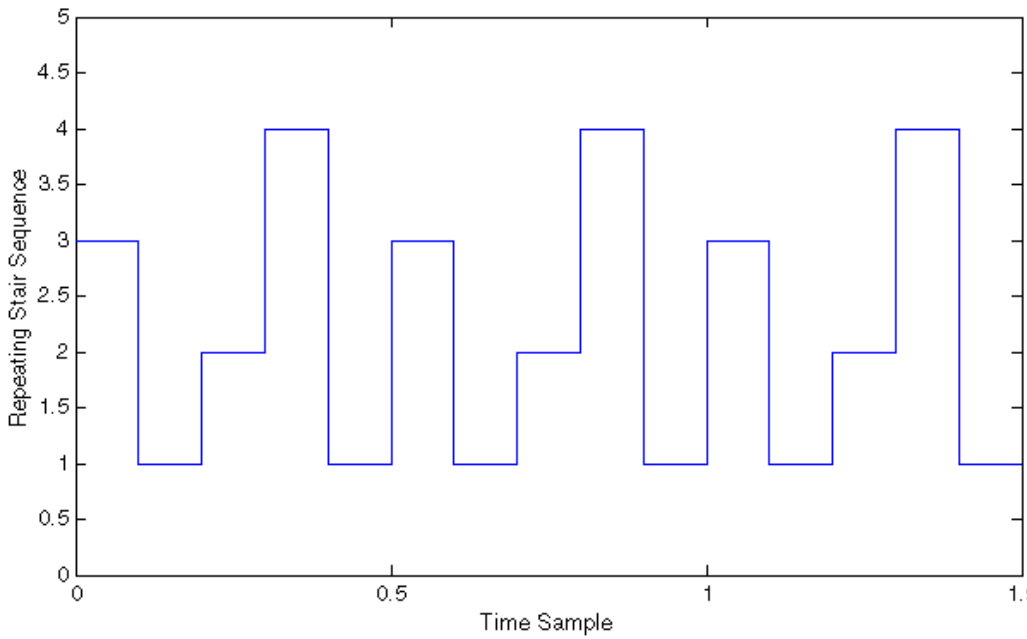
Output and repeat discrete time sequence

Library

Sources

Description

The Repeating Sequence Stair block outputs and repeats a stair sequence that you specify with the **Vector of output values** parameter. For example, you can specify the vector as `[3 1 2 4 1]'`, which produces the following stair sequence:



Data Type Support

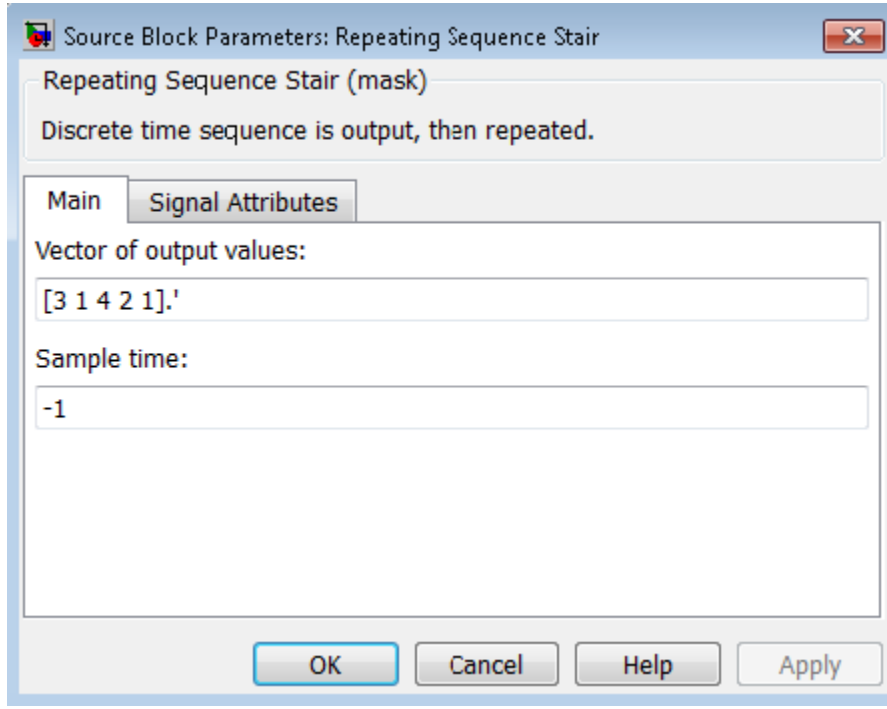
The Repeating Sequence Stair block outputs signals of the following data types:

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

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

Parameters and Dialog Box

The **Main** pane of the Repeating Sequence Stair block dialog box appears as follows:



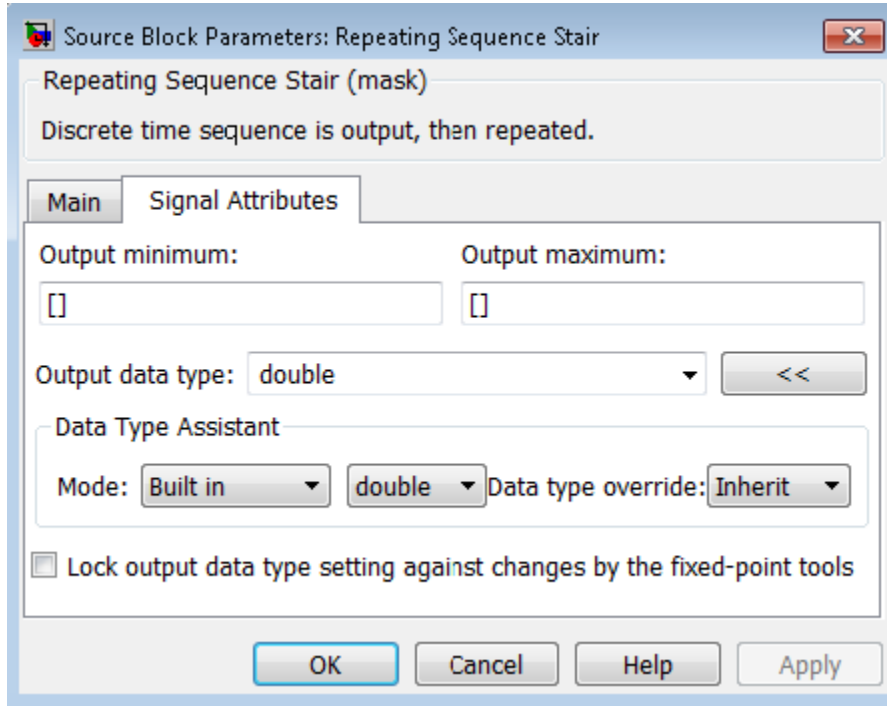
Vector of output values

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:



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

Output maximum

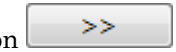
Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

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

Output data type

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

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

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

See "Control Signal Data Types" in 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Repeating Sequence`, `Repeating Sequence Interpolated`

Introduced before R2006a

Reset

Add reset port to subsystem

Library

Ports & Subsystems

Description 

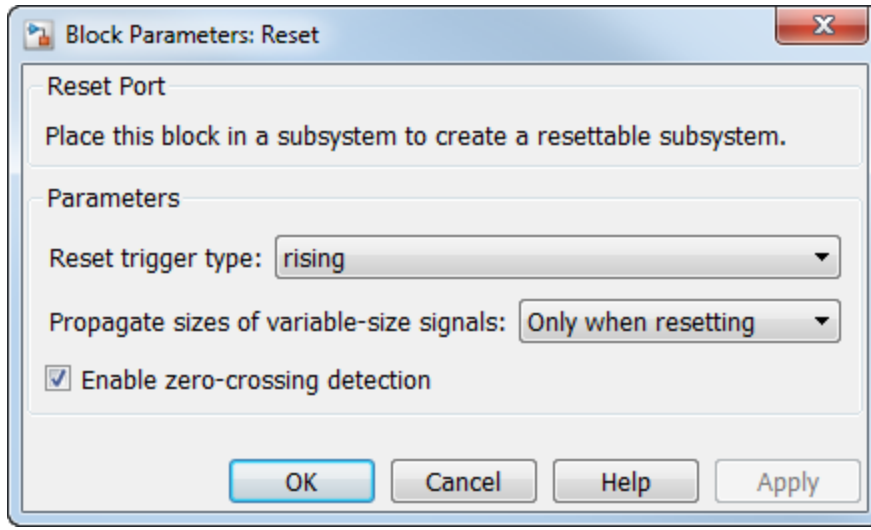
A Reset block is the control port for a resettable subsystem, and it determines when the subsystem states reset to their initial condition. Resettable subsystems contain a Reset block at their root level.

A resettable subsystem resets the blocks inside it when a trigger occurs at the reset port, similar to the reset port of a block. The reset event initializes the states of blocks inside a resettable subsystem to their initial conditions. For more information, see “Conditionally Reset Block States in a Subsystem”.

Data Type Support

The Reset block accepts signals of Simulink numeric and Boolean data types. It also accepts fixed-point signals of type ufix1. For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box



Reset trigger type

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

Settings

Default: `rising`

`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.

Propagate sizes of variable-size signals

Specify when to propagate a variable-size signal.

Settings

Default: `Only when resetting`

`Only when resetting`

Propagates variable-size signals only when resetting the resettable subsystem. When you select this option, sample time must be periodic.

`During execution`

Propagates variable-size signals at each time step.

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.

Characteristics

Sample Time	Determined by the signal at the reset port
Dimensionalized	Yes
Virtual	No

Zero-Crossing Detection	Yes, if enabled

See Also

"Conditionally Reset Block States in a Subsystem"

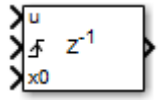
Introduced in R2015a

Resettable Delay

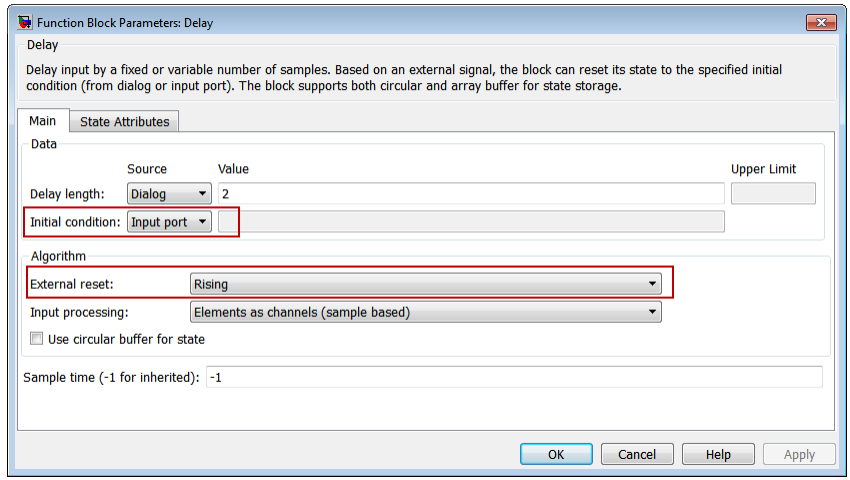
Delay input signal by variable sample period and reset with external signal

Library

Discrete

Description 

The `Resettable Delay` block is a variant of the `Delay` block that has the source of the initial condition set to `Input port` and the external reset algorithm set to `Rising`, by default.



Characteristics

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

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

See Also

`Delay` | `Tapped Delay` | `Unit Delay` | `Variable Integer Delay`

Introduced in R2012b

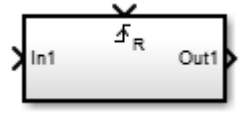
Resettable Subsystem

Represent subsystem whose states reset with external trigger

Library

Ports & Subsystems

Description



This block is preconfigured as a starting point for a resettable subsystem. For more information, see “Conditionally Reset Block States in a Subsystem”.

Characteristics

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

Introduced in R2015a

Reshape

Change dimensionality of signal

Library

Math Operations

Description

The Reshape block changes the dimensionality of the input signal to a dimensionality that you specify, using the block's **Output dimensionality** parameter. For example, you can use the block to change an N-element vector to a 1-by-N or N-by-1 matrix signal, and vice versa.

The **Output dimensionality** parameter lets you select any of the following output options.

Output Dimensionality	Description
1-D array	Converts a multidimensional array to a vector (1-D array) array signal. The output vector consists of the first column of the input matrix followed by the second column, etc. (This option leaves a vector input unchanged.)
Column vector	Converts a vector, matrix, or multidimensional input signal to a column matrix, i.e., an M-by-1 matrix, where M is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.
Row vector	Converts a vector, matrix, or multidimensional input signal to a row matrix, i.e., a 1-by-N matrix where

Output Dimensionality	Description
	N is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.
Customize	Converts the input signal to an output signal whose dimensions you specify, using the **Output dimensions** parameter. The value of the **Output dimensions** parameter can be a one- or multi-element vector. A value of `[N]` outputs a vector of size N. A value of `[M N]` outputs an M-by-N matrix. The number of elements of the input signal must match the number of elements specified by the **Output dimensions** parameter. For multidimensional arrays, the conversion is done along the first dimension.
Derive from reference input port	Creates a second input port, Ref, on the block. Derives the dimensions of the output signal from the dimensions of the signal input to the Ref input port. Selecting this option disables the **Output dimensions** parameter. When you select this parameter, the input signals for both inport ports, U and Ref, must have the same sampling mode (sample-based or frame-based).

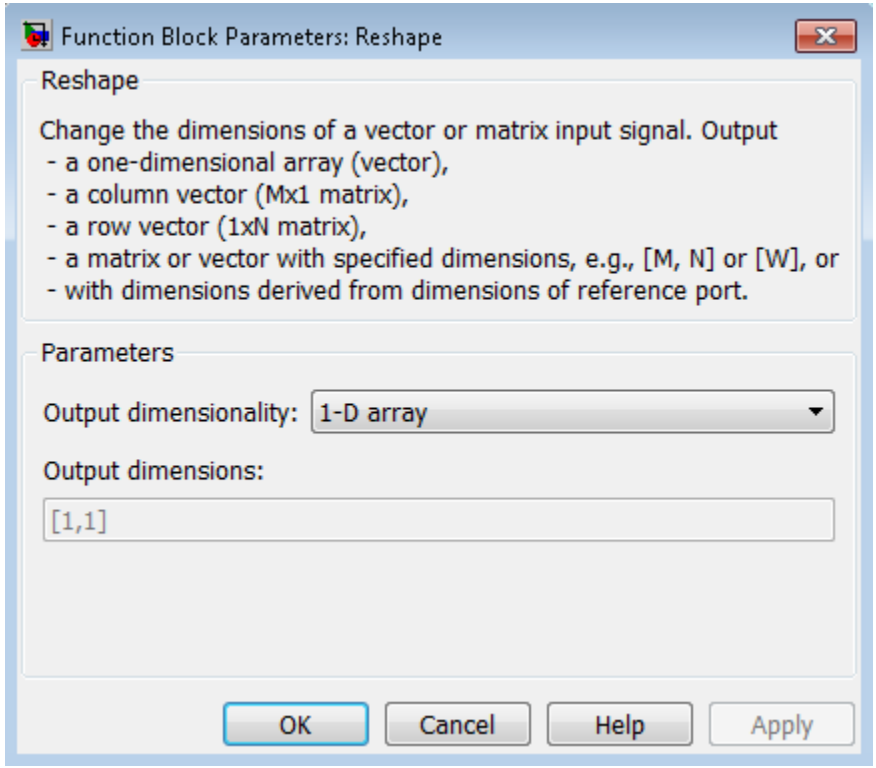
Data Type Support

The Reshape block accepts and outputs signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types.

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

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

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

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

Introduced before R2006a

Rocker Switch

Set on/off values to tune parameters or variables

Library

Dashboard

Description

The `Rocker Switch` block enables you to control tunable parameters and variables in your model during simulation. The block has two states that can be set to two different values.

To control a tunable parameter or variable using the `Rocker Switch` block, double-click the `Rocker Switch` block to open the dialog box. Select a block in the model canvas. The tunable parameter or variable appears in the dialog box **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable to the block.

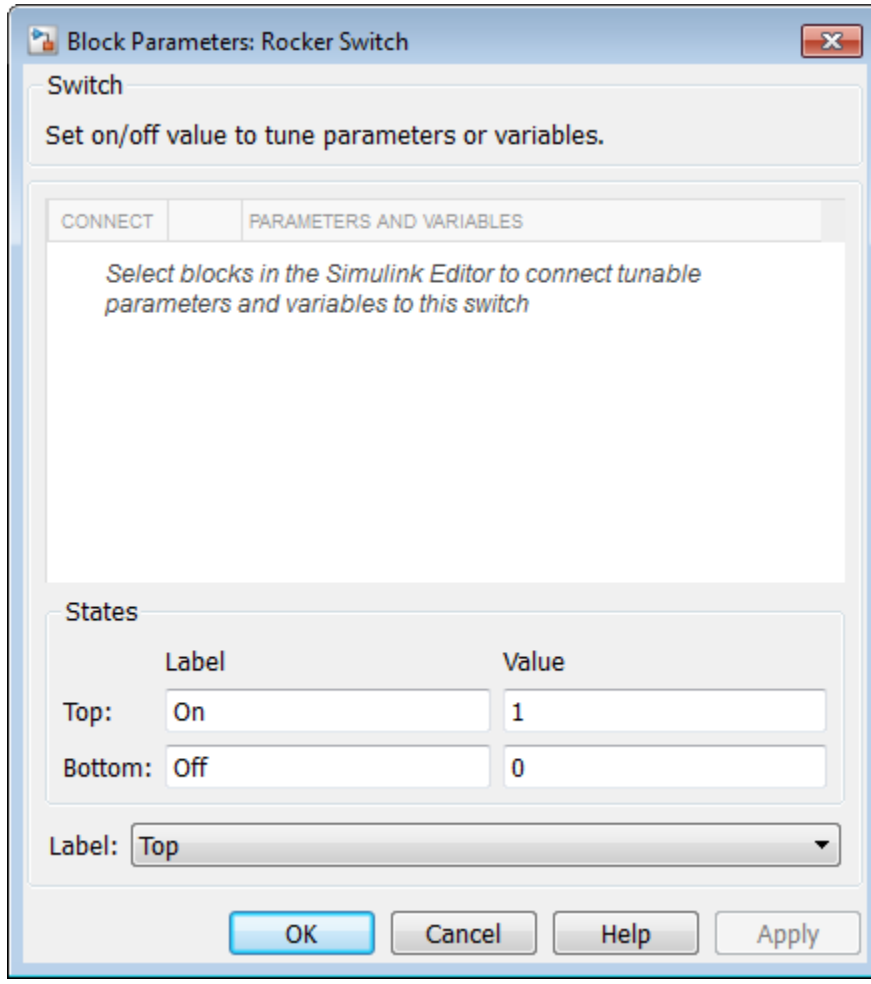
Limitations

The `Rocker Switch` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.
Parameters that index a variable array do not appear in the **Connection** table.	For example, a block parameter specified using the variable `engine(1)` will not appear in the table because the parameter uses an index of the variable `engine`,

Limitation	Workaround
	which is not a scalar variable. To make the parameter appear in the **Connection** table, change the block parameter field to a scalar variable, such as `engine_1`.

Parameters and Dialog Box



Connection

Select a block to connect and control a tunable parameter or variable.

To control a tunable parameter or variable, select a block in the model. The tunable parameter or variable appears in the **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable.

Settings

The table has a row for the tunable parameter or variable connected to the block. If there are no tunable parameters or variables selected in the model or the block is not connected to any tunable parameters or variables, then the table is empty.

States

Switch values and labels.

Settings

Default Labels: `Off` and `On`

Default Values: `0` and `1`

By default, the **Off** state label corresponds to the set value of `0`, and the **On** state label corresponds to the set value of `1`.

The state labels appear on the outside of the switch. You can change the state labels to another text string. You can change the state values to any real value that is between negative `realmax` and positive `realmax`.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top

Show the label at the top of the block.

Bottom

Show the label at the bottom of the block.

Hide

Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

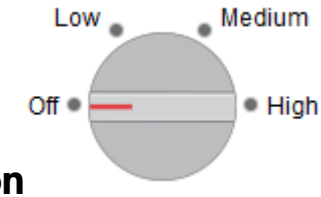
Introduced in R2015a

Rotary Switch

Set value on dial switch to tune parameters or variables

Library

Dashboard



Description

The `Rotary Switch` block enables you to control tunable parameters and variables in your model during simulation.

To control a tunable parameter or variable using the `Rotary Switch` block, double-click the `Rotary Switch` block to open the dialog box. Select a block in the model canvas. The tunable parameter or variable appears in the dialog box **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable to the block.

The state values determine the discrete values generated for the tunable parameter or variable. You can modify the states by editing the **State Value** and **State Label** in the **States** table.

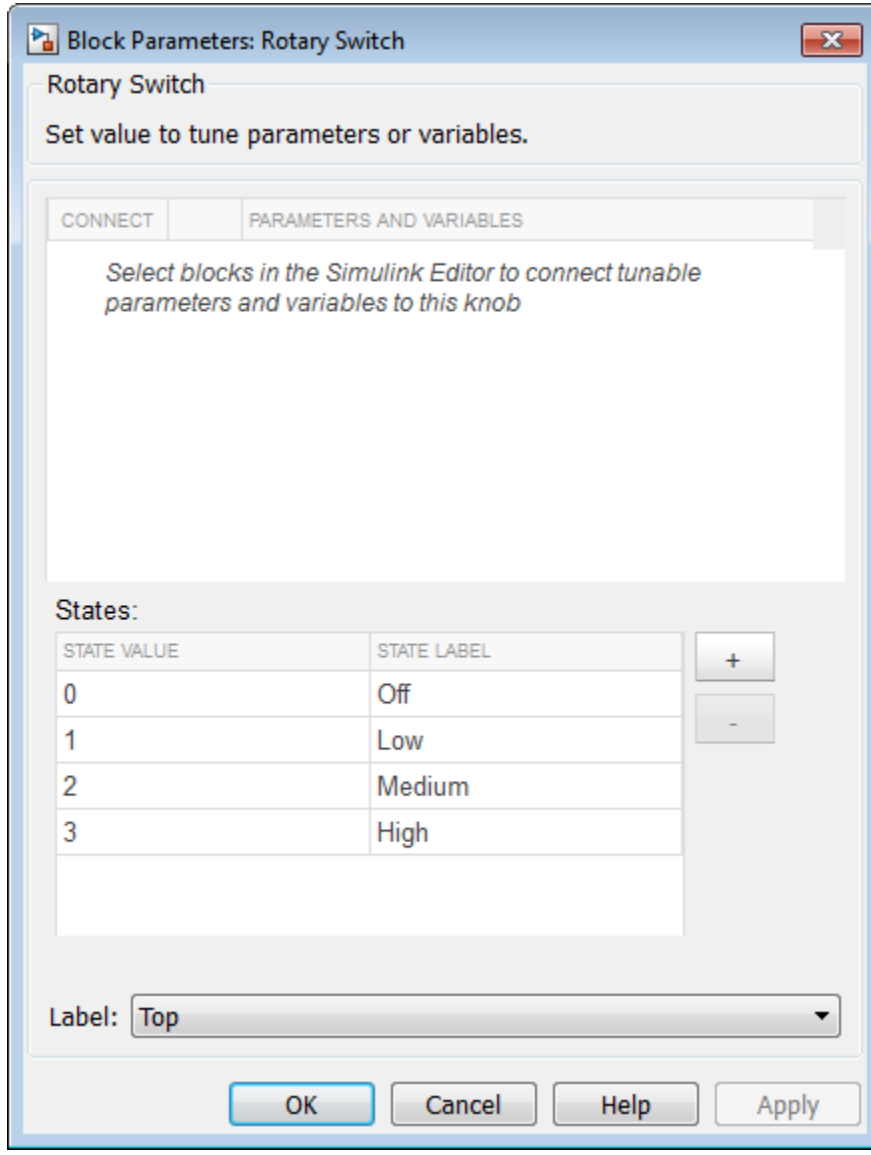
Limitations

The `Rotary Switch` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.

Limitation	Workaround
Parameters that index a variable array do not appear in the **Connection** table.	For example, a block parameter specified using the variable `engine(1)` will not appear in the table because the parameter uses an index of the variable `engine`, which is not a scalar variable. To make the parameter appear in the **Connection** table, change the block parameter field to a scalar variable, such as `engine_1`.

Parameters and Dialog Box



Connection

Select a block to connect and control a tunable parameter or variable.

To control a tunable parameter or variable, select a block in the model. The tunable parameter or variable appears in the **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable.

Settings

The table has a row for the tunable parameter or variable connected to the block. If there are no tunable parameters or variables selected in the model or the block is not connected to any tunable parameters or variables, then the table is empty.

States

The state values determine the discrete values generated for the tunable parameter or variable. You can modify the states by editing the **State Value** and **State Label** in the **States** table.

To add a state, click **+**, and enter the **State Value** and **State Label**.

To remove a state, select the state in the table, and click **-**.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

Introduced in R2015a

Rounding Function

Apply rounding function to signal

Library

Math Operations

Description

The Rounding Function block applies a rounding function to the input signal to produce the output signal.

You can select one of the following rounding functions from the **Function** list:

- `floor`

  Rounds each element of the input signal to the nearest integer value towards minus infinity.
- `ceil`

  Rounds each element of the input signal to the nearest integer towards positive infinity.
- `round`

  Rounds each element of the input signal to the nearest integer.
- `fix`

  Rounds each element of the input signal to the nearest integer towards zero.

The name of the selected function appears on the block.

The input signal can be a scalar, vector, or matrix signal having real- or complex-valued elements of type `double`. The output signal has the same dimensions, data type, and

numeric type as the input. Each element of the output signal is the result of applying the selected rounding function to the corresponding element of the input signal.

Tip Use the `Rounding Function` block instead of the `Fcn` block when you want vector or matrix output, because the `Fcn` block produces only scalar output.

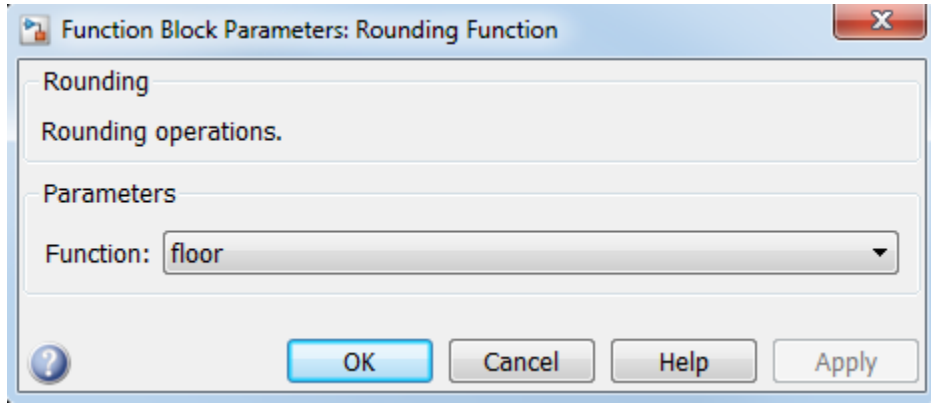
Also, the `Rounding Function` block provides two more rounding modes. The `Fcn` block supports `floor` and `ceil`, but does not support `round` and `fix`.

Data Type Support

The Rounding Function block accepts and outputs real signals of type `double` or `single`.

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

Parameters and Dialog Box



Function

Specify the rounding function.

Sample time

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

Characteristics

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

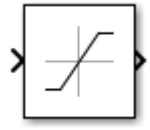
Introduced before R2006a

Saturation

Limit range of signal

Library

Discontinuities

Description 

The Saturation block imposes upper and lower limits on an input signal.

When the input is...	Where...	The block output is the...
Within the range specified by the **Lower limit** and **Upper limit** parameters	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

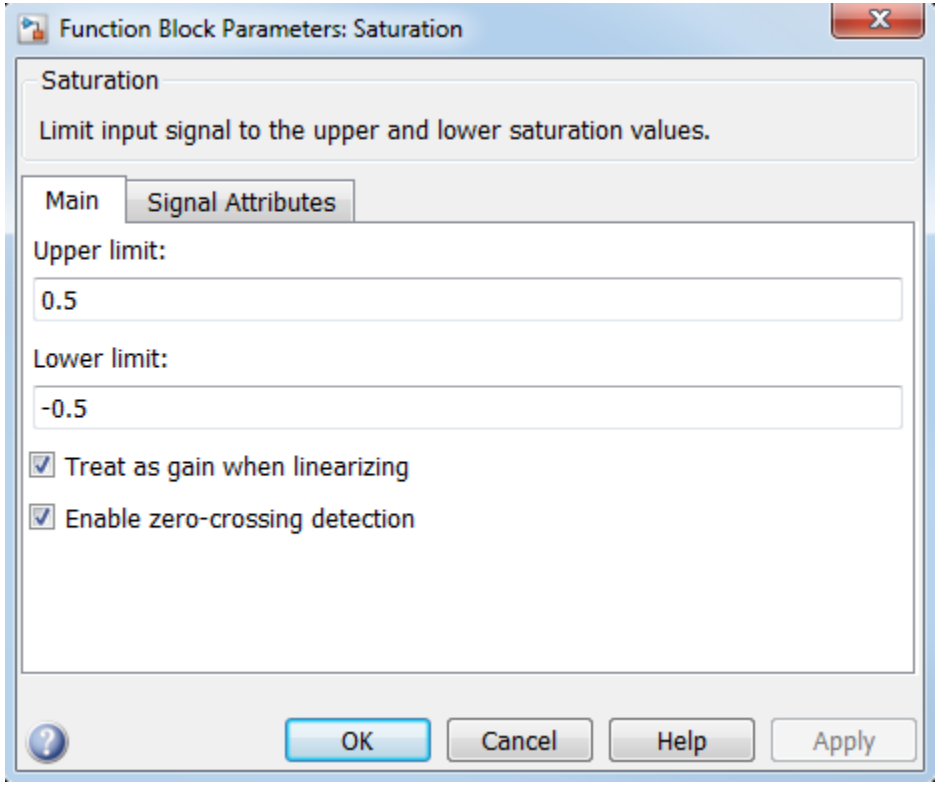
The Saturation block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

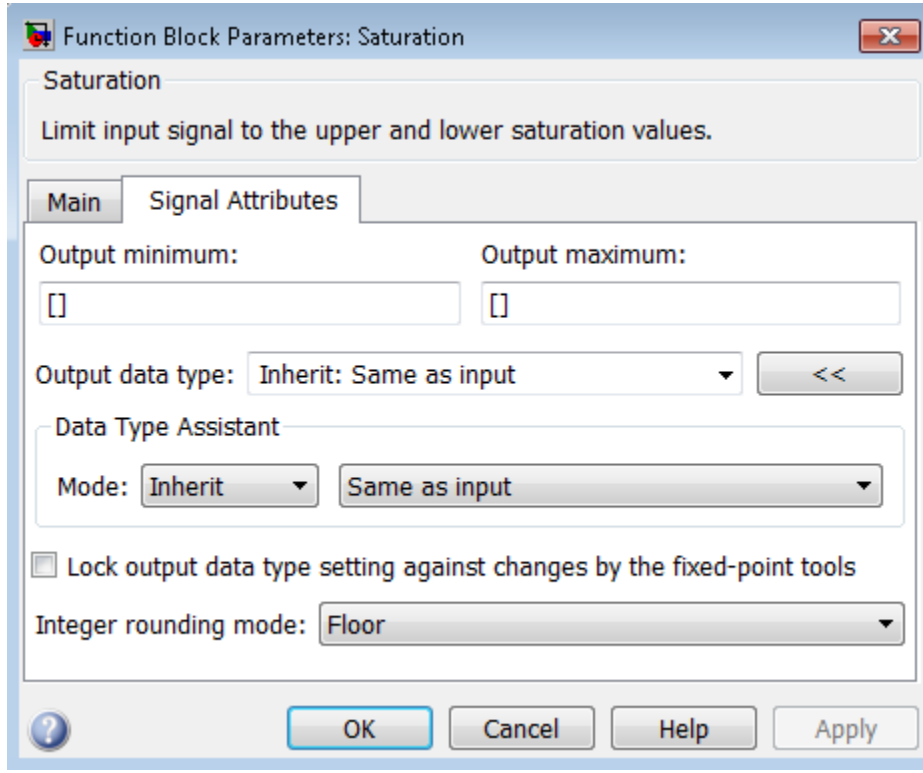
For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

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



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.
Main
Signal Attributes
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
OK
Cancel
Help
Apply

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 “Control Signal Data Types”.

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 6-101 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 6-101 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 6-101 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

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Sample time

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

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

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

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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.

Simulink uses the minimum to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 6-101 for the command-line information.

See Also

For more information, see “Control Signal 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 6-101 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.

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.

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**

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.

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.

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.

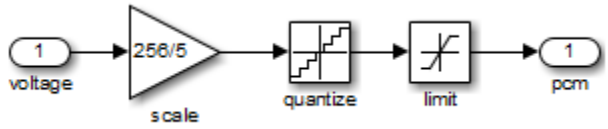
See Also

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

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

See Also

`Saturation Dynamic`

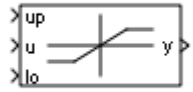
Introduced before R2006a

Saturation Dynamic

Bound range of input

Library

Discontinuities



Description

The Saturation Dynamic block bounds the range of an input signal to upper and lower saturation values. An input signal outside of these limits saturates to one of the bounds where:

- The input below the lower limit is set to the lower limit.
- The input above the upper limit is set to the upper limit.

The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

Data Type Support

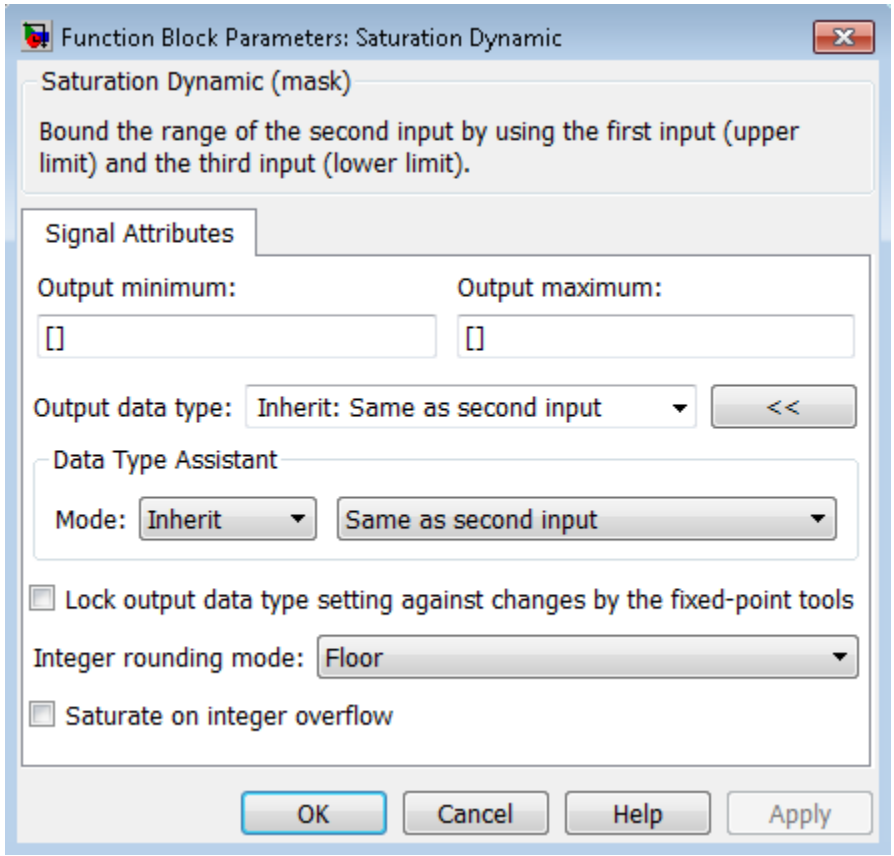
The Saturation Dynamic block accepts signals of the following data types:

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

Tip Although this block accepts Boolean signals, avoid this usage.

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

Parameters 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

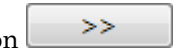
Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- 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 "Control Signal Data Types" in the Simulink User's Guide for more information.

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

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

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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation

Action	Reasons for Taking This Action	What Happens for Overflows	Example
			result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

Characteristics

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

Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

`Saturation`

Introduced before R2006a

Scope

Display signals generated during simulation

Description 

The Simulink Scope block displays time domain signals with respect to simulation time.

Input signal characteristics:

- **Signal** — Continuous (sample-based) or discrete (sample-based and frame-based).
- **Signal data type** — Any data type that Simulink supports including real, complex, fixed-point, and enumerated data types. See “Data Types Supported by Simulink”.
- **Signal dimension** — Scalar, one-dimensional (vector), two dimensional (matrix), or multidimensional. Display multiple channels within one signal depending on the dimension. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

Scope display features:

- **Simulation control** — Debug models from a Scope window using Run, Step Forward, and Step Backward toolbar buttons.
- **Multiple signals** — Plot multiple signals on the same *y*-axis (display) using multiple input ports.
- **Multiple** y**-axes (displays)** — Display multiple *y*-axes. All of the *y*-axes have a common time range on the *x*-axis.
- **Modify parameters** — Modify scope parameter values before and during a simulation.
- **Axis autoscaling** — During or at the end of a simulation. Margins are drawn at the top and bottom of the axes.
- **Display data after simulation** — If a Scope is closed at the start of a simulation, scope data is still written to the scope during a simulation. As a result, if you open the Scope after a simulation, the Scope displays simulation results for attached input signals.

Oscilloscope features:

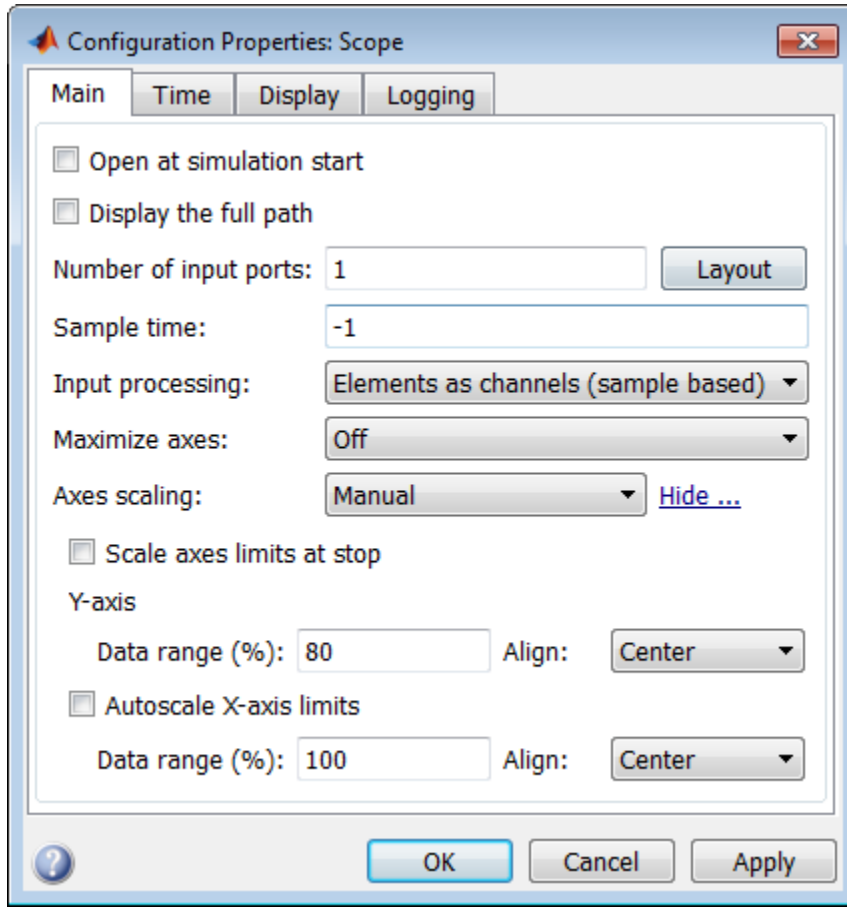
- **Triggers** — Set triggers to sync repeating signals and pause the display when events occur.
- **Data analysis** — Measure time and value differences between two signal data points. If you have a DSP System Toolbox™ license, measure signal characteristics including signal statistics, transitions, and peaks.

Note: Do not use Scope blocks in a Library. If you place a Scope block inside a library block with a locked link or in a locked library, Simulink displays an error when trying to open the Scope window.

To display internal data from a library block, add an output port to the library block, and then connect the port to a Scope block in your model.

Note: For information on controlling a Scope programmatically, see "Control Scopes Programmatically" in the Simulink documentation.

Configuration Properties



Open at simulation start

Specify when a Scope window opens.

Settings

Default: Clear for Scope block. Select for Time Scope block.

Select

Open Scope window when simulation starts.

Clear

Do not open a closed Scope at the start of a simulation.

Scope Configuration property: `OpenAtSimulationStart`.

Display the full path

Display full block path on Scope title bar.

Settings

Default: Clear

Select

Display block path and name.

Clear

Display block name.

Scope Configuration property: No corresponding property.

Number of input ports

Specify number of input ports on a Scope block, specified by a positive integer string. Maximum number of input ports is 96. This property does not apply to floating scopes and scope viewers.

Default: 1

Scope Configuration property: `NumInputPorts`.

Layout button

Specify number of displays. The maximum layout dimension is four rows by four columns.

- If the number of displays are equal to the number of ports, signals from each port appear on separate displays.
- If the number of displays are less than the number of ports, signals from additional ports appear on the last display.
- For display layouts with more than one row and column, ports are mapped to displays by column and then by rows.

To expand the layout dialog beyond 4 rows by 4 columns, click within the dialog an drag the layout to a maximum of 16 rows by 16 columns.

Settings

Default: 1

Scope Configuration property: `LayoutDimensions`.

Sample time

Specify time interval between Scope block updates during a simulation, specified as a positive real string. This property does not apply to floating scopes and scope viewers.

Settings

Default: `-1` for inherited

Scope Configuration property: `SampleTime`.

Input processing

Specify sample-based or frame-based processing of signals.

Settings

Default: `Elements as channels (sample based)` for Scope block. `Columns as channels (frame based)` for Time Scope block.

`Elements as channels (sample based)`

Process signal values in a channel at each time interval.

`Columns as channels (frame based)`

Process signal values in a channel as a group of values from multiple time intervals. Frame-based processing is available only with discrete input signals.

Scope Configuration property: `FrameBasedProcessing`.

Maximize axes

Maximize size of signal plots. Each of the plots expands to fit the full display. Maximizing the size of signal plots removes the background area around the plots.

Settings

Default: `Off` for Scope block. `Auto` for Time Scope block.

`Auto`

If **Title** and **Y-label** properties are not specified, maximize all plots.

`On`

Maximize all plots. Values in **Title** and **Y-label** are hidden.

`Off`

Do not maximize plots.

Scope Configuration property: `MaximizeAxes`.

Axes scaling

Specify when to scale *y*-axis to display all signal values.

Settings

Default: `Manual`

`Manual`

Manually scale y-axis range with **Scale Y-axis Limits** toolbar button.

`Auto`

Scale *y*-axis range during and after simulation. Selecting this option displays the **Do not allow Y-axis limits to shrink** check box.

`After N Updates`

Scale *y*-axes after specified number of block updates (time intervals). Selecting this option displays the **Number of updates** text box.

Dependency

If this property is set to `After N Updates`, also specify the property **Number of updates**.

Scope Configuration property: `AxesScaling`.

Do not allow Y-axis limits to shrink

Specify when *y*-axis range limits can change.

Settings

Default: Select

Select

Allow *y*-axis range limits to increase but not decrease during a simulation.

Clear

Allow y-axis range limits to increase and decrease.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property and set the **Axes scaling** property to `Auto` to display this property.

Number of updates

Specify the number of updates that occur during a simulation before a Scope scales the *y*-axes, specified as a positive integer string.

Settings

Default: `10`

Dependency

Display and activate this property by clicking the **Configure** link to the right of the **Axes scaling** property and set the **Axes scaling** property to `After N Updates`.

Scope Configuration property: `AxesScalingNumUpdates`.

Scale axes limits at stop

Specify when to scale axes.

Settings

Default: Select

Select

Scale axes when simulation stops.

Clear

Always scale axes.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

The *y*-axes limits are always scaled. The *x*-axis limits are scaled only if you also select the **Autoscale X-axis limits** check box.

Y-axis Data range (%)

Specify percentage of *y*-axis range for plotting data. For example, if you set this property to 100, plotted data uses the entire *y*-axis range.

Settings

Default: 80

Values are 1 through 100.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Y-axis Align

Specify where to align plotted data along the *y*-axis data range when **Y-axis Data range** is set to less than 100 percent.

Settings

Default: `Center`

`Top`
Align signals with maximum values at top of *y*-axis range.

`Center`
Align signals with maximum and minimum values centered.

`Bottom`
Align signals with minimum values at bottom of *y*-axis range.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Autoscale X-axis limits

Scale *x*-axis range limits when scaling axes.

Settings

Default: Clear

☑ Select
Scale *x*-axis range to fit all signal values. If **Axes scaling** is set to `Auto`, scales the data currently within the axes, not the entire signal in the data buffer.

☐ Clear
Do not scale *x*-axis range.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

X-axis Data range (%)

Specify percentage of *x*-axis range for plotting data. For example, if you set this property to `100`, plotted data uses the entire *x*-axis range.

Settings

Default: `100`

Values are 1 through 100.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

X-axis Align

Specify where to align plotted data along the *x*-axis when **X-axis Data range** is set to less than 100 percent.

Settings

Default: `Center`

`Top`
: Align signals with maximum values at top of *x*-axis range.

`Center`
: Align signals with maximum and minimum values centered.

`Bottom`
: Align signals with minimum values at bottom of *x*-axis range.

Dependency

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Time span

Specify length of *x*-axis range to display.

The block calculates the beginning and end times of the time range using the **Time display offset** and **Time span** properties. For example, if you set the **Time display offset** to `10` and the **Time span** to `20`, the scope sets the time range from `10` to `30`.

Settings

Default: `Auto`

`Auto`

Difference between the simulation start and stop times.

`User defined`

Value less than the total simulation time.

Scope Configuration property: `TimeSpan`.

Time span overrun action

Specify how to display data beyond the visible *x*-axis range.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

Settings

Default: `Wrap`

`Wrap`

Draw a full screen of data from left to right, clear the screen, and then restart drawing of data.

`Scroll`

Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

Scope Configuration property: `TimeSpanOverrunAction`.

Time units

Specify units to display on the *x*-axis.

Settings

Default: `None` for Scope block. `Metric` for Time Scope block.

`Metric`

Display time units based on the length of **Time span**.

`Seconds`

Display Time (seconds).

`None`

Do not display time units.

Scope Configuration property: `TimeUnits`.

Time display offset

Offset the *x*-axis by a specified time value, specified as a real number or vector of real numbers.

For input signals with multiple channels, you can enter a scaler or vector.

- Scaler — Offset all channels of an input signal by the same time value.
- Vector — Independently offset the channels.

Settings

Default: `0`

Scope Configuration property: `TimeDisplayOffset`.

Time-axis labels

Specify how *x*-axis (time) labels display.

Settings

Default: `Bottom Displays Only` for Scope block. `All` for Time Scope block.

`All`

Display *x*-axis labels on all *y*-axes.

`None`

Do not display labels. Selecting `None` also clears the **Show time-axis label** check box.

`Bottom Displays Only`

Display *x*-axis label on the bottom *y*-axis.

Dependency

Set **Active display** before setting this property. Activate this property by selecting **Show time-axis label** and setting **Maximize axes** to `off`.

Scope Configuration property: `TimeAxisLabels`.

Show time-axis label

Display or hide *x*-axis (time) labels.

Settings

Default: `Clear` for Scope block. `Select` for Time Scope block.


Select

Display *x*-axis label for the active display


Clear

Hide *x*-axis labels.

Dependency

Set **Active display** before setting this property. If you select this property and set the **Time-axis labels** is set to `None`, this property is deactivated.

Scope Configuration property: `ShowTimeAxisLabel`.

Active display

Display for setting display-specific properties, specified as a positive integer. The number of a display corresponds to its column-wise placement index.

Settings

Default: 1

Dependency

Setting this property selects the display for setting the properties **Show Grid**, **Show legend**, **Title**, **Plot signals as magnitude and phase**, **Y-label**, and **Y-Limits**.

Scope Configuration property: `ActiveDisplay`.

Title

Specify a title for display, specified as a character string. The default value `%<SignalLabel>` uses the input signal name for the title.

Settings

Default: `%<SignalLabel>`

Dependency

Set **Active display** before setting this property.

Scope Configuration property: `Title`.

Show legend

Show signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. See the `Scope` block reference for an example.

Settings

Default: Clear

☑ Select

Display signal legend. Continuous signals have straight lines before their names and discrete signals have step-shaped lines.

☐ Clear

Hide signal legend.

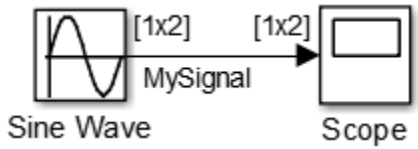
Dependency

Set **Active Display** before setting this property.

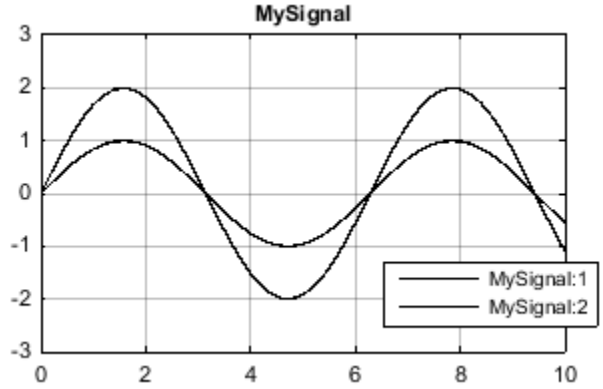
Scope Configuration property: `ShowLegend`.

Example

Connect a Sine Wave block to a Scope. Set the **Amplitude** parameter for the Sine Wave to `[1 2]`. Select the **Legends** check box for the Scope. Set the **Signal name** property for the signal to `MySignal`.



After simulating this model, the Scope window displays a sine wave for the two signal channels in `MySignal`, `MySignal:1`, and `MySignal:2`.



Edit the name of any channel in the legend by double-clicking the current name and entering a new channel name.

See also “Signal Dimensions”and “Determine Output Signal Dimensions”.

Show grid

Show vertical and horizontal grid lines.

Settings

Default: Select

Select

Display grid lines.

Clear

Hide grid lines.

Dependency

Set **Active Display** before setting this property.

Scope Configuration property: `ShowGrid`.

Plot signals as magnitude and phase

Split display into magnitude and phase plots.

Settings

Default: Clear

Select

Display magnitude and phase plots. If the signal is real, plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values.

Clear

Display signal plot. If the signal is complex, plots the real and imaginary parts on the same *y*-axis.

Dependency

Set **Active Display** before setting this property.

Scope Configuration property: `PlotAsMagnitudePhase`.

Y-limits (Minimum)

Specify minimum value of *y*-axis, specified as real number.

Settings

Default: `-10`

Dependency

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** applies this property value to the magnitude plot. The *y*-axis limits of the phase plot are always `[-180 180]`.

Scope Configuration property: `YLimits`.

Y-limits (Maximum)

Specify maximum value of *y*-axis, specified as real number.

Settings

Default: `+10`

Dependency

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** applies this property value to the magnitude plot. The *y*-axis limits of the phase plot are always `[-180 180]`.

Scope Configuration property: `YLimits`.

Y-label

Specify *y*-axis label, specified as a character string.

To display signal units, add `(%<SignalUnits>)` to the label. At the beginning of a simulation, Simulink replaces `(%SignalUnits)` with the units associated with the signals. For example, if you have a signal for velocity with units of `m/s`. enter

```
Velocity (%<SignalUnits>)
```

Settings

Default: No label for Scope block. `Amplitude` for Time Scope block.

Dependency

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** hides this property and plots are labeled `Magnitude` and `Phase`.

Scope Configuration property: `YLabel`.

Limit data points to last

Specify to limit buffered data values before plotting and saving signals.

Settings

Default: Clear, 5000

Select

Save specified number of data values for each signal. If the signal is frame-based, the number of buffered data values is the specified number of data values multiplied by the frame size.

For simulations with **Stop time** set to `inf`, consider selecting **Limit data points to last**.

In some cases, for example where the sample time is small, selecting this parameter can have the effect of plotting signals for less than the entire time range of a simulation. If a scope plots a portion of your signals, consider increasing the number of data values to save.

Clear

Save and plot all data values. Clearing **Limit data points to last** can cause an out-of-memory error for simulations that generate a large amount of data or for systems without enough available memory.

Dependency

If this property is selected, also specify the number of data values by entering a positive integer in the text box. This property limits the data values a scope plots and the data values saved in the MATLAB variable specified in **Variable name**. Data values are from the end of a simulation.

Scope Configuration properties: `DataLoggingLimitDataPoints` and `DataLoggingMaxPoints`.

Decimation

Reduce the amount of scope data to display and save.

Settings

Default: Clear, `2`

☑ Select

Plot and Log (save) scope data every `N`th data point, where `N` is the decimation factor entered in the text box.

☐ Clear

Save all scope data values.

Dependency

If this property is selected, also specify the decimation factor by entering a positive integer in the text box. The scope buffers every `N`th data point, where `N` is the decimation factor you specify. A value of `1` buffers all data values. This property limits the data values a scope plots and the data values saved in the MATLAB variable specified in **Variable name**.

Log data to workspace

Activate saving scope data to a variable in the MATLAB workspace.

Settings

Default: Clear

☑ Select

Activate logging and activate the **Variable name**, **Save format**, and **Decimation** properties. This property does not apply to floating scopes and scope viewers.

☐ Clear

Inactivate logging and logging properties are unavailable.

Dependency

If this property is selected, also specify the properties **Variable name** and **Save format**.

Scope Configuration property: `DataLogging`.

For a procedure showing how to save signals to the MATLAB Workspace using a Scope block, see “Save Simulation Data Using Floating Scope Block”.

Variable name

Specify a variable name for saving scope data in the MATLAB workspace, specified as a character string. This property does not apply to floating scopes and scope viewers.

Settings

Default: `ScopeData`

Dependency

Activate this property by selecting **Log data to workspace**.

Scope Configuration property: `DataLoggingVariableName`.

Save format

Select variable format for saving data to the MATLAB workspace. This property does not apply to floating scopes and scope viewers.

Settings

Default: `Dataset`

`Dataset`

Save data as a dataset object. This format does not support variable-size data, MAT-file logging, or external mode archiving. See Simulink.SimulationData.Dataset.

`Structure With Time`

Save data as a structure with associated time information.

`Structure`

Save data as a structure.

`Array`

Save data as an array with associated time information. This format does not support variable-size data.

Dependency

Activate this property by selecting **Log data to workspace**.

Scope Configuration property: `DataLoggingSaveFormat`.

Style Properties

Open the Style dialog box:

- From the menu, select **View > Style**.
- From the Configuration Properties button arrow. select the Style button .

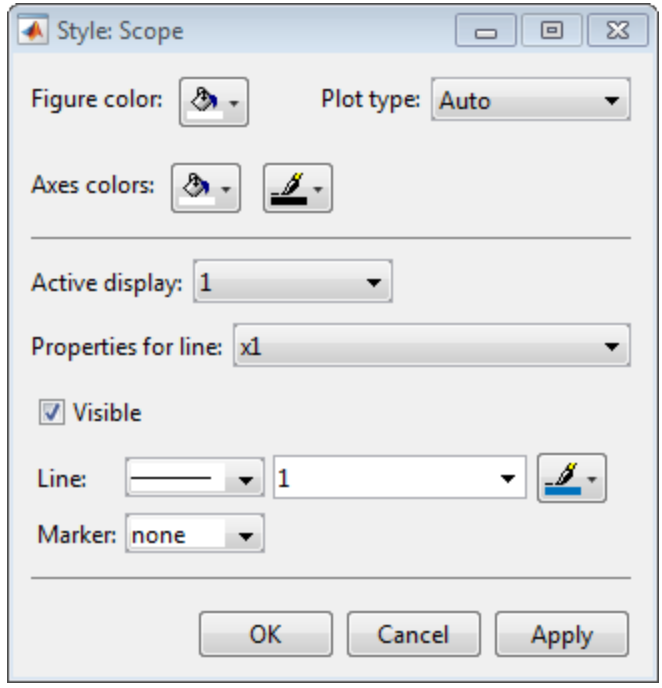


Figure color

Select background color for display.

Plot type

Specify how to plot a signal.

Default: `Auto` for Scope block. `Line` for Time Scope block.

- `Line` — Line graph.
- `Stairs` — Stair-step graph.
- `Auto` — Line graph if it is a continuous signal or a stair-step graph if it is a discrete signal.

Active display

Select active display for setting style properties.

Default: 1

Axes colors

Select the background color for axes (displays) with the first color pallet. Select the grid and label color with the second color pallet.

Properties for line

Select active line for setting line style properties.

Visible

Plot signal on active display.

Default: Select

 Select

Plot signal.

 Clear

Hide signal.

Line

Select line style, width, and color.

Marker

Select marker style.

Default: None

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	No

See Also

`Floating Scope` | `Scope Viewer`

How To

- “Scope Blocks and Scope Viewer Overview”
- “Simulate a Model Interactively”
- “Step Through a Simulation”
- “Scope Tasks”
- “Floating Scope and Scope Viewer Tasks”
- “Scope Trigger Panel”
- “Scope Measurement Panels”
- “Control Scopes Programmatically”

Introduced in R2015b

Second-Order Integrator, Second-Order Integrator Limited

Integrate input signal twice

Library

Continuous

Description

The `Second-Order Integrator` block and the `Second-Order Integrator Limited` block solve the second-order initial value problem:

$$\frac{d^2x}{dt^2} = u,$$
$$\left.\frac{dx}{dt}\right|_{t=0} = dx_o,$$
$$x|_{t=0} = x_o,$$

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 Integrator Limited` block is identical to the `Second-Order Integrator` block with the exception that it defaults to limiting the states based on the

specified upper and lower limits. For more information, see “Limiting the States” on page 1-1665.

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 “Solvers”.) The selected solver computes the states of the `Second-Order Integrator` block at the current time step using the current input value.

Use the block parameter dialog box to:

- Specify whether the source of each state initial condition is internal or external
- Specify a value for the state initial conditions
- Define upper and lower limits on either or both states
- Specify absolute tolerances for each state
- Specify names for both states
- Choose an external reset condition
- Enable zero-crossing detection
- Reinitialize dx/dt when x reaches saturation
- Specify that Simulink disregard the state limits and external reset for linearization operations

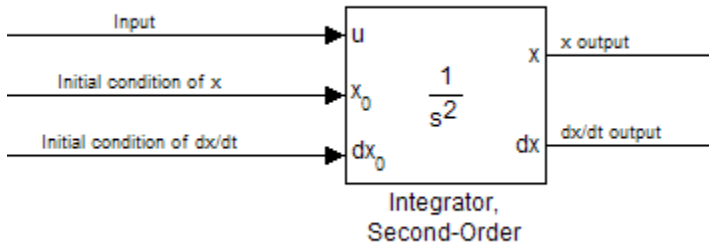
Defining Initial Conditions

You can define the initial conditions of each state individually as a parameter on the block dialog box or input one or both of them from an external signal.

- To define the initial conditions of state *x* as a block parameter, use the **Initial condition source x** drop-down menu to select `internal` and enter the value in the **Initial condition x** field.
- To provide the initial conditions from an external source for state *x*, specify the **Initial condition source x** parameter as `external`. An additional input port appears on the block.
- To define the initial conditions of state *dx/dt* as a block parameter, use the **Initial condition source dx/dt** drop-down menu to select `internal` and enter the value in the **Initial condition dx/dt** field.

- To provide the initial conditions from an external source for state *dx/dt*, specify **Initial condition source dx/dt** as `external`. An additional input port appears on the block.

If you choose to use an external source for both state initial conditions, your block appears as follows.



Note:

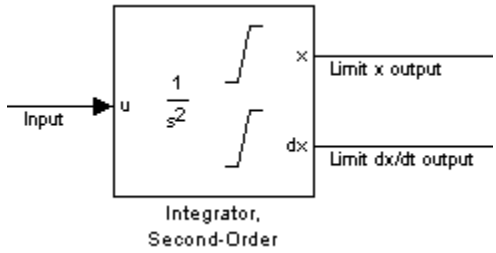
- Simulink does not allow initial condition values of `inf` or `NaN`.

- If you limit state *x* or state *dx/dt* by specifying saturation limits (see "Limiting the States" on page 1-1665) and one or more initial conditions are outside the corresponding limits, then the respective states are initialized to the closest valid value and a set of consistent initial conditions is calculated.

Limiting the States

When modeling a second-order system, you may need to limit the block states. For example, the motion of a piston within a cylinder is governed by Newton's Second Law and has constraints on the piston position (*x*). With the `Second-Order Integrator` block, you can limit the states *x* and *dx/dt* independent of each other. You can even change the limits during simulation; however, you cannot change whether or not the states are limited. An important rule to follow is that an upper limit must be strictly greater than its corresponding lower limit.

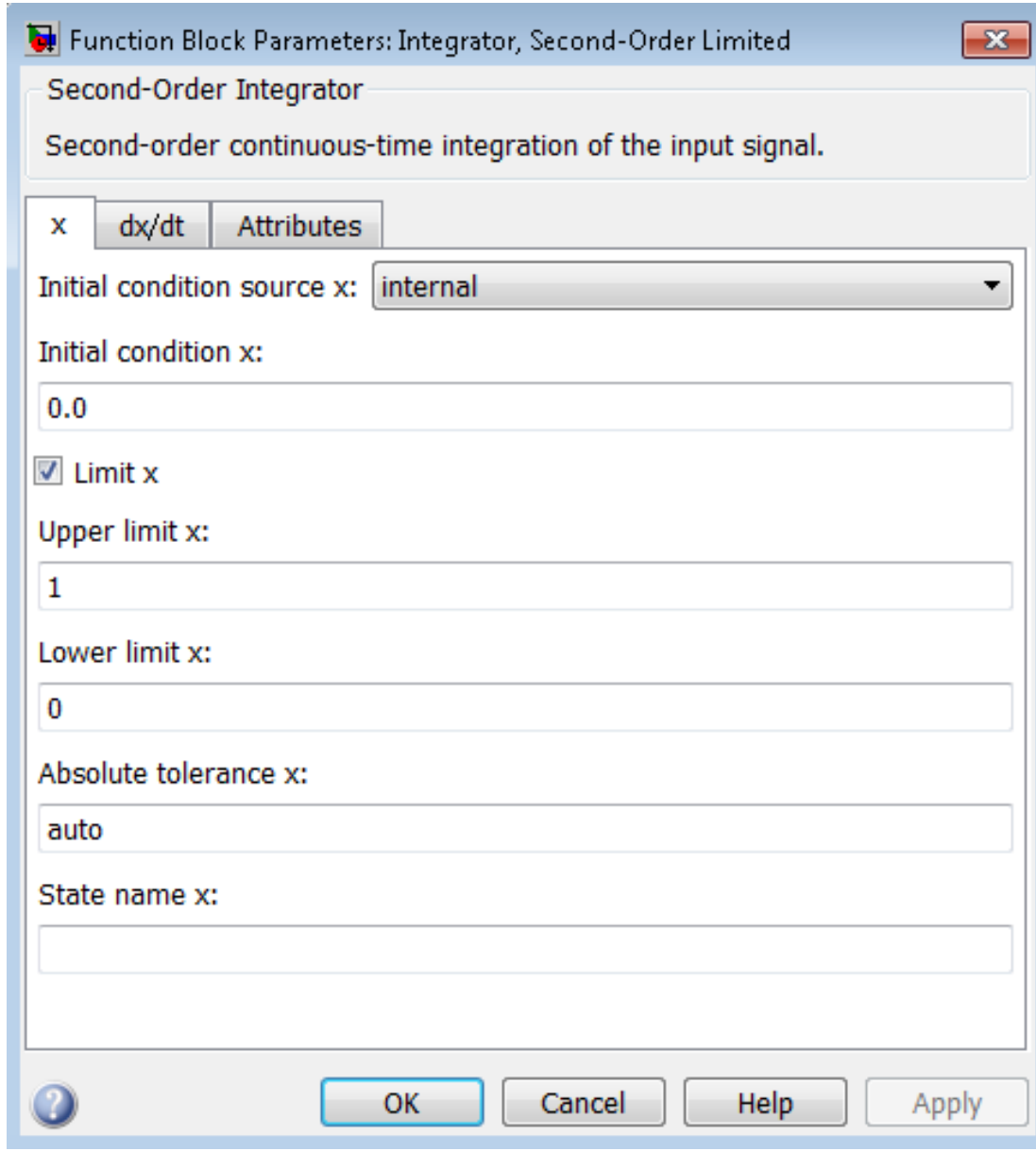
The block appearance changes when you limit one or both states. Shown below is the appearance of the block with both states limited.



For each state, you can use the block parameter dialog box to set appropriate saturation limits.

Limiting x only

If you use the `Second-Order Integrator Limited` block, both states are limited by default. But you can also manually limit state *x* on the `Second-Order Integrator` block by selecting **Limit x** and entering the limits in the appropriate parameter fields.



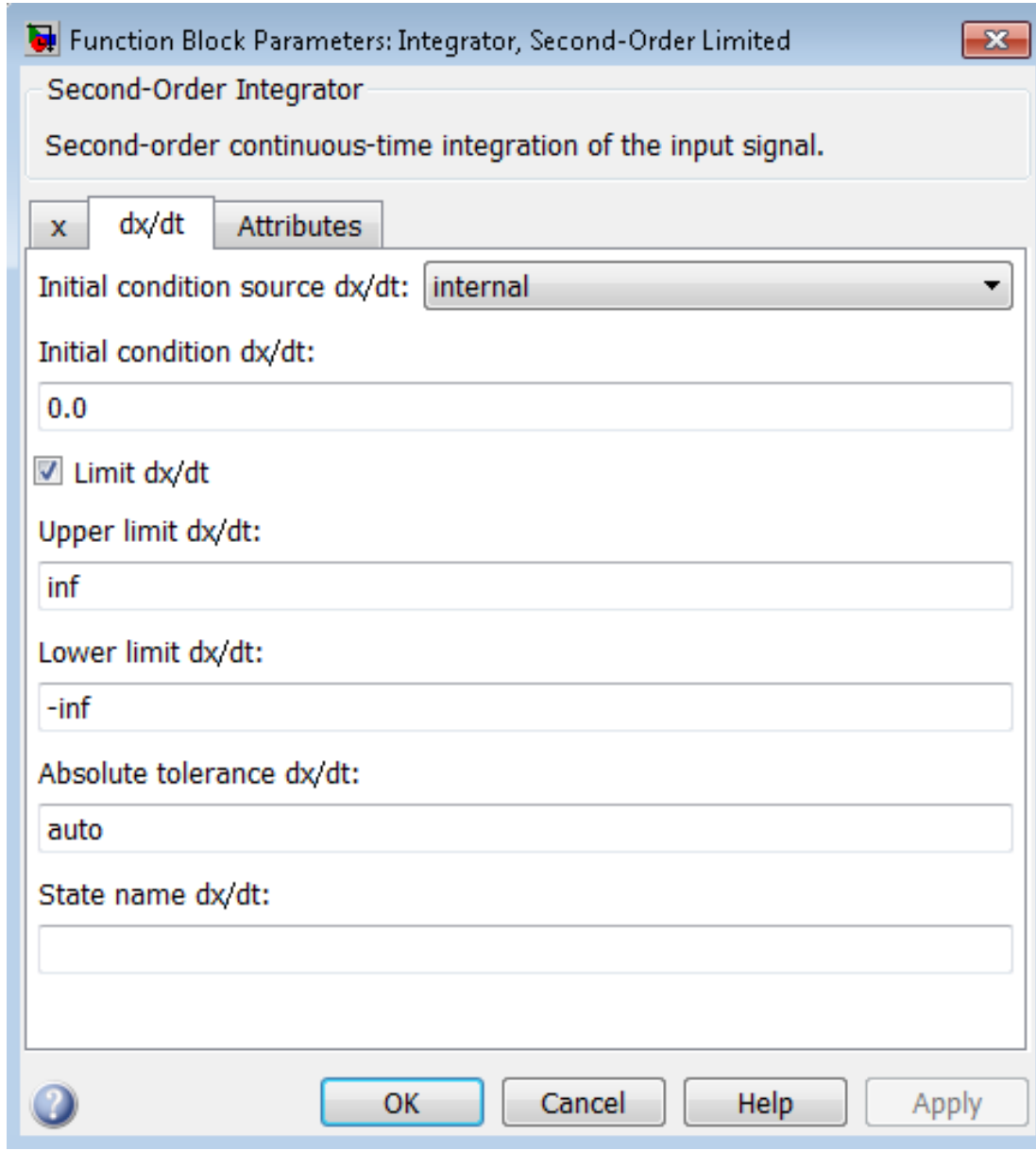
The block then determines the values of the states as follows:

- When x is less than or equal to its lower limit, the value of x is held at its lower limit and dx/dt is set to zero.
- When x is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.
- When x is greater than or equal to its upper limit, the value of x is held at its upper limit and dx/dt is set to zero.

You can choose to reinitialize dx/dt to a new value at the time when x reaches saturation. See "Reinitializing dx/dt when x reaches saturation" on page 1-1672

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
Second-Order Integrator
Second-order continuous-time integration of the input signal.
x
dx/dt
Attributes
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:
OK
Cancel
Help
Apply

If you limit only the state dx/dt, then the block determines the values of dx/dt as follows:

- When dx/dt is less than or equal to its lower limit, the value of dx/dt is held at its lower limit.
- When dx/dt is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.
- When dx/dt is greater than or equal to its upper limit, the value of dx/dt is held at its upper limit.

When state dx/dt is held at it upper or lower limit, the value of x is governed by the first-order initial value problem:

$$\frac{dx}{dt} = L,$$
$$x(t_L) = x_L,$$

where L is the dx/dt limit (upper or lower), t_L is the time when dx/dt reaches this limit, and x_L is the value of state x at that time.

Limiting Both States

When you limit both states, Simulink maintains mathematical consistency of the states by limiting the allowable values of the upper and lower limits for dx/dt. Such limitations are necessary to satisfy the following constraints:

- When x is at its saturation limits, the value of dx/dt must be zero.
- In order for x to leave the upper limit, the value of dx/dt must be strictly negative.
- In order for x to leave its lower limit, the value of dx/dt must be strictly positive.

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

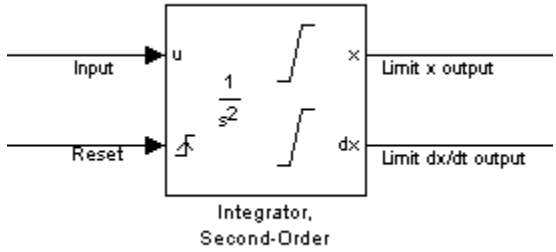
ODE when dx/dt is held at one of its limits. In such cases, when x reaches one of its limits, it is held at that limit and dx/dt is set to zero.

- 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 1-1672

Resetting the State

The block can reset its states to the specified initial conditions based on an external signal. To cause the block to reset its states, select one of the **External reset** choices on the **Attributes** pane. A trigger port appears on the block below its input port and indicates the trigger type.



- Select `rising` to reset the states when the reset signal rises from zero to a positive value, from a negative to a positive value, or a negative value to zero.
- Select `falling` to reset the states when the reset signal falls from a positive value to zero, from a positive to a negative value, or from zero to negative.
- Select `either` to reset the states when the reset signal changes from zero to a nonzero value or changes sign.

The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results (see "Algebraic Loops").

Enabling Zero-Crossing Detection

This parameter controls whether zero-crossing detection is enabled for this block. By default, the **Enable zero-crossing detection** parameter is selected on the **Attributes** pane. However, this parameter is only in affect if the **Zero-crossing control**, on the **Solver** pane of the Configuration Parameters dialog, is set to `Use local settings`. For more information, see "Zero-Crossing Detection".

Reinitializing *dx/dt* when *x* reaches saturation

For certain modeling applications, *dx/dt* must be reinitialized when state *x* reaches its limits in order to pull *x* out of saturation immediately. You can achieve this by selecting **Reinitialize dx/dt when x reaches saturation** on the **Attributes** pane.

If this option is on, then at the instant when *x* reaches saturation, Simulink checks whether the current value of the *dx/dt* initial condition (parameter or signal) allows the state *x* to leave saturation immediately. If so, Simulink reinitializes state *dx/dt* with the value of the initial condition (parameter or signal) at that instant. If not, Simulink ignores this parameter at the current instant and sets *dx/dt* to zero to make the block states consistent.

This parameter only applies at the time when *x* actually reaches saturation limit. It does not apply at any future time when *x* is being held at saturation.

Refer to the sections on limiting the states for more information. For an example, see the `sldemo_bounce` example.

Disregarding State Limits and External Reset for Linearization

For cases where you simplify your model by linearizing it, you can have Simulink disregard the limits of the states and the external reset by selecting **Ignore state limits and the reset for linearization**.

Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see "Error Tolerances for Variable-Step Solvers") to compute the output of the integrator blocks. If this value does not provide sufficient error control, specify a more appropriate value for state *x* in the **Absolute tolerance x** field and for state *dx/dt* in the **Absolute tolerance dx/dt** field of the parameter dialog box. Simulink uses the values that you specify to compute the state values of the block.

Specifying the Display of the Output Ports

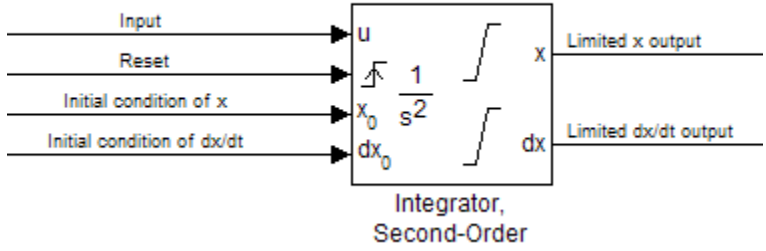
You can control whether or not to display the *x* or the *dx/dt* output port using the `ShowOutput` parameter. You can display one output port or both; however, you must select at least one.

Specifying the State Names

You can specify the name of *x* states and *dx/dt* states using the `StateNameX` and `StateNameDXDT` parameters. However, you must specify names for either both or neither; you cannot specify names for just *x* or just *dx/dt*. Both state names must have identical type and length. Furthermore, the number of names must evenly divide the number of states.

Selecting All Options

When you select all options, the block icon looks like this.



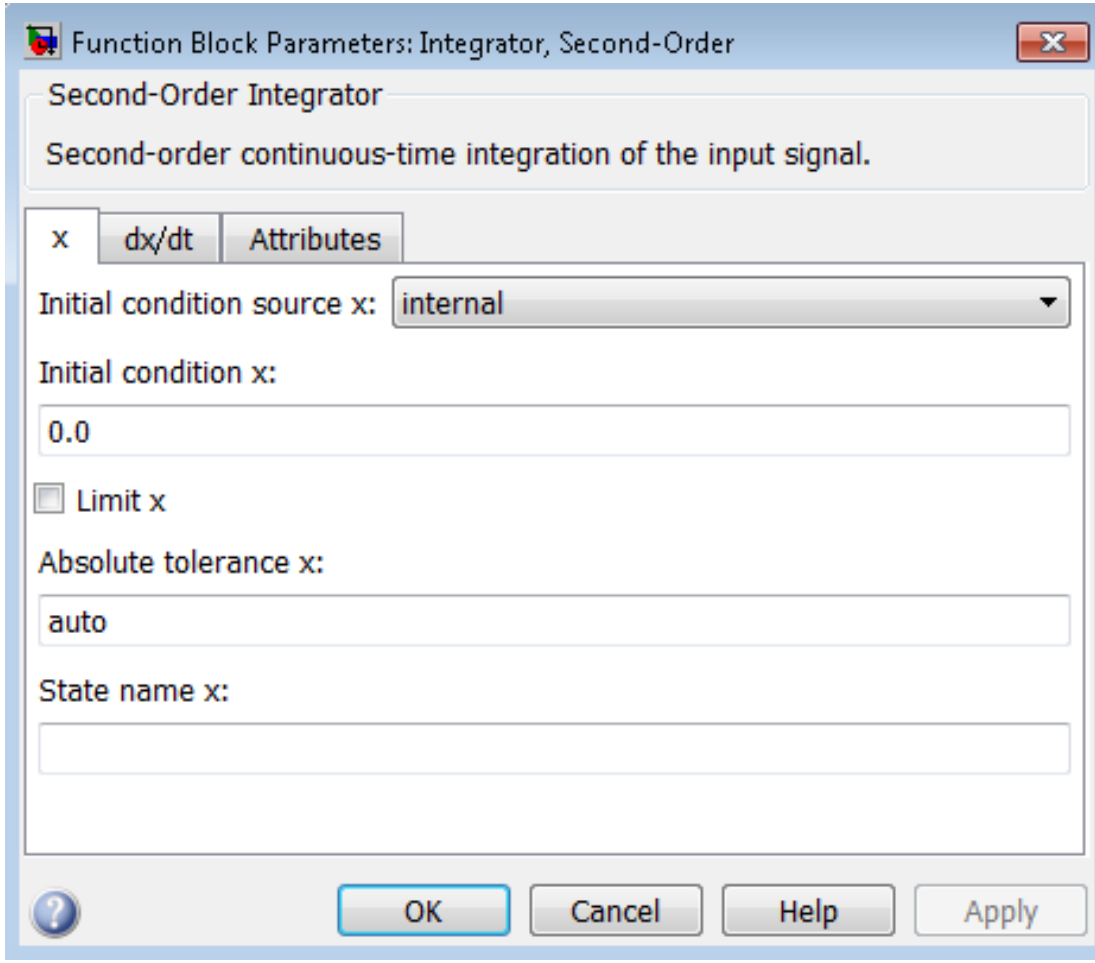
Data Type Support

The Integrator block accepts and outputs signals of type `double` on its data ports. The external reset port accepts signals of type `double` or `Boolean`.

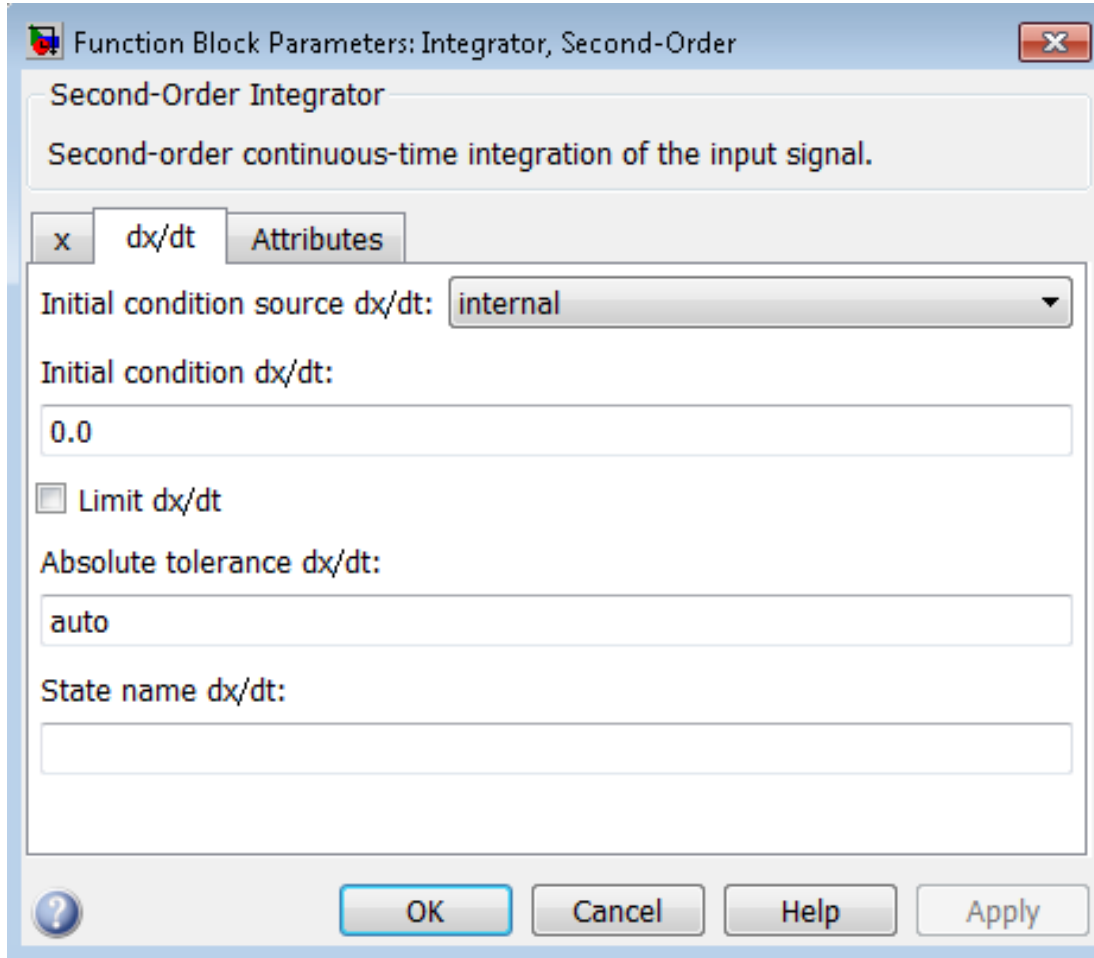
For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

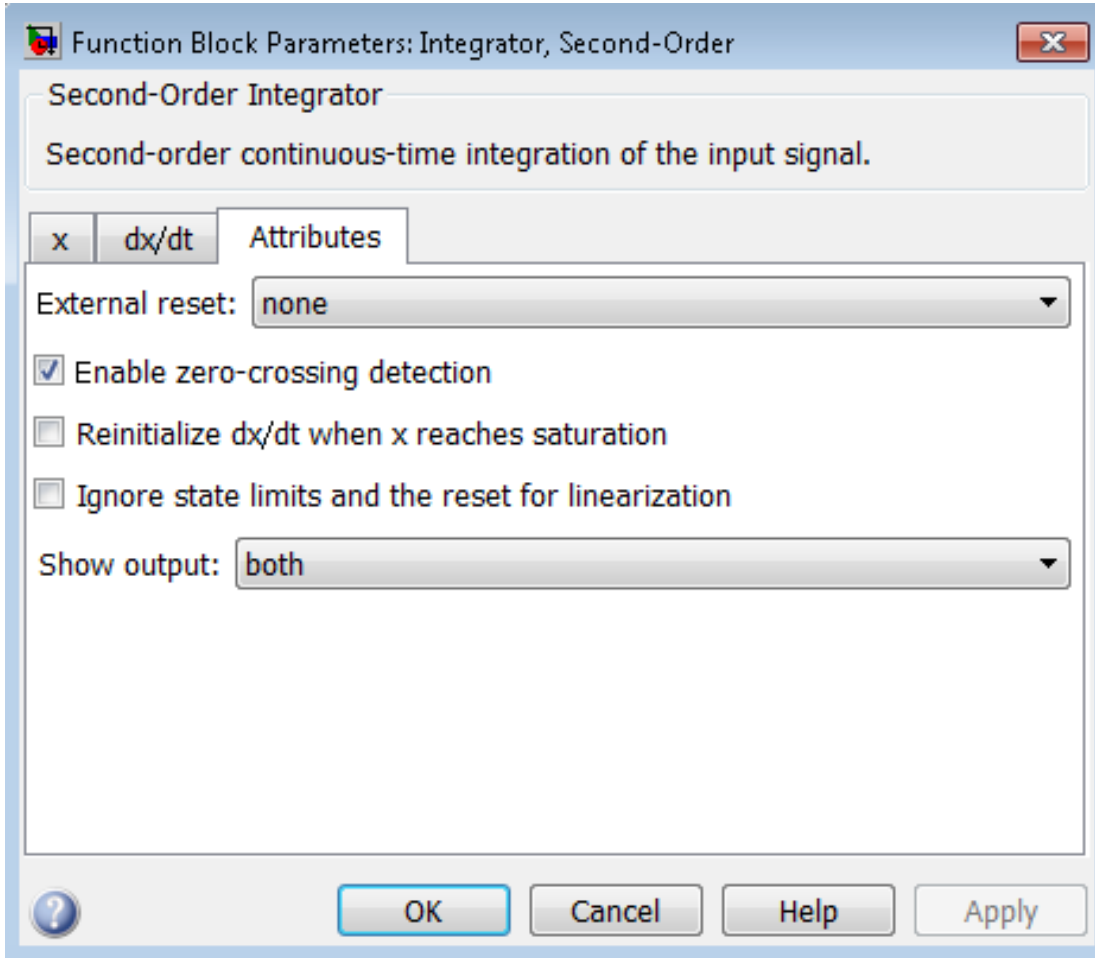
The **x** pane of the `Second-Order Integrator` block dialog box appears as follows:



The **dx/dt** pane of the `Second-Order Integrator` block dialog box appears as follows:



The **Attributes** pane of the `Second-Order Integrator` block dialog box appears as follows:



- "Initial condition source x" on page 1-1678
- "Initial condition x" on page 1-1679
- "Limit x" on page 1-1680
- "Upper limit x" on page 1-1681
- "Lower limit x" on page 1-1682
- "Absolute tolerance x" on page 1-1683
- "State name x" on page 1-1684

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: scalar or vector
Value: `'0'`
Default: `'0'`

Limit x

Limit state *x* of the block to a value between the **Lower limit x** and **Upper limit x** parameters.

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

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: `'inf'`
Default: `'inf'`

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: `'-inf'`
Default: `'-inf'`

Absolute tolerance x

Specify the absolute tolerance for computing state *x*.

Settings

Default: auto

- You can enter `auto`, –1, a positive real scalar or vector.
- If you enter `auto` or –1, Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute state *x*.
- If you enter a real scalar value, that value overrides the absolute tolerance in the Configuration Parameters dialog box and is used for computing all *x* states.
- If you enter a real vector, the dimension of that vector must match the dimension of state *x*. These values override the absolute tolerance in the Configuration Parameters dialog box.

Command-Line Information

Parameter: `AbsoluteToleranceX`
Type: string, scalar, or vector
Value: `'auto' | '-1'|` any positive real scalar or vector
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: `' '` | user-defined
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: `'0'`
Default: `'0'`

Limit dx/dt

Limit the dx/dt state of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters.

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: `'Off' | 'On'`
Default: `'Off'`

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: `'inf'`
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: `'-inf'`
Default: `'-inf'`

Absolute tolerance dx/dt

Specify the absolute tolerance for computing state *dx/dt*.

Settings

Default: `auto`

- You can enter `auto`, –1, a positive real scalar or vector.
- If you enter `auto` or –1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute the *dx/dt* output of the block.
- If you enter a numeric value, that value overrides the absolute tolerance in the Configuration Parameters dialog box.

Command-Line Information

Parameter: `AbsoluteToleranceDXDT`
Type: string, scalar, or vector
Value: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

State name dx/dt

Assign a unique name to state *dx/dt*.

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: `' '` | user-defined
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

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Reinitialize dx/dt when x reaches saturation

At the instant when state *x* reaches saturation, reset *dx/dt* to its current initial conditions.

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

Ignore state limits and the reset for linearization

For linearization purposes, have Simulink ignore the specified state limits and the external reset.

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: `'off' | 'on'`
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'`

Characteristics

Data Types	Double
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

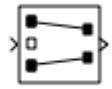
Introduced in R2010a

Selector

Select input elements from vector, matrix, or multidimensional signal

Library

Signal Routing



Description

The Selector block generates as output selected or reordered elements of an input vector, matrix, or multidimensional signal.

A Selector block accepts vector, matrix, or multidimensional signals as input. The parameter dialog box and the block's appearance change to reflect the number of dimensions of the input.

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

For example, assume a 6-D signal with a one-based index mode. The table of the Selector block dialog changes to include one row for each dimension. Suppose that you define each dimension with the following entries:

- Dimension 1
  - **Index Option**, select `Select all`
- Dimension 2
  - **Index Option**, select `Starting index (dialog)`

  - **Index**, enter `2`
  - **Output Size**, enter `5`
- Dimension 3
  - **Index Option**, select `Index vector (dialog)`
  - **Index**, enter `[1 3 5]`
- Dimension 4
  - **Index Option**, select `Starting index (port)`
  - **Output Size**, enter `8`
- Dimension 5
  - **Index Option**, select `Index vector (port)`
- Dimension 6
  - **Index Option**, select `Starting and ending indices (port)`

The output will be `Y=U(1:end,2:6,[1 3 5],Idx4:Idx4+7,Idx5, Idx6(1):Idx6(2))`, where `Idx4`, `Idx5`, and `Idx6` are the index ports for dimensions 4, 5, and 6.

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

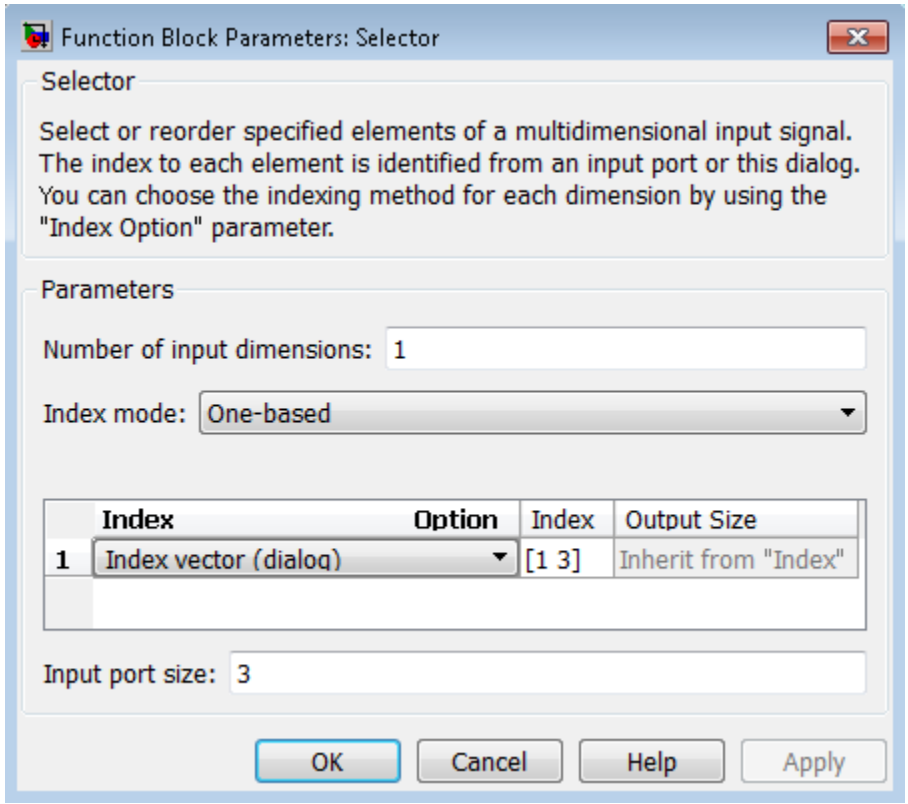
Data Type Support

The data port of the Selector block accepts signals of any signal type and any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types. The data port accepts mixed-type signals. The index port accepts built-in data types, but not Boolean. The elements of the output have the same type as the corresponding selected input elements.

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

Note: The Selector block is not designed to accept virtual bus input signals. For virtual bus inputs, use a Bus Selector block instead of a Selector block.

Parameters 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

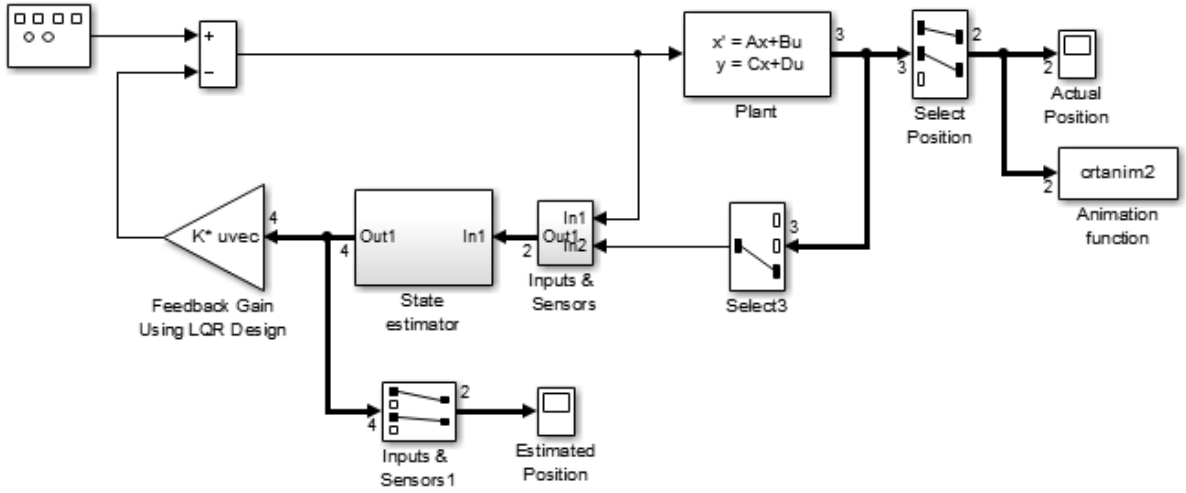
Specify the width of the block input signal (`-1` for inherited) — 1-D signals only.

Sample time

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

Examples

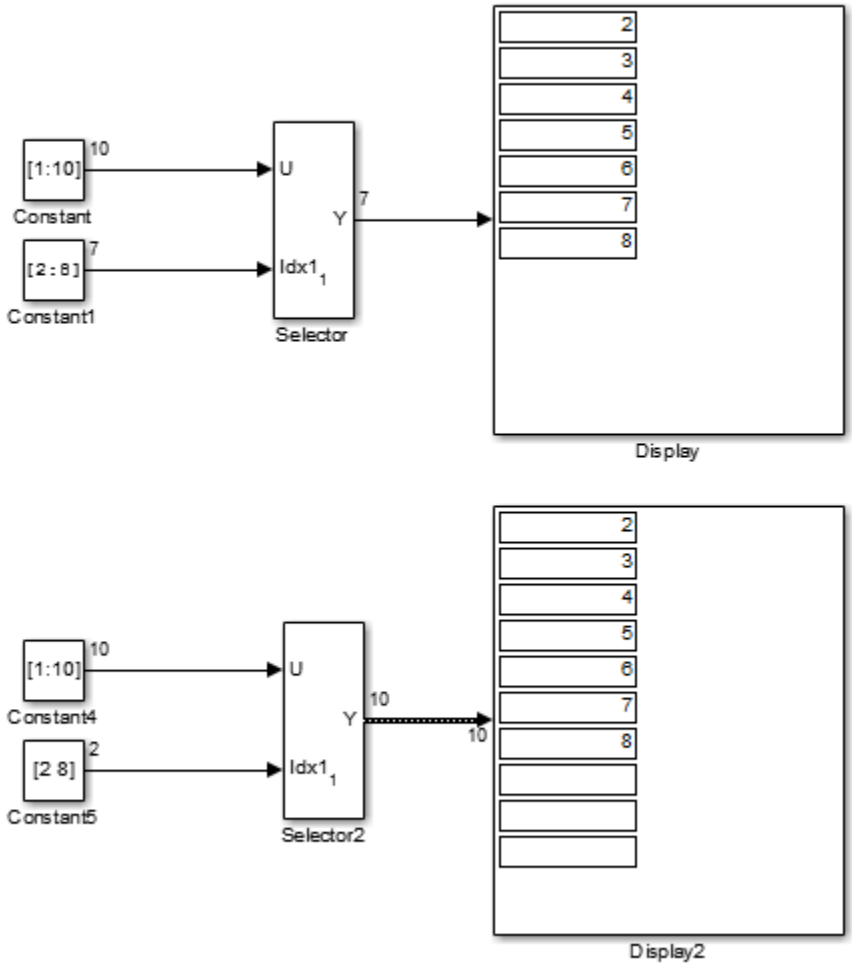
The `sldemo_dblcart1` model shows the use of three Selector blocks. The following diagram shows what the model looks like after you enable **Display** > **Signals & Ports** > **Signal Dimensions** and simulate the model.



All three Selector blocks set the **Index Option** to `Index vector (dialog)`, which allows you to specify the indices of the specific signals that you want to select, using the **Index** parameter. The **Input port size** parameter is set to the dimension of the largest input signal.

The following model shows the result of simulating two Selector blocks that have the same kind of input signals, but which have two different **Index Option** settings.

Both Selector blocks select 7 values from the input signal that feeds the `U` port. However, the `Selector` block outputs a fixed-size signal, whereas the `Selector2` block outputs a variable-size signal whose compiled signal dimension is 10 instead of 7.



The `Selector` block sets **Index Option** to `Index vector (port)`, which uses the input signal from `Constant1` as the index vector. The dimension of the input signal is 7, so the `Display` block shows the 7 values of the `Constant1` block. The `Selector2` block sets the **Input port size** parameter to `10`, which is the size of the largest input signal to the `Selector1` block.

The `Selector2` block uses the same configuration as the `Selector` block, except that the **Index Option** is set to `Starting and ending indices (port)`. The output uses the size of **Input port size** parameter (`10`), even though the size of the input signal is 7. The `Display2` block shows empty boxes for the three extra dimensions.

Characteristics

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

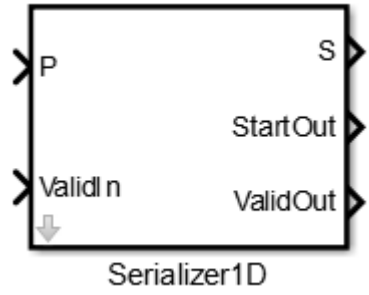
Introduced before R2006a

Serializer1D

Convert vector signal to scalar or smaller vectors

Library

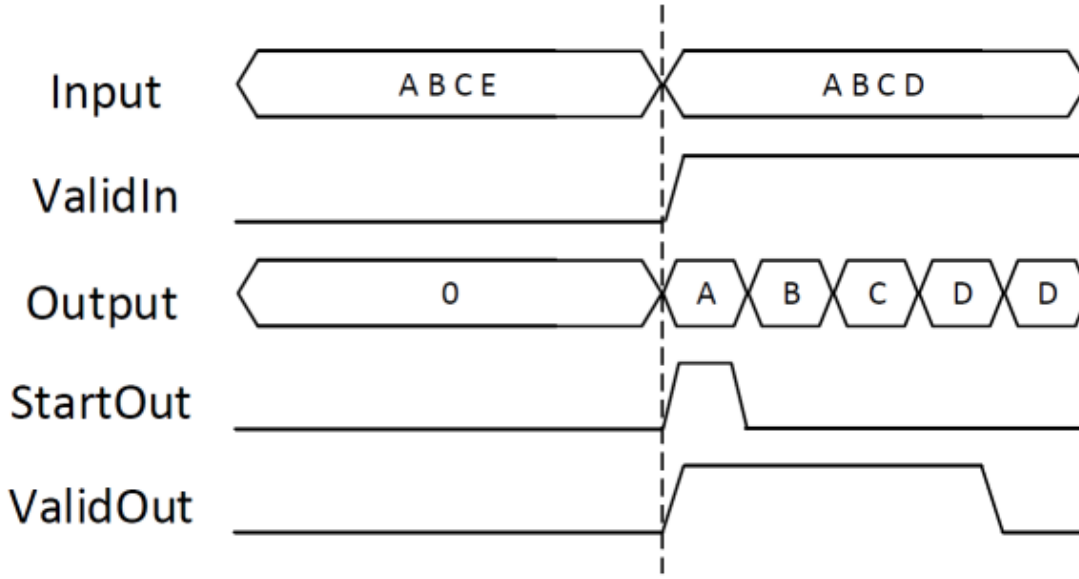
HDL Coder / HDL Operations



Description

The `Serializer1D` block converts a slower vector signal into a faster stream of scalar signals or smaller size vector signals. The slower vector is converted to a faster signal based on the **Ratio** and **Idle Cycle** values. Sample time also changes to match the faster, serialized output.

You can configure the serialization to depend on a valid Boolean signal ValidIn. The serialization can also output a Boolean signal to indicate when to start deserialization StartOut and a signal to indicate when the output data is valid ValidOut. Consider this example:



- **Ratio** is 4 and **Idle Cycles** is 1, so for each slow cycle vector output is 5 fast signals (4 serialized signals and one idle cycle).
- **ValidIn**, **StartOut**, and **ValidOut** are selected.

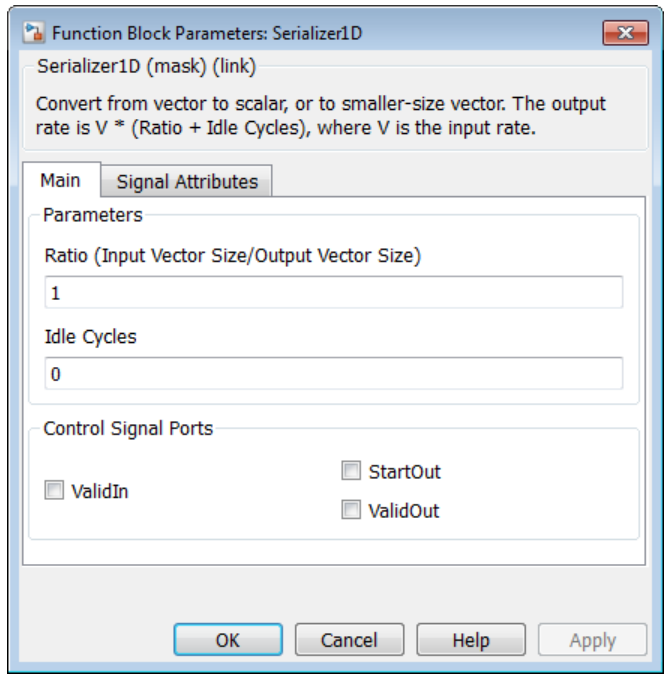
For the first input vector, ABCE, ValidIn is false. So the output is zero, StartOut is false, and ValidOut is false. For the second input vector signal, ABCD, ValidIn is true, so A, B, C, and D are serialized into four separate signals. StartOut is true at A to indicate where to start deserialization, and ValidOut is true for all four signals. `Serializer1D` outputs an additional signal, the idle cycle, after the four valid signals. This signal matches the last serialized signal, D, but ValidOut is false.

HDL Code Generation

For simulation results that match the generated HDL code, in the Configuration Parameters dialog box, in the Solver pane, **Tasking mode for periodic sample times** must be `SingleTasking`.

If you simulate this block using `MultiTasking` mode, the output data can update in the same cycle, but in the generated HDL code, the output data is updated one cycle later.

Dialog Box and Parameters



Ratio

Enter the serialization factor. Default is `1`.

The ratio is equal to the size of the input vector divided by the size of the output vector. Input vector size must be divisible by the ratio.

Idle Cycles

Enter the number of idle cycles to add at the end of each output. Default is `0`.

ValidIn

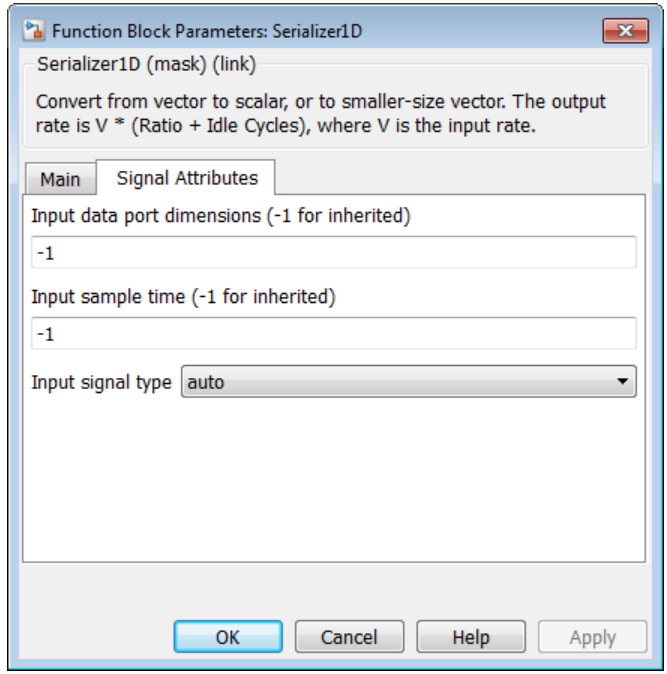
Select to activate the ValidIn port. Default is off.

StartOut

Select to activate the StartOut port. Default is off.

ValidOut

Select to activate the ValidOut port. Default is off.



Input data port dimensions (-1 for inherited)

Enter the size of the input data signal. Input vector size must be divisible by the ratio. By default, the block inherits size based on context within the model.

Input sample time (-1 for inherited)

Enter the time interval between sample time hits, or specify another appropriate sample time such as continuous. By default, the block inherits sample time based on context within the model. For more information, see “Sample Time”.

Input signal type

Specify the input signal type of the block as `auto`, `real`, or `complex`.

Ports

`P`

Input signal to serialize. Bus data types are not supported.

`ValidIn`

Indicates valid input signal. This port is available when you select the **ValidIn** check box.

Data type: Boolean

`S`

Serialized output signal. Bus data types are not supported.

`StartOut`

Indicates where to start deserialization. Use with the `Deserializer1D` block. This port is available when you select the **StartOut** check box.

Data type: Boolean

`ValidOut`

Indicates valid output signal. Use with the `Deserializer1D` block. This port is available when you select the **ValidOut** check box.

Data type: Boolean

See Also

`Deserializer1D`

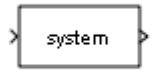
Introduced in R2014b

S-Function

Include S-function in model

Library

User-Defined Functions



Description

The S-Function block provides access to S-functions from a block diagram. The S-function named as the **S-function name** parameter can be a Level-1 MATLAB or a Level-1 or Level-2 C MEX S-function (see "S-Function Basics" for information on how to create S-functions).

Note: Use the `Level-2 MATLAB S-Function` block to include a Level-2 MATLAB S-function in a block diagram.

The S-Function block allows additional parameters to be passed directly to the named S-function. The function parameters can be specified as MATLAB expressions or as variables separated by commas. For example,

```
A, B, C, D, [eye(2,2);zeros(2,2)]
```

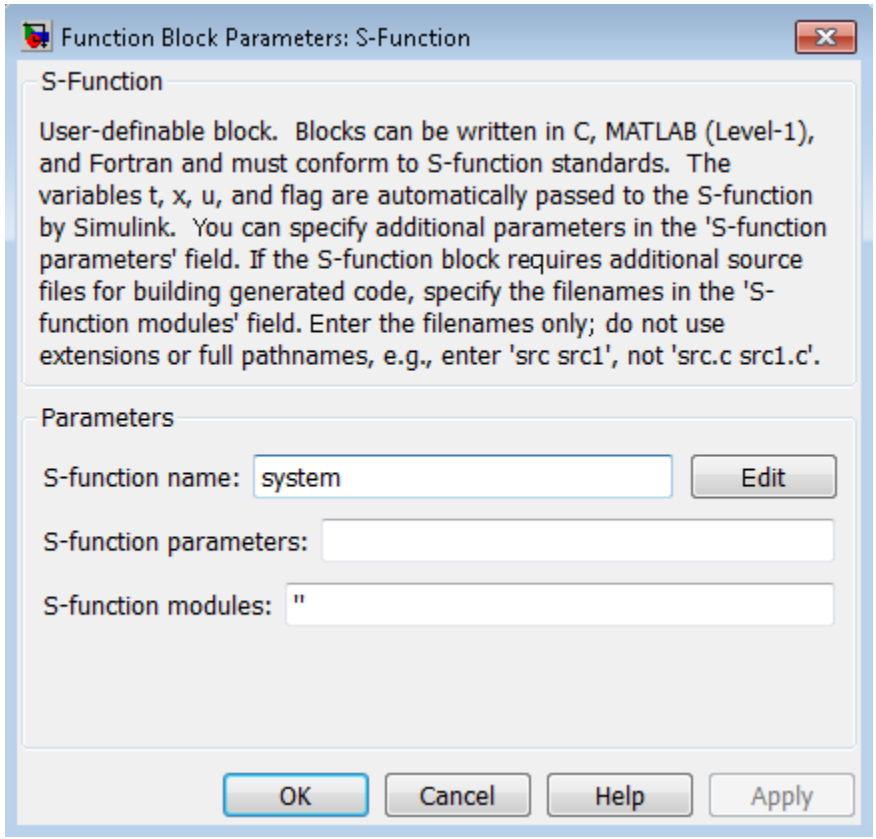
Note that although individual parameters can be enclosed in brackets, the list of parameters must not be enclosed in brackets.

The S-Function block displays the name of the specified S-function and the number of input and output ports specified by the S-function. Signals connected to the inputs must have the dimensions specified by the S-function for the inputs.

Data Type Support

Depends on the implementation of the S-Function block.

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Depends on contents of S-function
Direct Feedthrough	Depends on contents of S-function
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

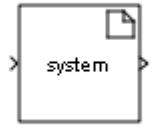
Introduced before R2006a

S-Function Builder

Create S-function from C code that you provide

Library

User-Defined Functions



Description

The S-Function Builder block creates a C MEX S-function from specifications and C source code that you provide. See “Build S-Functions Automatically” for detailed instructions on using the S-Function Builder block to generate an S-function.

Instances of the S-Function Builder block also serve as wrappers for generated S-functions in Simulink models. When simulating a model containing instances of an S-Function Builder block, Simulink software invokes the generated S-function associated with each instance to compute the instance's output at each time step.

Note: The S-Function Builder block does not support masking. However, you can mask a `Subsystem` block that contains an S-Function Builder block. For more information, see “Dynamic Masked Subsystem”.

Data Type Support

The S-Function Builder can accept and output complex, 1-D, or 2-D signals and nonvirtual buses. For each of these cases, the signals must have a data type that Simulink supports.

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

Parameters 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.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

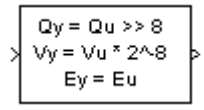
Shift Arithmetic

Shift bits or binary point of signal

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

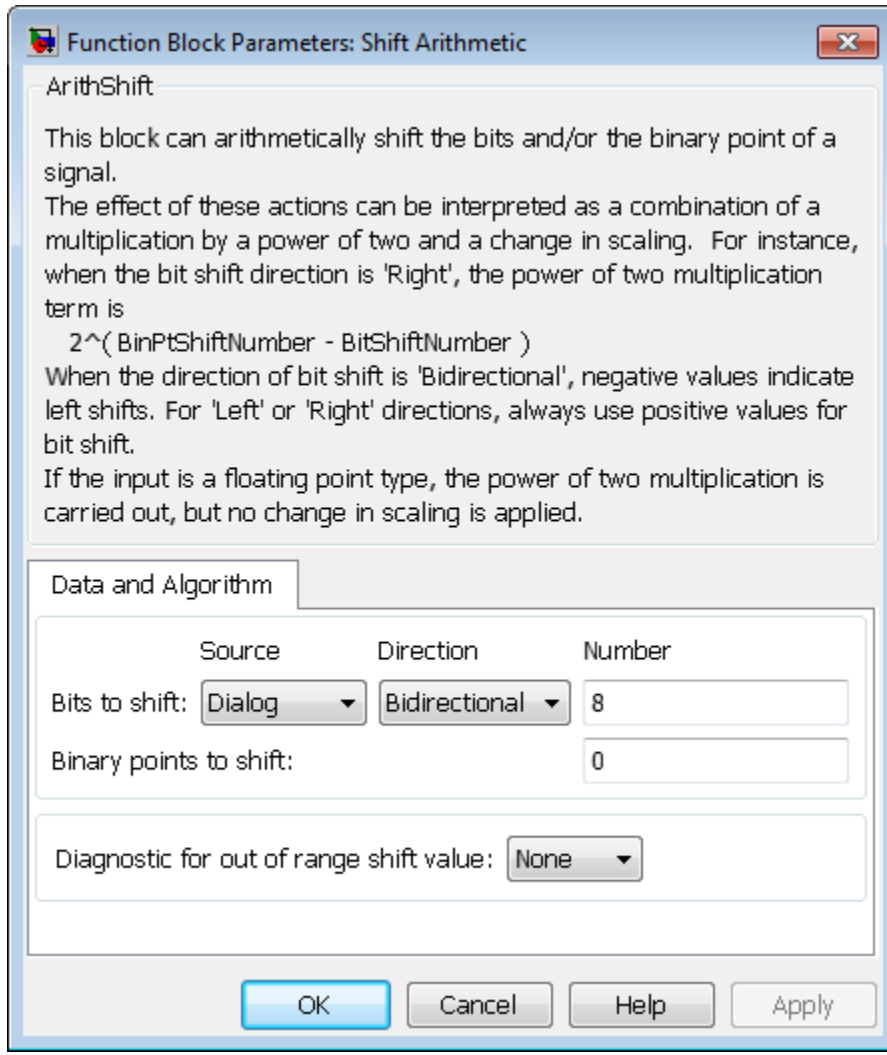
The block parameters support the following data types:

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:



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 Diagnostic Viewer.

For more information, see “Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values” on page 1-1719.

Check for out-of-range 'Bits to shift' in generated code

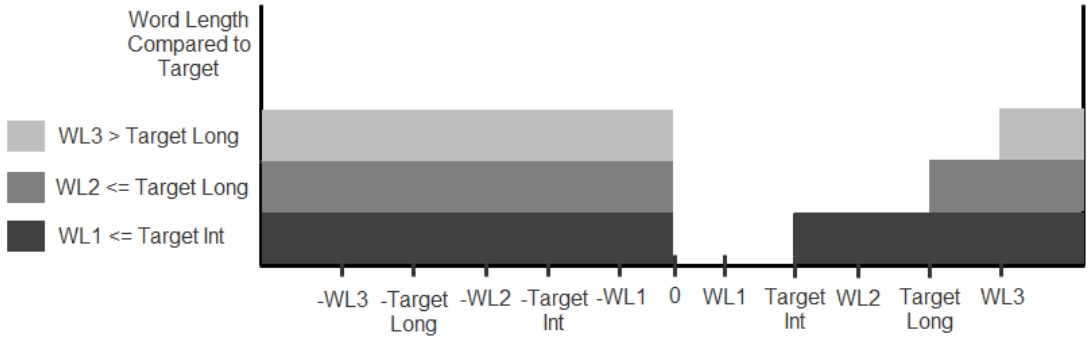
Select this check box to include conditional statements in the generated code that protect against out-of-range bit shift values. This check box is available when **Bits to shift: Source** is `Input port`.

For more information, see “Code Generation for Out-of-Range Bit Shift Values” on page 1-1720.

Out-of-Range Bit Shifts

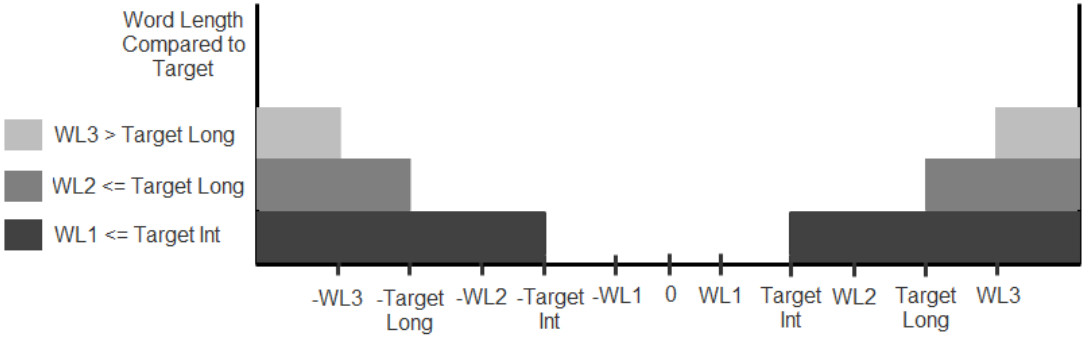
Definition of an Out-of-Range Bit Shift

Suppose that `WL` is the input word length. The shaded regions in the following diagram show out-of-range bit shift values for left and right shifts.



Out-of-Range Bit Shift Values for Left and Right Shifts

Similarly, the shaded regions in the following diagram show out-of-range bit shift values for bidirectional shifts.



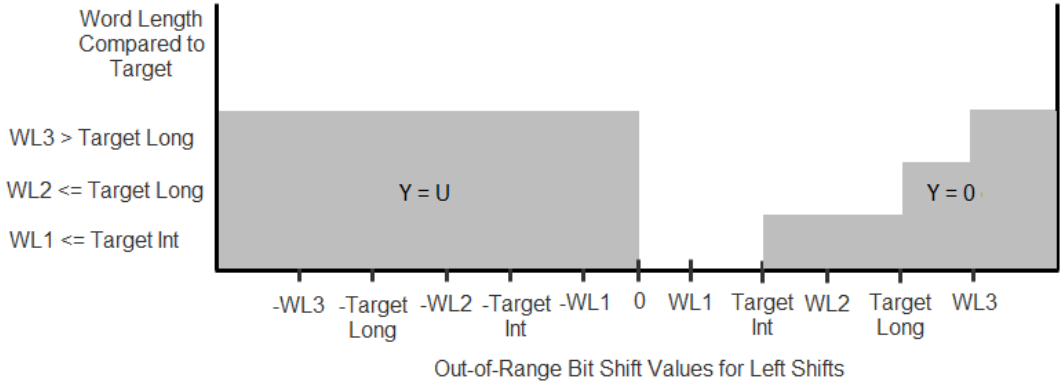
Out-of-Range Bit Shift Values for Bidirectional Shifts

The diagnostic for out-of-range bit shifts responds as follows, depending on the mode of operation:

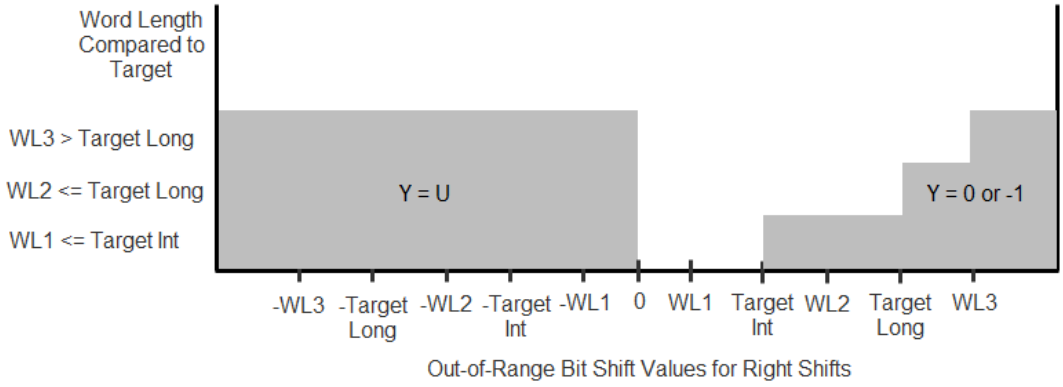
Mode	Diagnostic for out-of-range shift value		
	None	Warning	Error
Simulation	Do not report any warning or error.	Report a warning but continue simulation.	Report an error and stop simulation.
Accelerator modes and code generation	Has no effect.		

Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values

Suppose that `U` is the input, `WL` is the input word length, and `Y` is the output. The output for an out-of-range bit shift value for left shifts is as follows:

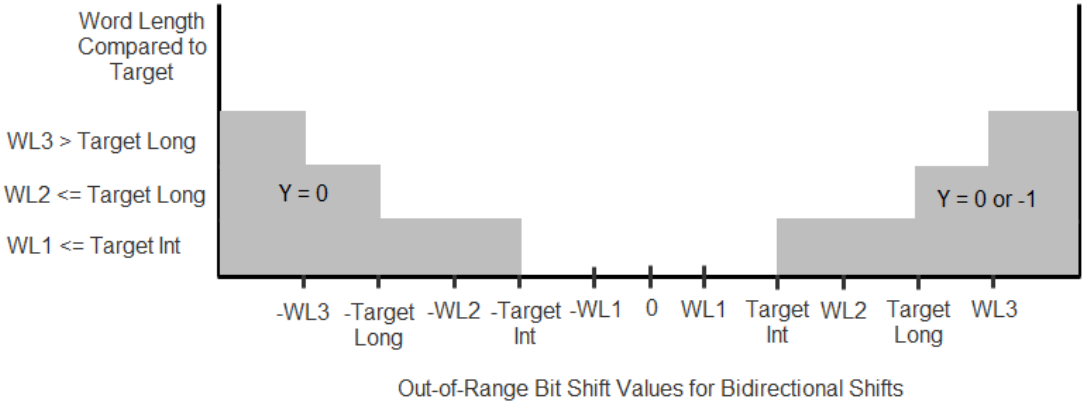


Similarly, the output for an out-of-range bit shift value for right shifts is as follows:



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:



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**.

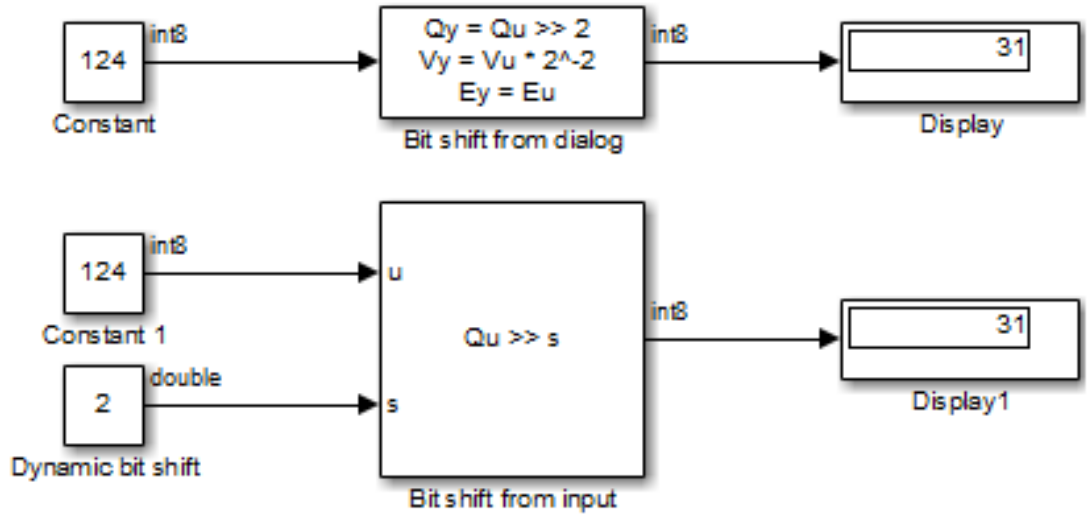
Check Box Setting	Generated Code	Simulation Results Compared to Generated Code
Selected	Includes conditional statements to protect against out-of-range bit shift values.	Simulation and Accelerator mode results match those of code generation.
Cleared	Does not protect against out-of-range bit shift values.	• 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.



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`

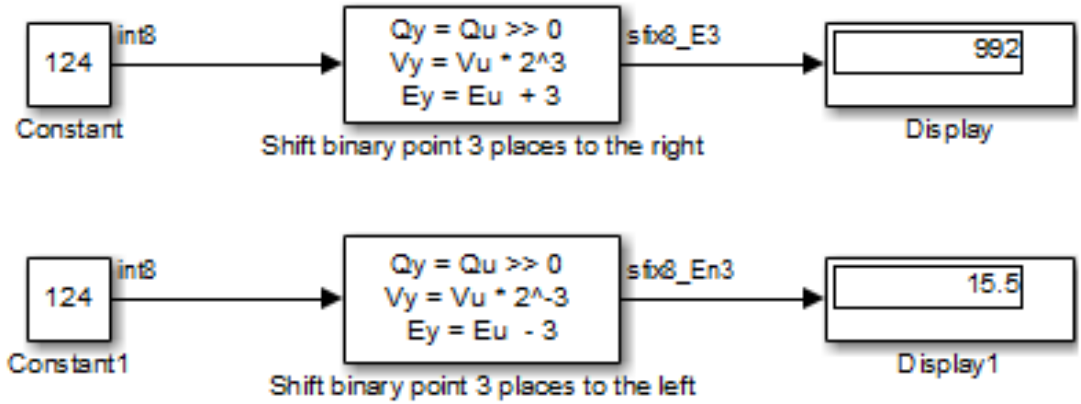
Block	Parameter	Setting
	Bits to shift: Direction	`Right`
	Bits to shift: Number	`2`
`Bit shift from input`	**Bits to shift: Source**	`Input port`
	Bits to shift: Direction	`Right`

The top Shift Arithmetic block takes an input of 124, which corresponds to 01111100 in binary format. Shifting the number of bits two places to the right produces 00011111 in binary format. Therefore, the block outputs 31.

The bottom Shift Arithmetic block performs the same operation as the top block. However, the bottom block receives the bit shift value through an input port instead of the dialog box. By supplying this value as an input signal, you can change the number of bits to shift during simulation.

Block Output for Binary Point Shifts

The following model shows the effect of binary point shifts.



The key block parameter settings of the Constant blocks are:

Block	Parameter	Setting
`Constant` and `Constant1`	**Constant value**	`124`
	Output data type	`int8`

The key block parameter settings of the Shift Arithmetic blocks are:

Block	Parameter	Setting
`Shift binary point 3 places to the right`	**Bits to shift: Source**	`Dialog`
	Bits to shift: Direction	`Bidirectional`
	Bits to shift: Number	`0`
	Binary points to shift: Number	`3`
`Shift binary point 3 places to the left`	**Bits to shift: Source**	`Dialog`
	Bits to shift: Direction	`Bidirectional`
	Bits to shift: Number	`0`

Block	Parameter	Setting
	Binary points to shift: Number	–3

The top Shift Arithmetic block takes an input of 124, which corresponds to 01111100 in binary format. Shifting the binary point three places to the right produces 01111100000 in binary format. Therefore, the top block outputs 995.

The bottom Shift Arithmetic block also takes an input of 124. Shifting the binary point three places to the left produces 01111.100 in binary format. Therefore, the bottom block outputs 15.5.

Characteristics

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

Introduced before R2006a

Sign

Indicate sign of input

Library

Math Operations

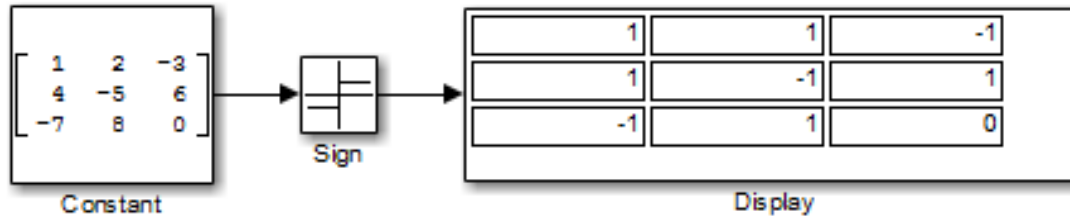
Description

Block Behavior for Real Inputs

For real inputs, the Sign block outputs the sign of the input:

Input	Output
Greater than zero	1
Equal to zero	0
Less than zero	–1

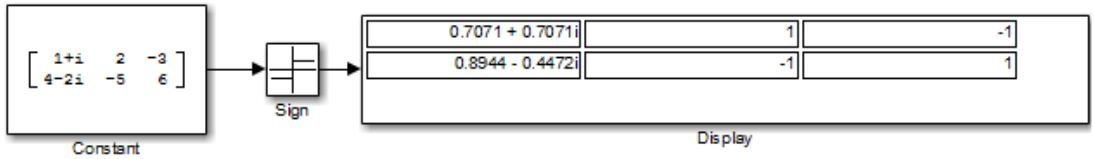
For vector and matrix inputs, the block outputs a vector or matrix where each element is the sign of the corresponding input element, as shown in this example:



Block Behavior for Complex Inputs

When the input `u` is a complex scalar, the block output matches the MATLAB result for:
`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:



Data Type Support

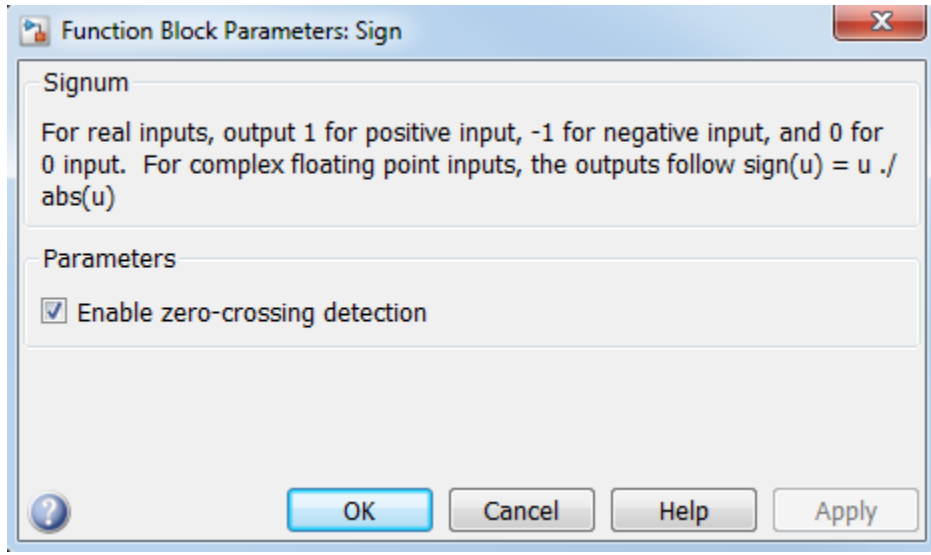
The Sign block supports real inputs of the following data types:

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

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

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

Parameters 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

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

Examples

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

- `sldemo_fuelsys` (in the Engine Gas Dynamics/Throttle & Manifold/Throttle subsystem)
- `sldemo_hardstop`

Characteristics

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

Introduced before R2006a

Signal Builder

Create and generate interchangeable groups of signals whose waveforms are piecewise linear

Library

Sources

Description

The Signal Builder block allows you to create interchangeable groups of piecewise linear signal sources and use them in a model. See “Signal Groups”.

Note: Use the `signalbuilder` function to create and access Signal Builder blocks programmatically.

Data Type Support

The Signal Builder block accepts signals only of type double and outputs a virtual nonhierarchical bus, scalar, or array of real signals of type `double`. It does not support data stores (see “Data Stores”).

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

Parameters 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

Data Types	Double
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.
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes
Code Generation	Yes

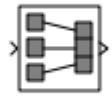
Introduced before R2006a

Signal Conversion

Convert signal to new type without altering signal values

Library

Signal Attributes



Description

The Signal Conversion block converts a signal from one type to another. Use the **Output** parameter to select the type of conversion to perform.

Data Type Support

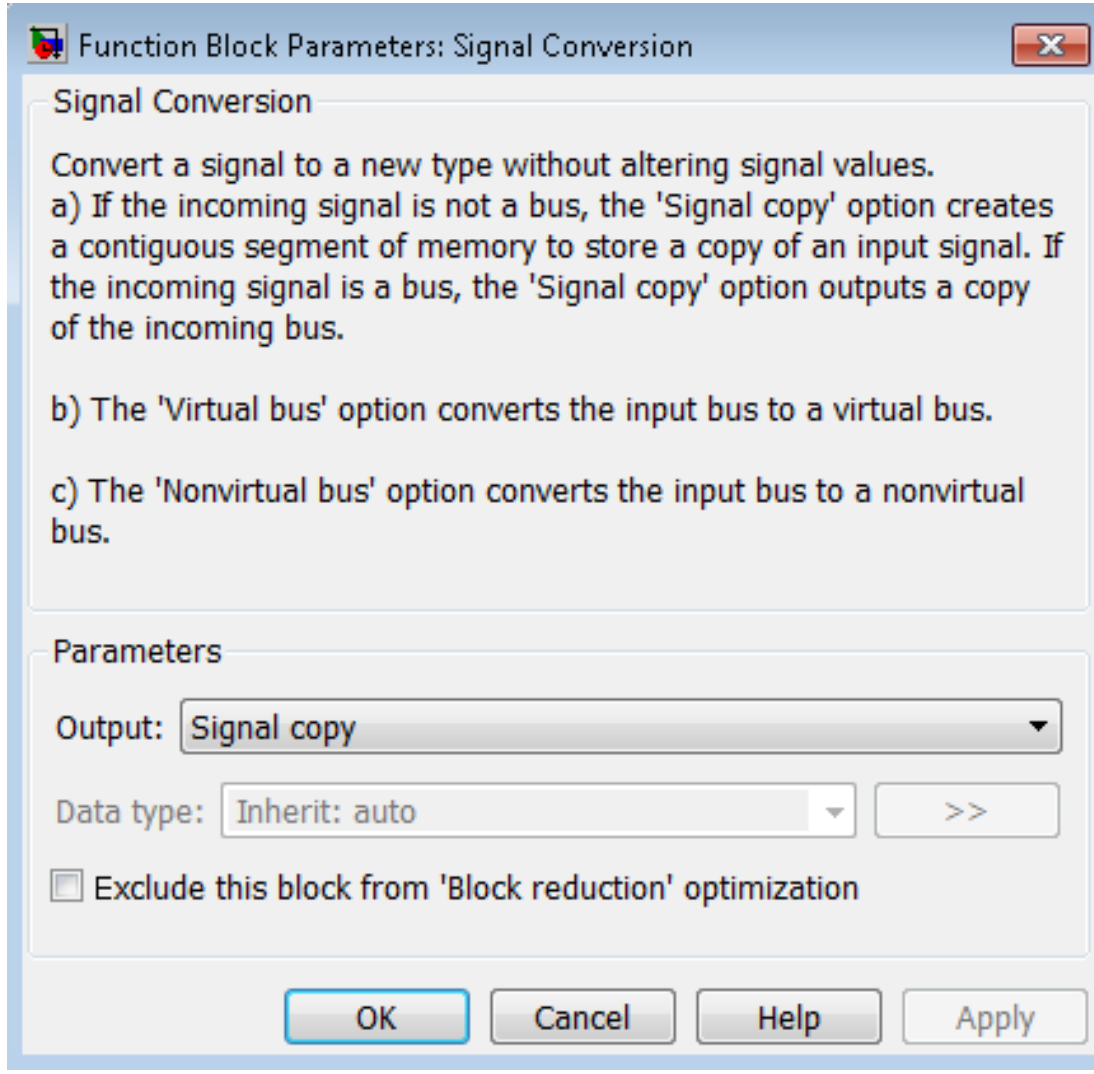
The Signal Conversion block accepts signals of the following data types:

- Scalar
- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated
- Virtual and nonvirtual bus signals

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

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

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 "Use 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. For more information, see "Composite Signals".	`Signal copy`
Save memory by converting a nonvirtual bus to a virtual bus.	`Virtual bus`
Pass a virtual bus signal to a modeling construct that requires a nonvirtual bus, such as a Model block.	`Nonvirtual bus`
Pass a nonvirtual bus signal from a Bus Selector block.	`Nonvirtual bus`

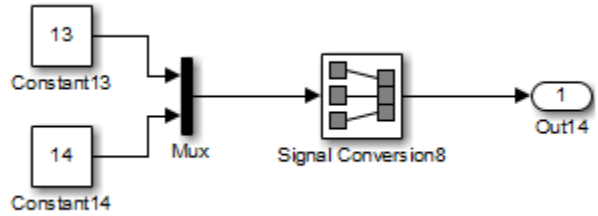
- The `Signal copy` option is the default. The type of conversion that the Signal Conversion block performs using the `Signal copy` option depends on the type of input signal.

Type of Input Signal	Conversion that the Signal Copy Option Performs
Muxed (nonbus) signal	Converts the muxed signal, whose elements occupy 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

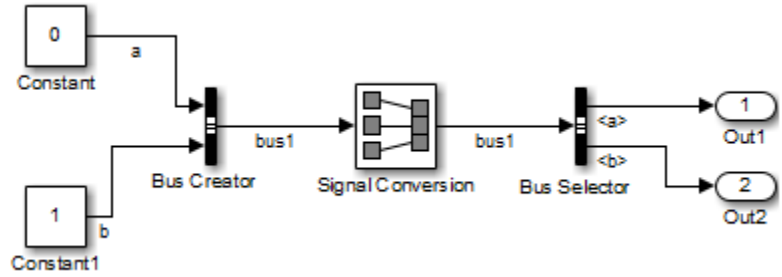
Type of Input Signal	Conversion that the Signal Copy Option Performs
	contiguous areas (represented by the block's output) at each time step.
Bus signal	Outputs a contiguous copy of the bus signal that is the input to the Signal Conversion block.

For an array of buses input signal, use the `Signal copy` option.

In the following example, a muxed signal inputs to a Signal Conversion block that has the **Output** parameter set to `Signal copy`. The Signal Conversion block converts the input signal to a vector.

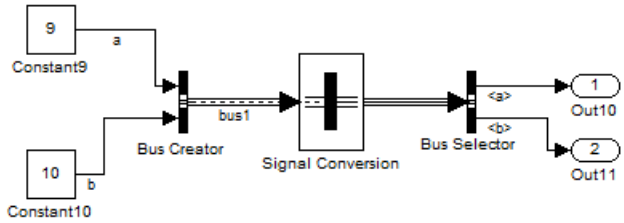


In the following example, a nonvirtual bus signal from a Bus Creator block inputs to a Signal Conversion block that has **Output** set to `Signal copy`. The Signal Conversion block creates another contiguous copy of the input bus signal.



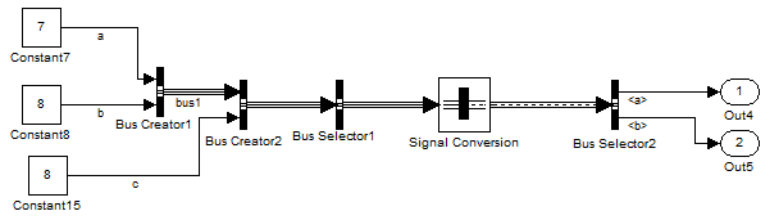
- The `Virtual bus` option converts a nonvirtual bus to a virtual bus.

In the following example, a Bus Creator block inputs to a Signal Conversion block that has **Output** set to `Virtual bus`. The Signal Conversion block converts the nonvirtual bus signal from the Bus Creator block to a virtual bus signal that inputs to the Bus Selector block.



- The `Nonvirtual bus` option converts a virtual bus to a nonvirtual bus.

  In the following example, the Signal Conversion block converts a virtual bus signal from the first Bus Selector block to a nonvirtual bus signal that inputs to the second Bus Selector block. The Signal Conversion block has its **Output** parameter set to `Nonvirtual bus`, and specifies a bus object that matches the bus signal hierarchy of the bus that the first Bus Creator block outputs.



Data type

Specify the output data type of the nonvirtual bus that the Signal Conversion block produces.

This option is available only when you set the **Output** parameter to `Nonvirtual bus`.

The default option is `Inherit: auto`, which uses a rule that inherits a data type.

You must specify a `Simulink.Bus` object in the **Data type** parameter for one or both of the following blocks:

- Signal Conversion block
- An upstream `Bus Creator` block

If you specify a bus object for the Signal Conversion block, but not for its upstream Bus Creator block, then use a bus object that matches the hierarchy of the bus that upstream Bus Creator block outputs.

If you specify a bus object for both the Signal Conversion block and its upstream Bus Creator block, use the same bus object for both blocks.

You can select the button to the right of the Data type parameter to open the Data Type Assistant, which helps you to set the **Data type** parameter.

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

Exclude this block from 'Block reduction' optimization

This option is available only when you set the **Output** parameter to `Signal copy`. If the elements of the input signal occupy contiguous areas of memory, then as an optimization, Simulink software eliminates the block from the compiled model . If you select the **Exclude this block from 'Block reduction' optimization** check box, the optimization occurs the next time you compile the model. For more information, see “Block reduction”.

Characteristics

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

See Also

`Bus Creator` | `Data Type Conversion`

More About

- "Buses"

Introduced before R2006a

Signal Generator

Generate various waveforms

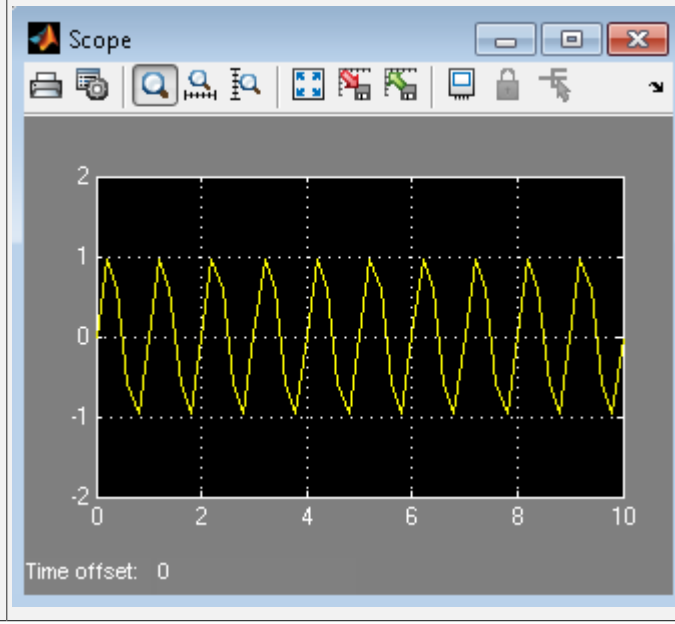
Library

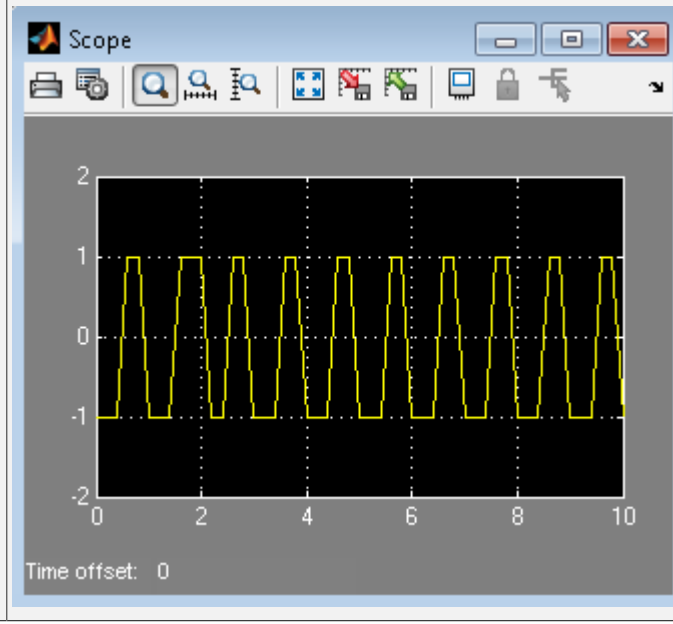
Sources

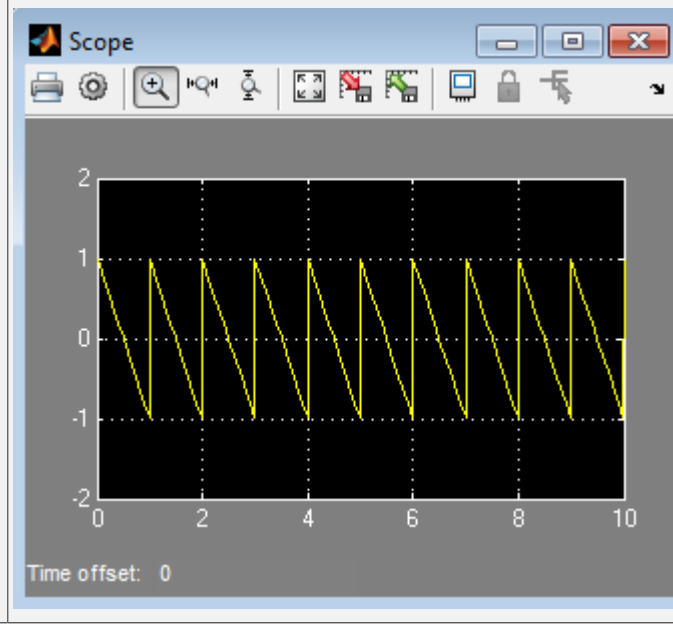
Description

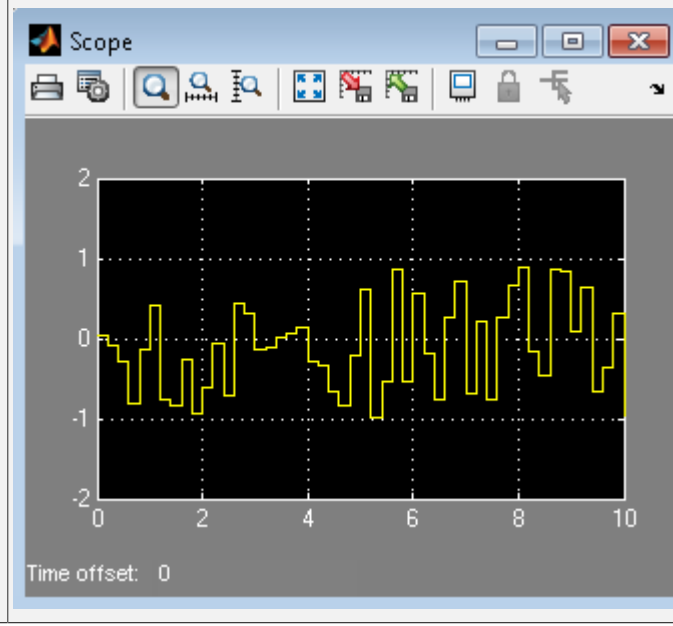
Supported Operations

The Signal Generator block can produce one of four different waveforms: sine wave, square wave, sawtooth wave, and random wave. You can express signal parameters in Hertz (the default) or radians per second. Using default parameter values, you get one of the following waveforms:

Waveform	Scope Output
Sine wave	

Waveform	Scope Output
Square wave	

Waveform	Scope Output
Sawtooth wave	

Waveform	Scope Output
Random wave	

A negative **Amplitude** parameter value causes a 180-degree phase shift. You can generate a phase-shifted wave at other than 180 degrees in many ways. For example, you can connect a Clock block signal to a `MATLAB Function` block and write the equation for the specific wave.

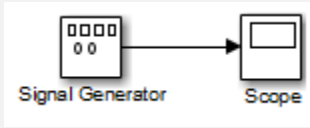
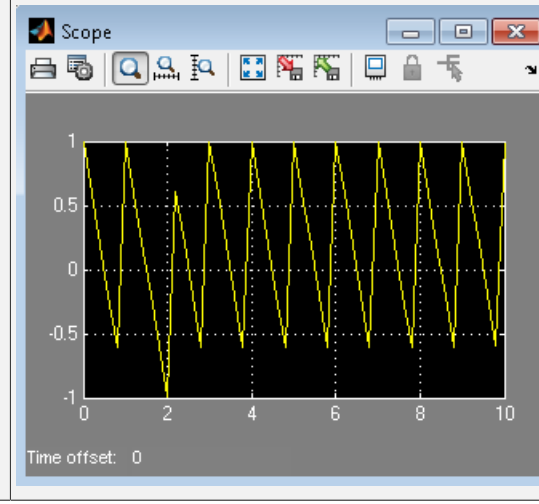
You can vary the output settings of the Signal Generator block while a simulation is in progress. This is useful to determine quickly the response of a system to different types of inputs.

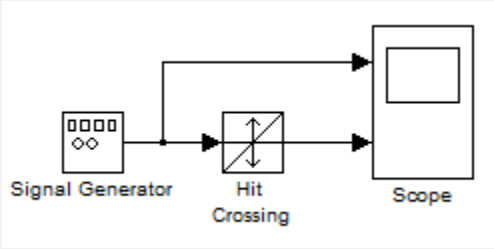
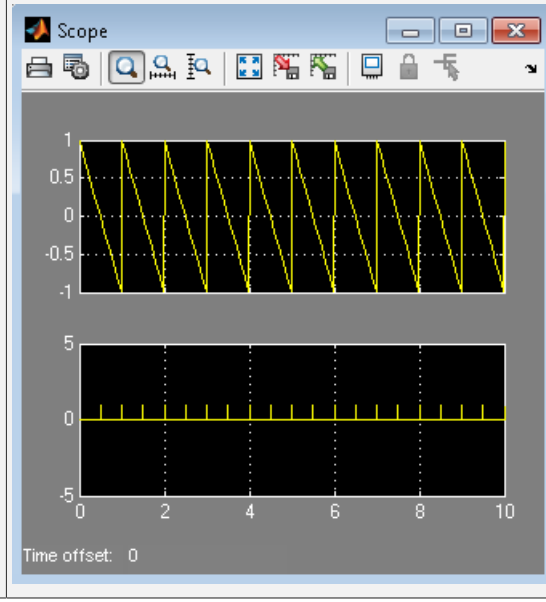
The **Amplitude** and **Frequency** parameters determine the amplitude and frequency of the output signal. The parameters must be of the same dimensions after scalar expansion. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensions as the **Amplitude** and **Frequency** parameters (after scalar expansion). If you select the **Interpret vector parameters as 1-D** check box, the block outputs a vector (1-D) signal if the **Amplitude** and **Frequency** parameters are row or column vectors, that is, single row or column 2-D arrays. Otherwise, the block outputs a signal of the same dimensions as the parameters.

Solver Considerations

If your model uses a fixed-step solver, Simulink uses the same step size for the entire simulation. In this case, the Signal Generator block output provides a uniformly sampled representation of the ideal waveform.

If your model uses a variable-step solver, Simulink might use different step sizes during the simulation. In this case, the Signal Generator block output does not always provide a uniformly sampled representation of the ideal waveform. To ensure that the block output is a uniformly sampled representation, add a `Hit Crossing` block directly downstream of the Signal Generator block. The following models show the difference in Signal Generator block output with and without the Hit Crossing block.

Model That Uses a Variable-Step Solver	Signal Generator Block Output
	

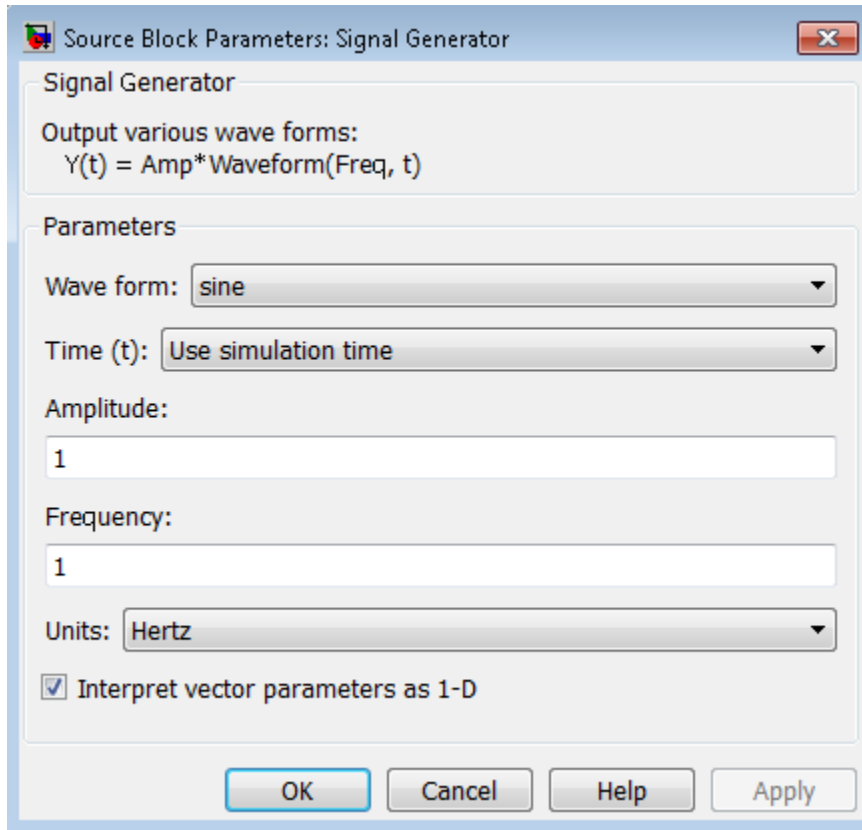
Model That Uses a Variable-Step Solver	Signal Generator Block Output
	

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.

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 as `Hertz` or `rad/sec`. The default is `rad/sec`.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the **Amplitude** and **Frequency** parameters result in a vector output signal (see "Determining the Output Dimensions of Source Blocks"). This option is not available when an external signal specifies time. In this case, if the **Amplitude** and **Frequency** parameters are column or row matrix values, the output is a 1-D vector.

Examples

The following Simulink examples show how to use the Signal Generator block:

- `sldemo_dblcart1`
- `slexAircraftExample`
- `penddemo`

Characteristics

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

See Also

`Pulse Generator`

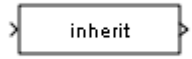
Introduced before R2006a

Signal Specification

Specify desired dimensions, sample time, data type, numeric type, and other attributes of signal

Library

Signal Attributes



Description

The Signal Specification block allows you to specify the attributes of the signal connected to its input and output ports. If the specified attributes conflict with the attributes specified by the blocks connected to its ports, Simulink software displays an error when it compiles the model. For example, at the beginning of a simulation, if no conflict exists, Simulink eliminates the Signal Specification block from the compiled model. In other words, the Signal Specification block is a virtual block. It exists only to specify the attributes of a signal and plays no role in the simulation of the model.

You can use the Signal Specification block to ensure that the actual attributes of a signal meet desired attributes. For example, suppose that you and a colleague are working on different parts of the same model. You use Signal Specification blocks to connect your part of the model with your colleague's. If your colleague changes the attributes of a signal without informing you, the attributes entering the corresponding Signal Specification block do not match. When you try to simulate the model, you get an error.

You can also use the Signal Specification block to ensure correct propagation of signal attributes throughout a model. The capability of allowing the Simulink to propagate attributes from block to block is powerful. However, if some blocks have unspecified attributes for the signals they accept or output, the model does not have enough information to propagate attributes correctly. For these cases, the Signal Specification block is a good way of providing the information Simulink needs. Using the Signal Specification block also helps speed up model compilation when blocks are missing signal attributes.

The Signal Specification block supports signal label propagation.

Data Type Support

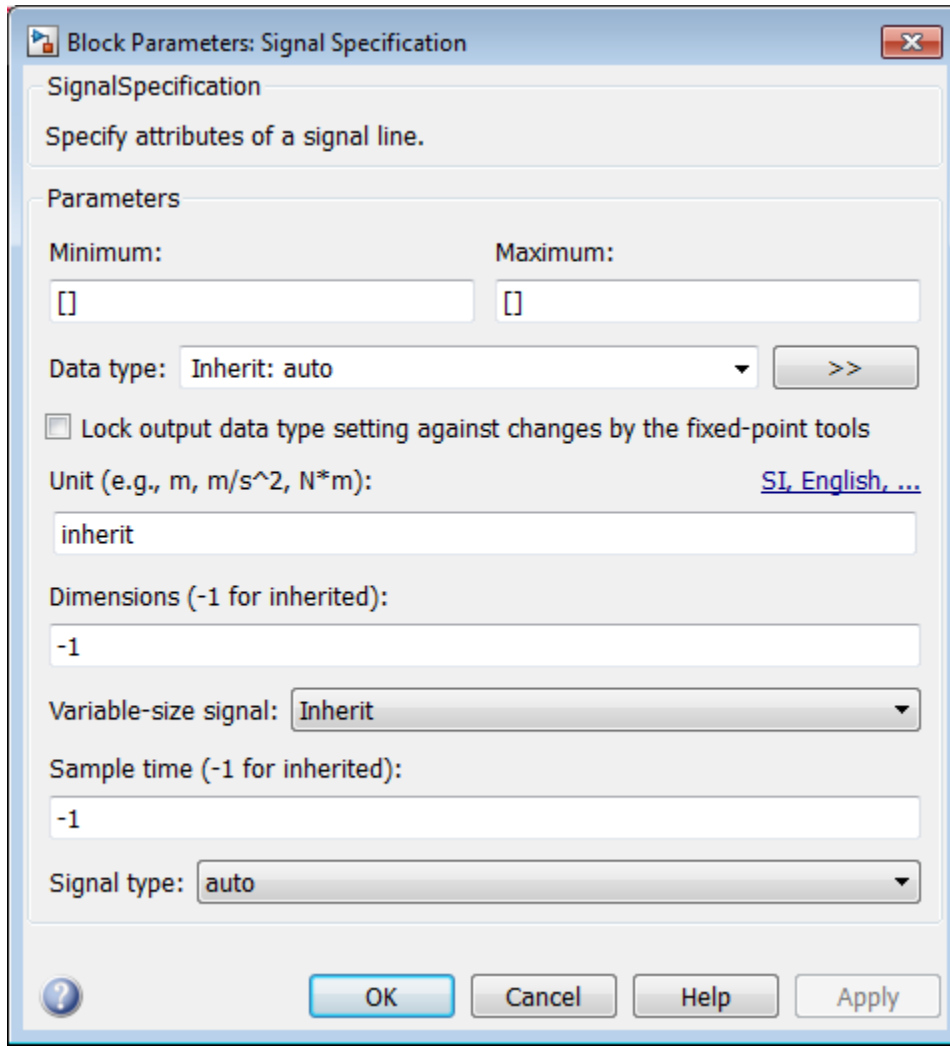
The Signal Specification block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types, as well as bus objects. The input data type must match the data type specified by the **Data type** parameter.

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

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

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

Parameters and Dialog Box



- "Minimum" on page 1-1754
- "Maximum" on page 1-1755
- "Data type" on page 1-1756

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>`

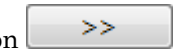
Data type is a bus object.

`<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

"Control Signal 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 “Control Signal Data Types”.

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`

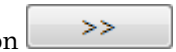
Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus 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 6-101 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 6-101 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 6-101 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**

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 6-101 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 6-101 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 6-101 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: `'off'`

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

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

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

See Also

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

Unit (e.g., m, m/s^2, N*m)

Specify physical unit of the input signal to the block.

Settings

Default: `inherit`

To specify a unit, begin typing in the text box. As you type, the parameter displays potential unit string matches. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

- If a `Unit System Configuration` block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a `Unit System Configuration` block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

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 "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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified by the **Sample time** parameter
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

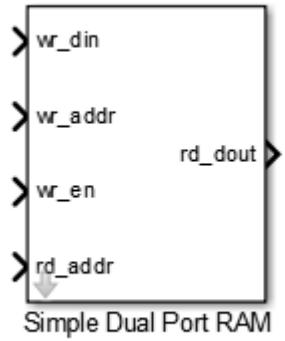
Introduced before R2006a

Simple Dual Port RAM

Dual port RAM with single output port

Library

HDL Coder / HDL Operations



Description

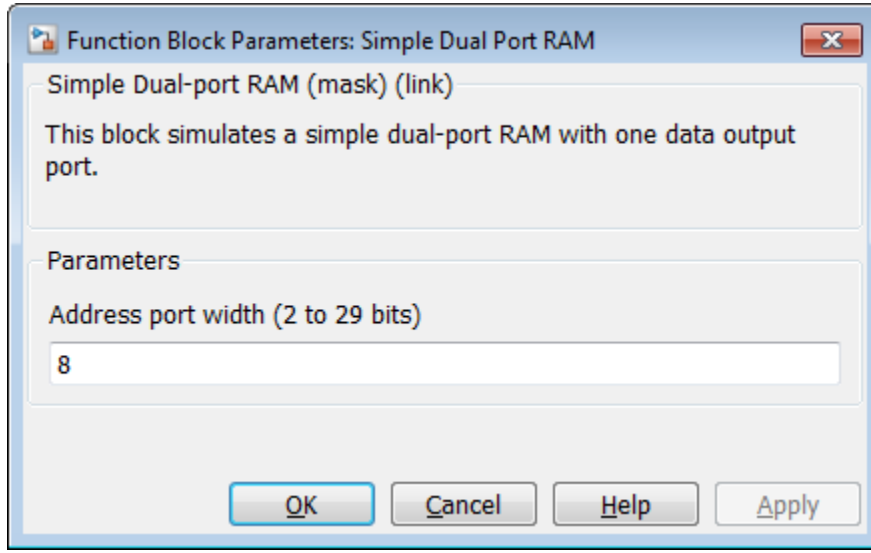
The `Simple Dual Port RAM` block models RAM that supports simultaneous read and write operations, and has a single output port for read data. You can use this block to generate HDL code that maps to RAM in most FPGAs.

The `Simple Dual Port RAM` is similar to the `Dual Port RAM`, but the `Dual Port RAM` has both a write data output port and a read data output port.

Read-During-Write Behavior

During a write operation, if a read operation occurs at the same address, old data appears at the output.

Dialog Box and Parameters



Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

Ports

The block has the following ports:

`wr_din`

Write data input. The data can have any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`wr_addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of `0`

`wr_en`

Write enable.

Data type: Boolean

`rd_addr`

Read address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of `0`

`rd_dout`

Output data from read address, `rd_addr`.

See Also

`Dual Port RAM` | `Dual Rate Dual Port RAM` | `Single Port RAM`

Introduced in R2014a

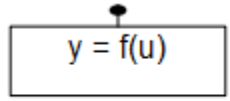
Simulink Function

Function definition used by Function Caller block or Stateflow chart

Library

User-Defined Functions

Description



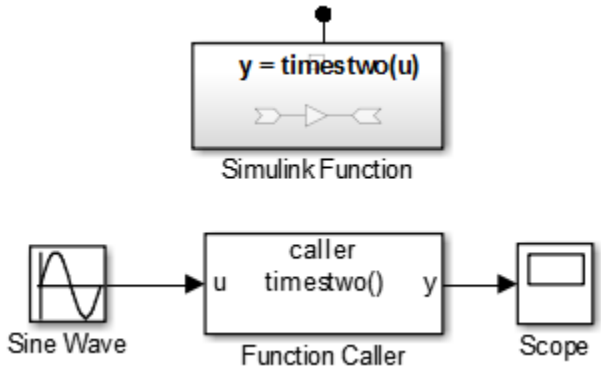
Simulink Function

The Simulink Function block defines a function using Simulink blocks within it. You can call this function from a Function Caller block, MATLAB Function block, or a Stateflow chart. Place a Simulink Function block in the root level of a model or in the root level of a Model block (referenced model). Do not place a Simulink Function block in a Subsystem block.

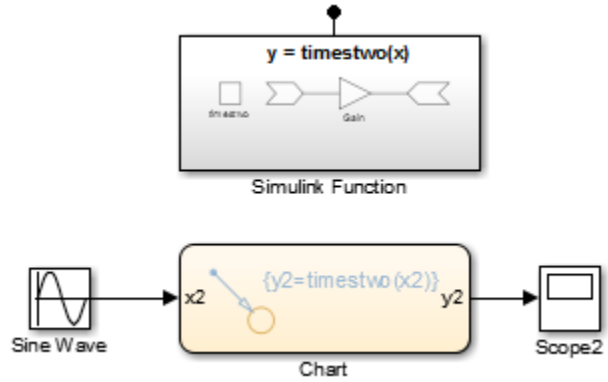
Function interface. The function interface appears on the face of a Simulink Function block. The block also contains blocks that correspond to parts of the functions. For example, editing the block text, adds and deletes Argument Inport and Argument Outport blocks to match the function interface. Editing also sets the **Function name** parameter in the Trigger Port block within the Simulink Function block.

For example, entering `y = myfunction(u)` on the face of the Simulink Function block adds one Argument Inport block (`u`) and one Argument Outport block (`y`) inside the block. It also sets the **Function name** parameter in the Trigger Port block to `myfunction`.

When calling a function using a Function Caller block, the parameter **Function prototype** in the Function Caller block must match exactly the function interface you specify on the Simulink Function block. This match includes the names of input and output arguments. For example, the Simulink Function block and the Function Caller block both use the `u` and `y` argument names.



When calling a function from a Stateflow transition or state label, you can use different argument names. For example, the Simulink Function block uses `x` and `y` arguments, but the Stateflow transition use `x2` and `y2` arguments to call the function.



Connection to local signals. In addition to Argument Inport and Argument Outport blocks, a Simulink Function block can interface to signals in the local environment of the block through Inport or Outport blocks. These signals are hidden from the caller. You can use them to connect and communicate between two Simulink Function blocks or connect to root Inport and Outport blocks to represent external I/O. Typically, you connect signal outputs to Scope or To Workspace blocks.

You can connect the output from a Simulink Function block to sink blocks including logging (To File, To Workspace) and viewing (Scope, Display) blocks. However, these blocks execute last after all other blocks.

Differences Between Function-Call Subsystem and Simulink Function Blocks. In general, a Function-Call Subsystem block provides better signal traceability because you use direct signal connections to represent the triggering condition and the I/O. Simulink Function blocks provide better packaging and eliminate the need for routing input and output signal lines through the model hierarchy.

Attribute	Function-Call Subsystem block	Simulink Function block
Method of executing/invoking function	Triggered using a signal line	Called by reference to the function name
Formal input arguments (Argument Inport blocks) and output arguments (Argument Outport blocks)	No	Yes
Graphical (signal) inputs (Inport block) and outputs (Outport block)	Yes	Yes

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

See Also

`Argument Outport` | `Argument Inport` | `Function Caller` | `Function-Call Subsystem` | `Subsystem`

Related Examples

- “Simulink Functions in Simulink Models”
- “Diagnostics Using a Client-Server Architecture”

Introduced in R2014b

Single Port RAM

Single port RAM

Library

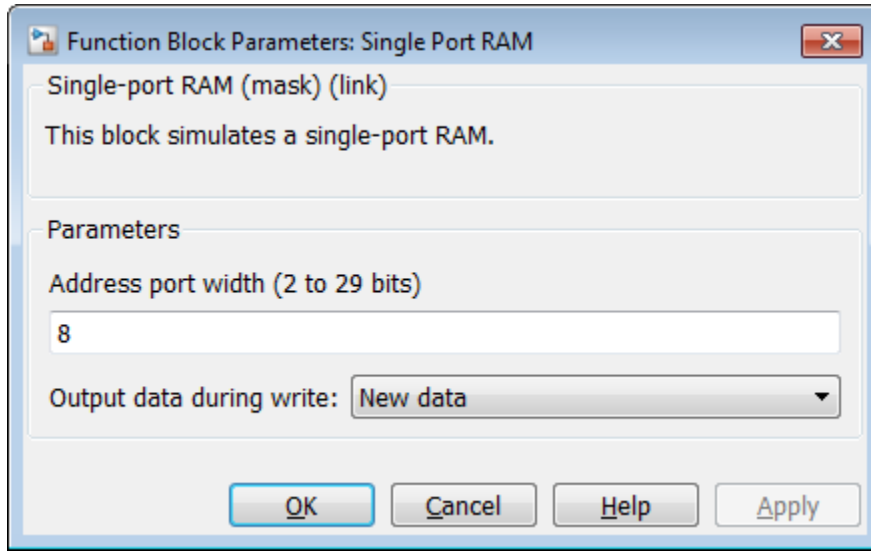
HDL Coder / HDL Operations

Description

The `Single Port RAM` block models RAM that supports sequential read and write operations.

If you want to model RAM that supports simultaneous read and write operations, use the `Dual Port RAM` or `Simple Dual Port RAM`.

Dialog Box and Parameters



Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

Output data during write

Controls the output data, `dout`, during a write access.

- `New data` (default): During a write, new data appears at the output port, `dout`.
- `Old data`: During a write, old data appears at the output port, dout.

Ports

The block has the following ports:

`din`

Data input. The data can have any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of `0`

`we`

Write enable.

Data type: Boolean

`dout`

Output data from address, `addr`.

See Also

`Dual Port RAM` | `Dual Rate Dual Port RAM` | `Simple Dual Port RAM`

Introduced in R2014a

Sine, Cosine

Implement fixed-point sine or cosine wave using lookup table approach that exploits quarter wave symmetry

Library

Lookup Tables (Sine block or Cosine block)

Description

The Sine and Cosine block implements a sine and/or cosine wave in fixed point using a lookup table method that exploits quarter wave symmetry.

The Sine and Cosine block can output the following functions of the input signal, depending upon what you select for the **Output formula** parameter:

- $\sin(2\pi u)$
- $\cos(2\pi u)$
- $\exp(i2\pi u)$
- $\sin(2\pi u)$ and $\cos(2\pi u)$

You define the number of lookup table points in the **Number of data points for lookup table** parameter. The block implementation is most efficient when you specify the lookup table data points to be `(2^n)+1`, where n is an integer.

Tip To obtain meaningful block output, the block input values should fall within the range [0, 1). For input values that fall outside this range, the values are cast to an unsigned data type, where overflows wrap. For these out-of-range inputs, the block output might not be meaningful.

Use the **Output word length** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

Data Type Support

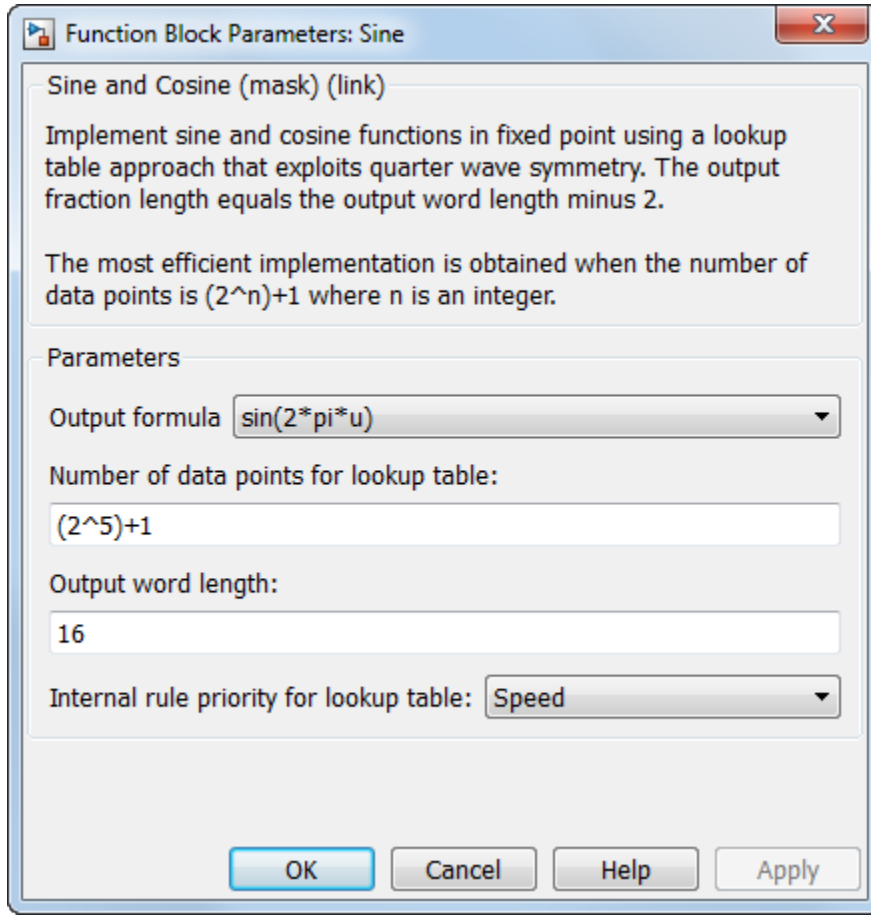
The Sine and Cosine block accepts signals of the following data types:

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

The output of the block is a fixed-point data type.

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

Parameters 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.

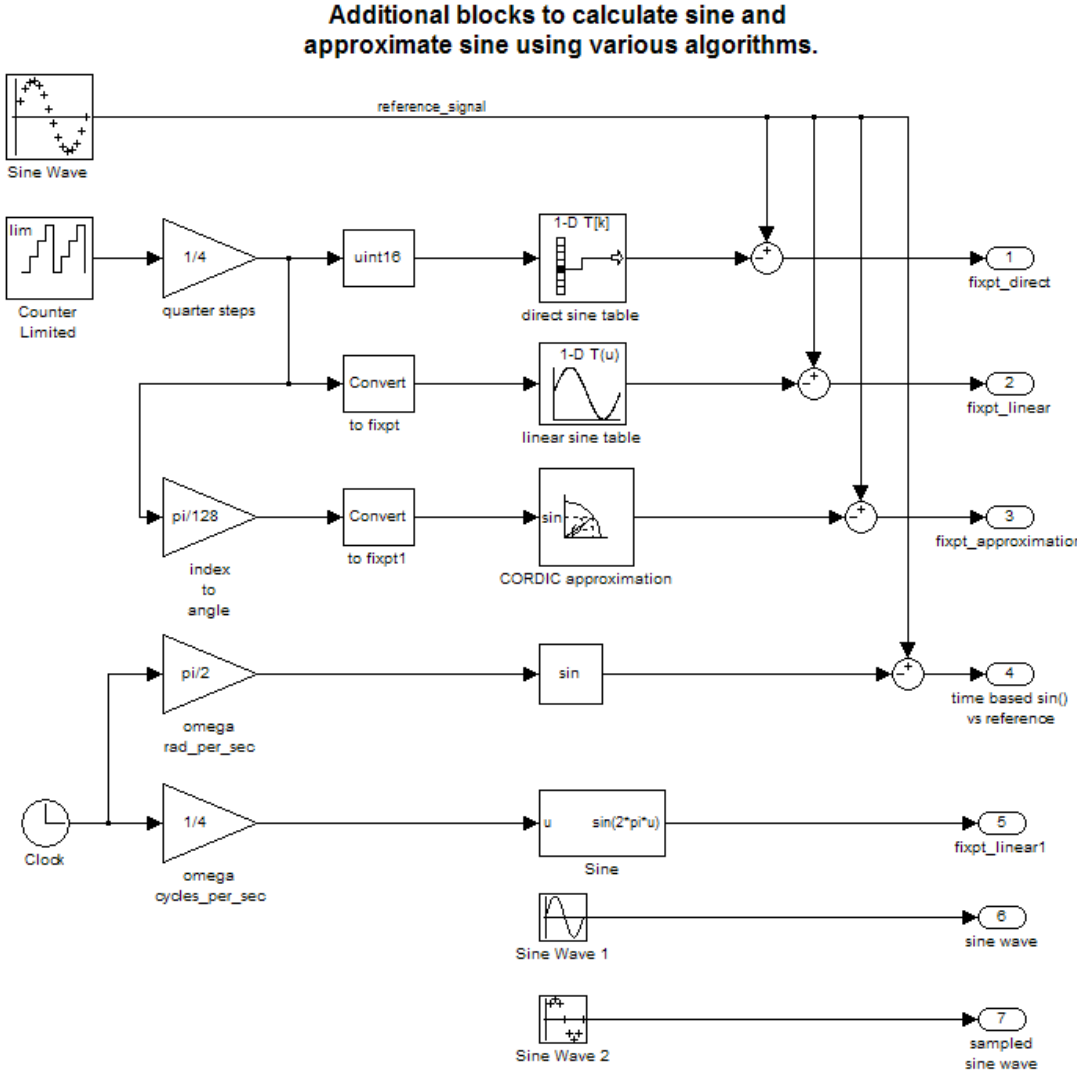
Note: The block uses double-precision floating-point values to construct lookup tables. Therefore, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length to values greater than 53 bits does not improve the precision of your output.

Internal rule priority for lookup table

Specify the internal rule for intermediate calculations. Select `Speed` for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

Examples

The `sldemo_tonegen_fixpt` model shows how you can use the `Sine` block to implement a fixed-point sine wave.


Additional blocks to calculate sine and approximate sine using various algorithms.
Sine Wave
reference_signal
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

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

See Also

`Sine Wave`, `Trigonometric Function`

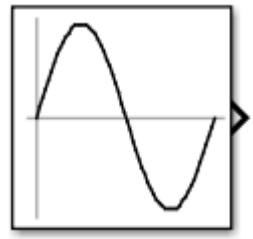
Introduced before R2006a

Sine Wave

Generate sine wave, using simulation time as time source

Library

Sources



Description

The Sine Wave block outputs a sinusoidal waveform. The block can operate in time-based or sample-based mode.

Note: This block is the same as the `Sine Wave Function` block that appears in the Math Operations library. If you select `Use external signal` for the **Time** parameter in the block dialog box, you get the Sine Wave Function block.

Time-Based Mode

The output of the Sine Wave block is determined by:

$$y = amplitude \times \sin(frequency \times time + phase) + bias.$$

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 block output at a given time step does not depend on the output of the previous time steps. Therefore, this mode avoids the accumulation of round-off errors. Additionally, sample-based mode supports reset semantics in subsystems that offer it. For example, if a Sine Wave block is in a resettable subsystem that receives a reset trigger, the repeating integer `k` resets and the block output resets to its initial condition.

Parameter Dimensions

The numeric parameters of this block must have the same dimensions after scalar expansion.

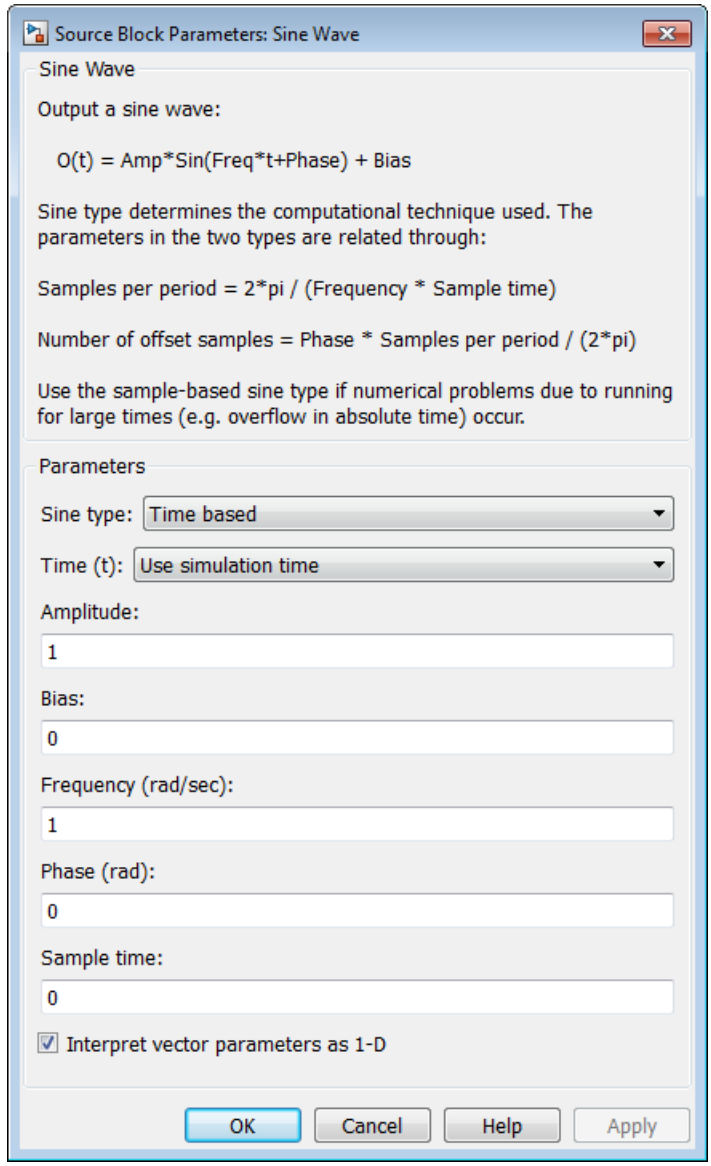
- If **Interpret vector parameters as 1-D** is not selected, the block outputs a signal of the same dimensions and dimensionality as the parameters.
- If **Interpret vector parameters as 1-D** is selected and the numeric parameters are row or column vectors, the block outputs a vector signal. Otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support

The Sine Wave block accepts and outputs real signals of type `double`.

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

Parameters 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

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

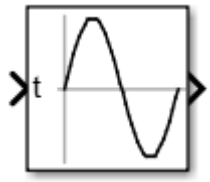
Introduced before R2006a

Sine Wave Function

Generate sine wave, using external signal as time source

Library

Math Operations



Description

This block is the same as the `Sine Wave` block that appears in the Sources library. If you select `Use simulation time` for the **Time** parameter in the block dialog box, you get the Sine Wave block. See the documentation for the Sine Wave block for more information.

Characteristics

Data Types	Double
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Slider

Set value on sliding scale to tune parameters or variables

Library

Dashboard

Description

The `Slider` block enables you to control the value of tunable parameters and variables in your model during simulation.

To control a tunable parameter or variable using the `Slider` block, double-click the `Slider` block to open the dialog box. Select a block in the model canvas. The tunable parameter or variable appears in the dialog box **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable to the block.

The tick range determines the range of values available for the tunable parameter or variable. You can modify the tick range by modifying the **Minimum**, **Maximum**, and **Tick Interval** values.

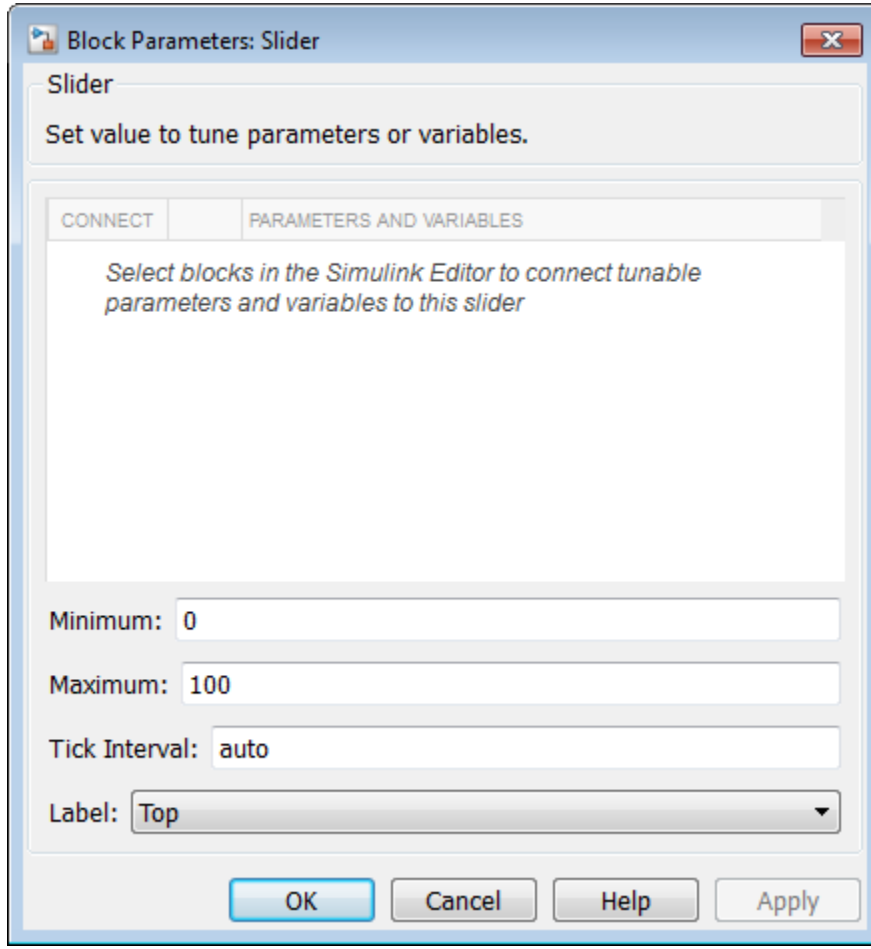
Limitations

The `Slider` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.
Parameters that index a variable array do not appear in the **Connection** table.	For example, a block parameter specified using the variable `engine(1)` will not appear in the table because the parameter

Limitation	Workaround
	uses an index of the variable `engine`, which is not a scalar variable. To make the parameter appear in the **Connection** table, change the block parameter field to a scalar variable, such as `engine_1`.

Parameters and Dialog Box



Connection

Select a block to connect and control a tunable parameter or variable.

To control a tunable parameter or variable, select a block in the model. The tunable parameter or variable appears in the **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable.

Settings

The table has a row for the tunable parameter or variable connected to the block. If there are no tunable parameters or variables selected in the model or the block is not connected to any tunable parameters or variables, then the table is empty.

Minimum

Minimum tick mark value.

Settings

Default: 0

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

Dependencies

The **Minimum** tick value must be less than the **Maximum** tick value.

Maximum

Maximum tick mark value.

Settings

Default: 100

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

Dependencies

The **Maximum** tick value must be greater than the **Minimum** tick value.

Tick Interval

Interval between major tick marks.

Settings

Default: `auto`

Specify this number as a finite, real, positive, integer, scalar value. Specify as `auto` for the block to adjust the tick interval automatically.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see "Tune and Visualize Your Model with Dashboard Blocks".

Introduced in R2015b

Slider Gain

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

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

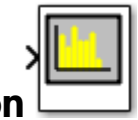
See Also

`Gain`

Introduced before R2006a

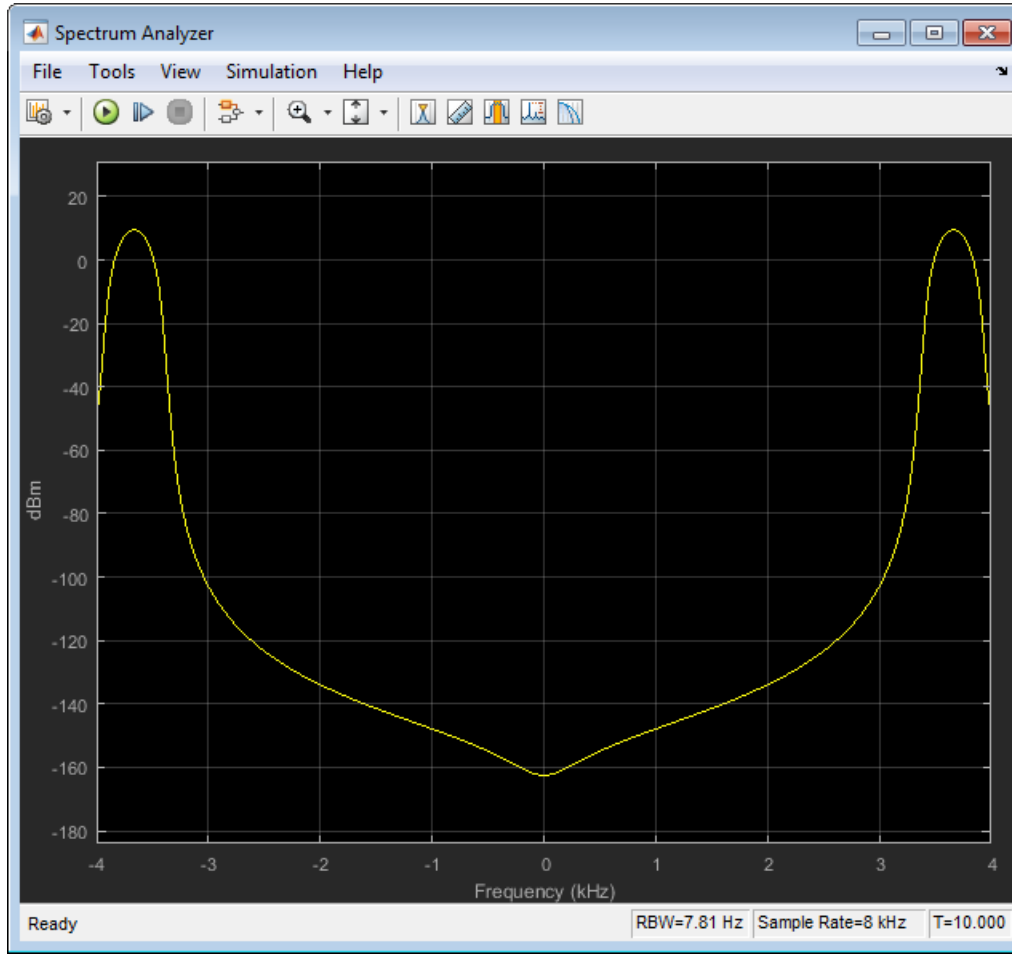
Spectrum Analyzer

Display frequency spectrum of time-domain signals

Description 

The Spectrum Analyzer block, referred to here as the scope, displays frequency spectra of signals. The Spectrum Analyzer block accepts input signals, through one or more input ports, with the following characteristics:

- Discrete sample time
- Real- or complex-valued
- Fixed number of channels of variable length
- Floating- or fixed-point data type



You can use the Spectrum Analyzer block in models running in Normal or Accelerator simulation modes. You can also use the Spectrum Analyzer block in models running in Rapid Accelerator or External simulation modes, with some limitations. See the “Supported Simulation Modes” on page 1-1856 section for more information.

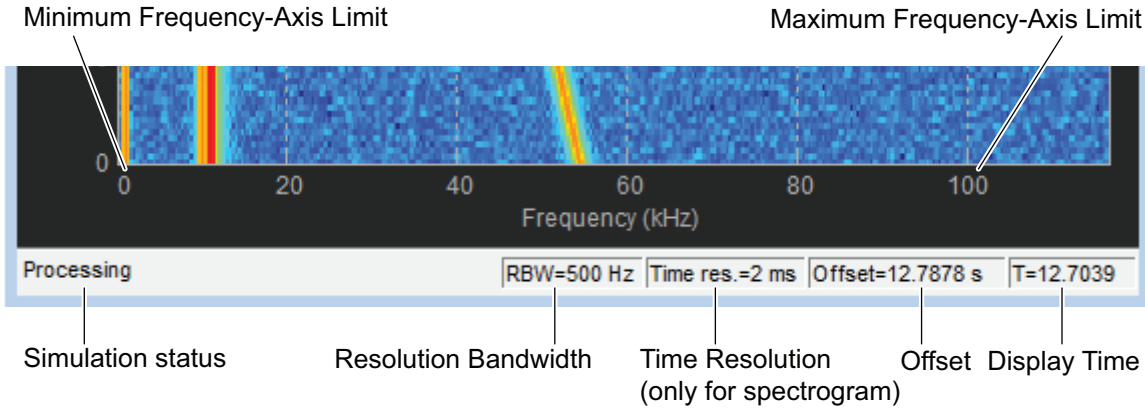
You can use the Spectrum Analyzer block inside of all subsystems and conditional subsystems. *Conditional subsystems* include enabled subsystems, triggered subsystems, enabled and triggered subsystems, and function-call subsystems. See “Conditional Subsystems” in the Simulink documentation for more information.

You can configure and display Spectrum Analyzer settings from the command line with spbscopes.SpectrumAnalyzerConfiguration.

See the following sections for more information on the Spectrum Analyzer:

Signal Display

The Spectrum Analyzer indicates the spectrum computation settings that are represented in the current display. Check the **Resolution Bandwidth**, **Time Resolution**, and **Offset** indicators on the status bar in the scope window for this information. These indicators relate to the Minimum Frequency-Axis limit and Maximum Frequency-Axis limit values on the *frequency*-axis of the scope window. The values specified by these indicators may be changed by modifying parameters in the **Spectrum Settings** panel. You can also view the object state and the amount of time data that correspond to the current display. Check the Simulation Status and Simulation time indicators on the status bar in the scope window for this information. The following figure highlights these aspects of the Spectrum Analyzer window.



Note: To prevent the scope from opening when you run your model, right-click the scope icon and select `Comment Out`. If the scope is already open, and you comment it out in the model. the scope displays, "No data can be shown because this scope is commented out." Select `Uncomment` to turn the scope back on.

- *Frequency Span* — The range of values shown on the *frequency*-axis on the Spectrum Analyzer window.

  Details

  Spectrum Analyzer sets the frequency span using the values of parameters on the **Main options** pane of the **Spectrum Settings** panel.

  - `Span (Hz)` and `CF (Hz)` visible — The *Frequency Span* value equals the **Span** parameter in the **Main options** pane.
  - `FStart (Hz)` and `FStop (Hz)` — The frequency span value equals the difference of the **FStop** and **FStart** parameters in the **Main options** pane, as given by the formula: $f_{span} = f_{stop} - f_{start}$.

  By default, the **Full Span** check box in the **Main options** pane is enabled. In this case, the Spectrum Analyzer computes and plots the spectrum over the entire *Nyquist* frequency interval. When the **Two-sided spectrum**

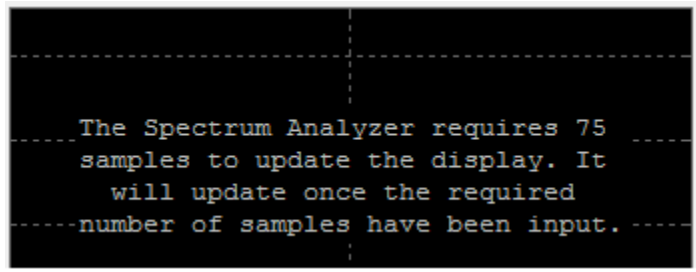
check box in the **Trace options** pane is enabled, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

- *Resolution Bandwidth* — The smallest positive frequency or frequency interval that can be resolved.

Details

Spectrum Analyzer sets the resolution bandwidth using the value of the frequency resolution parameter on the **Main options** pane of the **Spectrum Settings** panel. By default, this parameter is set to `RBW (Hz)` and `'Auto'`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified *Frequency Span*.

You can set the resolution bandwidth to whatever value you choose. For this reason, there is a minimum boundary on the number of input samples required to compute a spectral update. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to *RBW* by the following equation: $N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$. Overlap percentage, O_p, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** panel. *NENBW* is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Window Options** pane. F_s is the sample rate. In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer produces a message on the display, as shown in the following figure.



Spectrum Analyzer removes this message and displays a spectral estimate as soon as enough data has been input.

If the frequency resolution setting on the **Main options** pane of the **Spectrum Settings** is `Window length`, you specify the window length and the resulting RBW is $\frac{NENBW * Fs}{N_{window}}$. The **Samples/update** in this case is directly related to *RBW* by the following equation: $N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$

- *Time Resolution* — The time resolution for a spectrogram line.

  Details

  Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The minimum attainable resolution is the amount of data time it takes to compute a single spectral estimate. *Time Resolution* is displayed only when the spectrum **Type** is `Spectrogram`.

- *Offset* — The constant frequency offset to apply to the entire spectrum or a vector of frequency offsets to apply to each spectrum for multiple inputs.

  Details

  Spectrum Analyzer adds this constant offset or the vector of offsets to the values on the *frequency*-axis using the value of **Offset** on the **Trace options** pane of the **Spectrum Settings** panel. The offset is the current time value at the middle of the interval of the line displayed at 0 seconds. The actual time of a particular spectrogram line is the offset minus the *y*-axis time listing. You must take this parameter into consideration when you set the **Span (Hz)** and **CF (Hz)** parameters on the **Main options** pane of the **Spectrum Settings** panel to ensure that the frequency span is within Nyquist limits. The offset is displayed on the plot only when the spectrum **Type** is `Spectrogram`.

- *Simulation Status* — Provides the current status of the model simulation.

  Details

  The status can be one of the following conditions:

  - Processing — Occurs after you construct the `SpectrumAnalyzer` object.

- Stopped — Occurs after you run the release method.

The *Simulation Status* is part of the *Status Bar* in the Spectrum Analyzer window. You can choose to hide or display the entire *Status Bar*. From the Spectrum Analyzer menu, select **View > Status Bar**.

- *Display time* — The amount of time that has progressed since the last update to the Spectrum Analyzer display.

Details

Every time data is processed by the block, the simulation time increases by the number of rows in the input signal divided by the sample rate, as given by the following formula: $t_{sim} = t_{sim} + \frac{\text{length(0:length(xsine))-1}}{\text{SampleRate}}$. When **Reduce Plot Rate to Improve Performance** is checked, the simulation time and display time might differ. At the beginning of a simulation, you can modify the **SampleRate** parameter on the **Main options** pane of the **Spectrum Settings** panel.

The *Display time* indicator is a component of the *Status Bar* in the Spectrum Analyzer window. You can choose to hide or display the entire *Status Bar*. From the Spectrum Analyzer menu, select **View > Status Bar** .

For more information, see “Spectrum Settings” on page 1-1814.

Reduce Plot Rate to Improve Performance

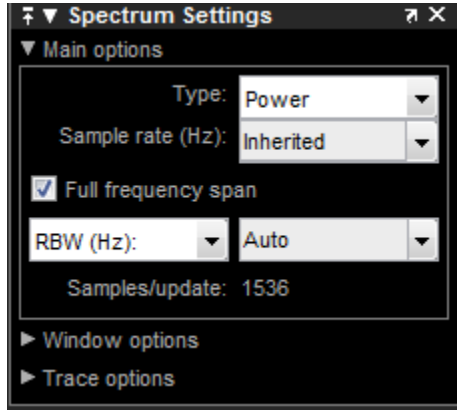
By default, Spectrum Analyzer updates the display at fixed intervals of time at a rate not exceeding 20 hertz. If you want Spectrum Analyzer to plot a spectrum on every simulation time step, you can disable the **Reduce Plot Rate to Improve Performance** option. In the Spectrum Analyzer menu, select **Simulation > Reduce Plot Rate to Improve Performance** to clear the check box. Tunable.

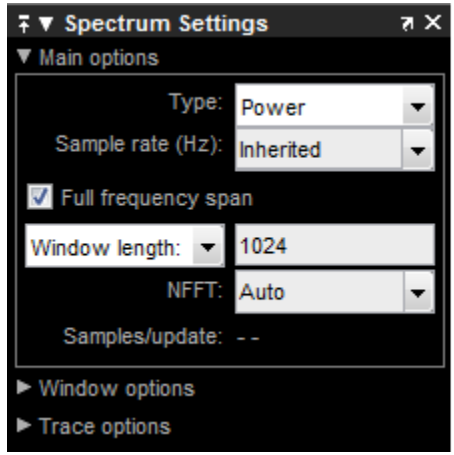
Note: When this check box is selected, Spectrum Analyzer may display a misleading spectrum in some situations. For example, if the input signal is wide-band with non-stationary behavior, such as a chirp signal, Spectrum Analyzer might display a stationary spectrum. The reason for this behavior is that Spectrum Analyzer buffers the input signal data and only updates the display periodically at approximately 20 times per second. Therefore, Spectrum Analyzer does not render changes to the spectrum that

occur and elapse between updates, which gives the impression of an incorrect spectrum. To ensure that spectral estimates are as accurate as possible, clear the **Reduce Plot Rate to Improve Performance** check box. When you clear this box, Spectrum Analyzer calculates spectra whenever there is enough data, rendering results correctly.

Spectrum Settings

The **Spectrum Settings** panel appears at the right side of the Spectrum Analyzer figure. This panel enables you to modify settings to control the manner in which the spectrum is calculated. You can choose to hide or display the **Spectrum Settings** panel. In the Spectrum Analyzer menu, select **View** > **Spectrum Settings**. Alternatively, in the Spectrum Analyzer toolbar, select the Spectrum Settings button.





The **Spectrum Settings** panel is separated into three panes, labeled **Main Options**, **Window Options**, and **Trace Options**. You can expand each pane to see the available options.

Main Options Pane

The **Main Options** pane enables you to modify the main options.

- **Type** — The type of spectrum to display. Available options are `Power`, `Power density`, and `Spectrogram`. When you set this parameter to `Power`, the Spectrum Analyzer shows the power spectrum. When you set this parameter to `Power density`, the Spectrum Analyzer shows the power spectral density. The power spectral density is the magnitude of the spectrum normalized to a bandwidth of 1 hertz. When you set this parameter to `Spectrogram`, the Spectrum Analyzer shows the spectrogram, which displays frequency content over time. The most recent spectrogram update is at the bottom of the display and time scrolls from the bottom to the top of the display.Tunable
- **Channel** — Select the signal channel for which the spectrogram settings apply. This option displays only when the **Type** is `Spectrogram` and only if there is more than one signal channel input.
- **Sample rate (Hz)** — The sample rate, in hertz, of the input signals. Select `Inherited` to use the same sample rate as the input signal. To specify a sample rate, enter its value.

- **Full frequency span** — Enable this check box to have Spectrum Analyzer compute and plot the spectrum over the entire *Nyquist* frequency interval. By default, when the **Two-sided spectrum** check box is also enabled, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. If you clear the **Two-sided spectrum** check box, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. Tunable.
- **Span (Hz)** and **CF (Hz)**, or **FStart (Hz)** and **FStop (Hz)** — When **Span (Hz)** is showing in the **Main Options** pane, you define the range of values shown on the *frequency*-axis on the Spectrum Analyzer window using frequency span and center frequency. From the drop-down list, select **FStart (Hz)** to define the range of *frequency*-axis values using start frequency and stop frequency instead.
  - **Span (Hz)** — The frequency span, in hertz. This parameter defines the range of values shown on the *frequency*-axis on the Spectrum Analyzer window. Tunable.
  - **CF (Hz)** — The center frequency, in hertz. This parameter defines the value shown at the middle point of the *frequency*-axis on the Spectrum Analyzer window. Tunable.
  - **FStart (Hz)** — The start frequency, in hertz. This parameter defines the value shown at the leftmost side of the *frequency*-axis on the Spectrum Analyzer window. Tunable.
  - **FStop (Hz)** — The stop frequency, in hertz. The parameter defines the value shown at the rightmost side of the *frequency*-axis on the Spectrum Analyzer window. Tunable.
- **RBW (Hz) / Window length** — The frequency resolution method.

  If set to **RBW (Hz)**, the resolution bandwidth, in hertz. This property defines the smallest positive frequency that can be resolved. By default, this property is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 *RBW* intervals over the specified frequency span.

  If you set this property to a numeric value, then you must specify a value that ensures there are at least two *RBW* intervals over the specified frequency span. In other words, the ratio of the overall frequency span to *RBW* must be at least two: $\frac{span}{RBW} > 2$. Tunable.

If set to **Window length**, the length of the window, in samples, used to control the frequency resolution and compute the spectral estimates. The window length must be an integer scalar greater than 2.

. Tunable.

The time resolution value is determined based on frequency resolution method, the RBW setting, and the time resolution setting.

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
RBW (Hz)	`Auto`	`Auto`	1/RBW s
RBW (Hz)	`Auto`	Manually entered	Time Resolution s
RBW (Hz)	Manually entered	`Auto`	1/RBW s
RBW (Hz)	Manually entered	Manually entered	Must be equal to or greater than the minimum attainable time resolution, 1/RBW s. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.
Window length	—	`Auto`	1/RBW s RBW = *(NENBW**Fs)/Window Length, where *NENBW* is the normalized effective noise bandwidth of the specified window.

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
Window length	—	Manually entered	Must be equal to or greater than the minimum attainable time resolution, *(NENBW**Fs)/ Window Length. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.

- **NFFT** — The number of Fast Fourier Transform (FFT) points. You can set the **NFFT** only when in Window length mode. This property defines the length of the FFT that Spectrum Analyzer uses to compute spectral estimates. Acceptable options are `Auto` or a positive, scalar integer. The **NFFT** value must be greater than or equal to the **Window length**. By default, when **NFFT** is set to `Auto`, Spectrum Analyzer sets the number of FFT points to the window length. When in RBW mode, an FFT length is used that equals the window length required to achieve the specified RBW value.

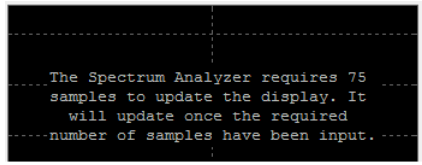
  When this property is set to a positive integer, this property is equivalent to the `n` parameter that you can set when you run the MATLAB `fft` function. Tunable.

- **Time res. (s)** — The time resolution, in seconds. Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The minimum attainable resolution is the amount of data time it takes to compute a single spectral estimate. The tooltip displays the minimum attainable resolution given the current settings. This property applies only to spectrograms. Tunable

- **Time span (s)** — The time span over which the Spectrum Analyzer displays the spectrogram, in seconds. The time span is the product of the desired number of spectral lines and the time resolution. The tooltip displays the minimum allowable time span, given the current settings. If the time span is set to `Auto`, 100 spectral lines are used. This property applies only to spectrograms.Tunable

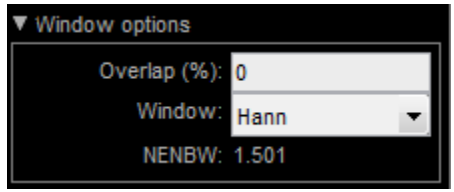
- **Samples/update** — The number of input samples required to compute one spectral update. You cannot modify this property; it is shown here for display purposes only. This property is directly related to *RBW* by the following equation:

$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$ or to the window length by this equation: $N_{samples} = \left(1 - \frac{O_p}{100}\right) x\ WindowLength$. $NENBW$ is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Window Options** pane. F_s is the sample rate. If the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify, Spectrum Analyzer produces a message on the display as shown in the following figure.



Window Options Pane

The **Window Options** pane enables you to modify the window options.

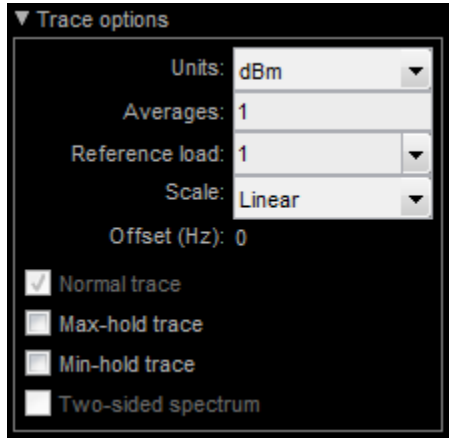


- **Overlap (%)** — The segment overlap percentage. This parameter defines the amount of overlap between the previous and current buffered data segments. The overlap creates a window segment that is used to compute a spectral estimate. The value must be greater than or equal to zero and less than 100. Tunable.
- **Window** — The windowing method to apply to the spectrum. Windowing is used to control the effect of sidelobes in spectral estimation. The window you specify affects the window length required to achieve a resolution bandwidth and the required number of samples per update. For more information about windowing, see Windows in the Spectral Analysis section of the Signal Processing Toolbox documentation. Tunable.

- **Attenuation (dB)** — The sidelobe attenuation, in decibels (dB). This property applies only when you set the **Window** parameter to `Chebyshev` or `Kaiser`. You must specify a value greater than or equal to `45`. Tunable.
- **NENBW** — Normalized Effective Noise Bandwidth of the window. You cannot modify this parameter; it is a readout shown here for display purposes only. This parameter is a measure of the noise performance of the window. It is the width of a rectangular filter that accumulates the same noise power with the same peak power gain. NENBW can be calculated from the windowing function using the following equation: $NENBW = N_{window} \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$. The rectangular window has the smallest NENBW, with a value of 1. All other windows have a larger NENBW value. For example, the Hann window has an NENBW value of approximately 1.5.

Trace Options Pane

The **Trace Options** pane enables you to modify the trace options.



- **Units** — The units of the spectrum. Available options are `dBm`, `dBW`, and `Watts`. Tunable.
- **Averages** — Specify as a positive, scalar integer the number of spectral averages. This property applies only when the Spectrum **Type** is `Power` or `Power density`.

Spectrum Analyzer computes the current power spectrum estimate by computing a running average of the last *N* power spectrum estimates. This property defines the number of spectral averages, *N*. Tunable.

- **Reference load** — The reference load, in ohms, used to scale the spectrum. Specify as a real, positive scalar the load, in ohms, that the Spectrum Analyzer uses as a reference to compute power values. Tunable.
- **Scale** — Linear or logarithmic scale. When the frequency span contains negative frequency values, Spectrum Analyzer disables the logarithmic option. Tunable.
- **Offset** — The constant frequency offset to apply to the entire spectrum or a vector of frequencies to apply to each spectrum for multiple inputs. The offset parameter is added to the values on the *frequency*-axis in the Spectrum Analyzer window. It is not used in any spectral computations. You must take the parameter into consideration when you set the **Span (Hz)** and **CF (Hz)** parameters to ensure that the frequency span is within Nyquist limits. The Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz if **Two-sided spectrum** is selected, and $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz otherwise. Tunable
- **Normal trace** — Normal trace view. This property applies only when the Spectrum **Type** is `Power` or `Power density`. By default, when this check box is enabled, Spectrum Analyzer calculates and plots the power spectrum or power spectrum density. Spectrum Analyzer performs a smoothing operation by averaging a number of spectral estimates. To clear this check box, you must first select either the **Max hold trace** or the **Min hold trace** check box. Tunable.
- **Max hold trace** — Maximum hold trace view. This property applies only when the Spectrum **Type** is `Power` or `Power density`. Select this check box to enable Spectrum Analyzer to plot the maximum spectral values of all the estimates obtained. Tunable.
- **Min hold trace** — Minimum hold trace view. This property applies only when the Spectrum **Type** is `Power` or `Power density`. Select this check box to enable Spectrum Analyzer to plot the minimum spectral values of all the estimates obtained.

  of all the estimates obtained. Tunable.
- **Two-sided spectrum** — Select this check box to enable two-sided spectrum view. In this view, both negative and positive frequencies are shown. If you clear this check

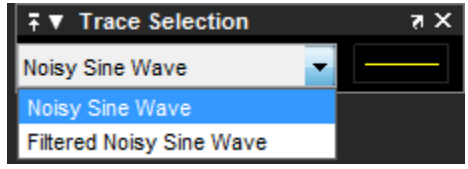
box, Spectrum Analyzer shows a one-sided spectrum with only positive frequencies. Spectrum Analyzer requires that this parameter is selected when the input signal is complex-valued.

Measurements Panels

The Measurements panels are the other four panels that appear to the right side of the Spectrum Analyzer figure.

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 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**.

Cursor Measurements Panel

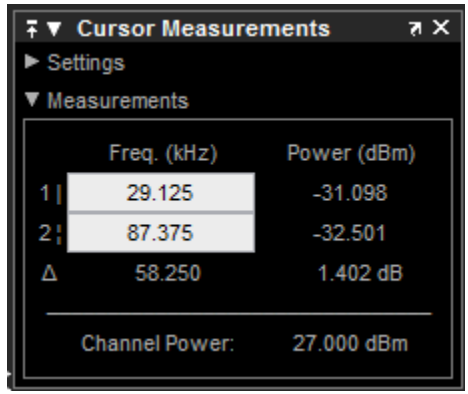
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note: If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

In the Scope menu, select **Tools > Measurements > Cursor Measurements**.

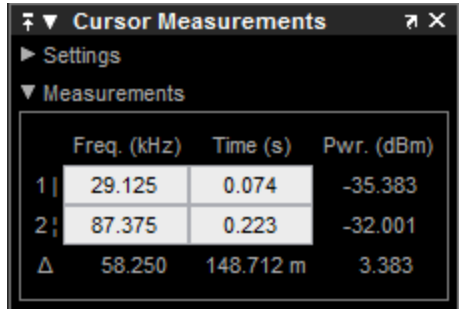
Alternatively, in the Scope toolbar, click the Cursor Measurements button.

The **Cursor Measurements** panel appears as follows for power and power density spectra.



The **Cursor Measurements** panel appears as follows for spectrograms.

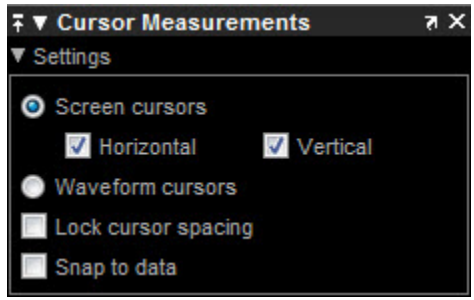
Note: You must pause the spectrogram display before you can use cursors.



The **Cursor Measurements** panel is separated into two panes, labeled **Settings** and **Measurements**. You can expand each pane to see the available options.

You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

The **Settings** pane enables you to modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors (for power and power density spectra only).
- **Horizontal** — Shows horizontal screen cursors (for power and power density spectra only).
- **Vertical** — Shows vertical screen cursors (for power and power density spectra only).
- **Waveform Cursors** — Shows cursors that attach to the input signals (for power and power density spectra only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

Measurements Pane

The **Measurements** pane displays the frequency (Hz), time (s), and power (dBm) value measurements. Time is displayed only in spectrogram mode. **Channel Power** shows the total power between the cursors.

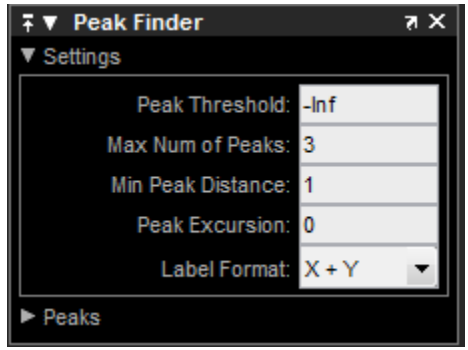
- **1 |** — Shows or enables you to modify the frequency or time (for spectrograms only), or both, at cursor number one.
- **2 :** — Shows or enables you to modify the frequency or time (for spectrograms only), or both, at cursor number two.
- **Δ** — Shows the absolute value of the difference in the frequency, time (for spectrograms only), and power between cursor number one and cursor number two.
- **Channel Power** — Shows the total power in the channel defined by the cursors.

The letter after the value associated with a measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix.

Peak Finder Panel

Note: The **Peak Finder** panel requires a DSP System Toolbox or Simscape™ license.

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered to be peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. 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.



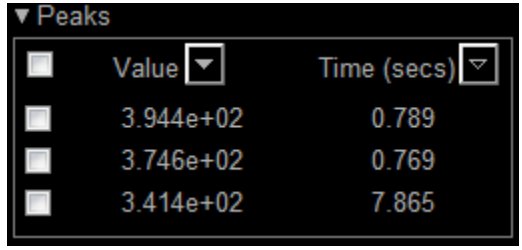
The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

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 from 1 through 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 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, 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.

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.



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 in the direction of the button arrow. If the sort button is not filled (), then the peak values are sorted in the opposite direction of the button arrow. The **Max Num of Peaks** parameter still controls the number of peaks listed.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. 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.

The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for 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.

Channel Measurements Panel

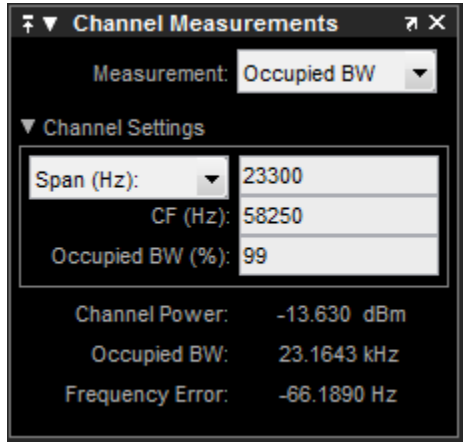
The **Channel Measurements** panel displays occupied bandwidth or adjacent channel power ratio (ACPR) measurements. You can choose to hide or display this pane in the Scope menu by selecting **Tools > Measurements > Channel Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements  button.

In addition to the measurements, the **Channel Measurements** panel has an expandable **Channel Settings** pane.

- **Measurement** — The type of measurement data to display. Available options are `Occupied BW` or `ACPR`. See “Algorithms” on page 1-1843 for information on how Occupied BW is calculated. ACPR is the adjacent channel power ratio, which is the ratio of the main channel power to the adjacent channel power.

When you select `Occupied BW` as the **Measurement**, the following fields appear.

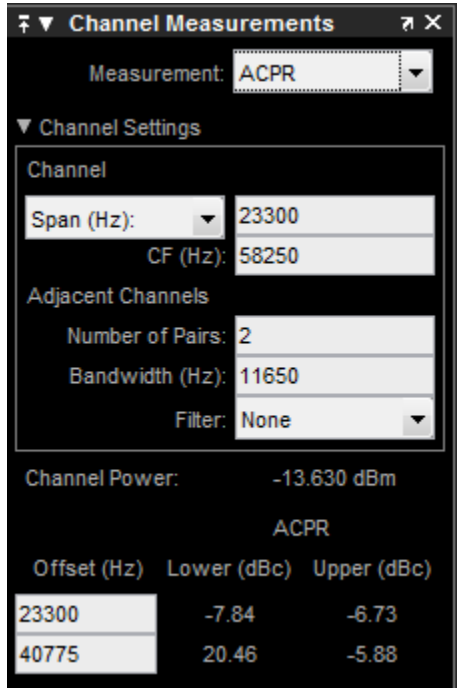


- **Channel Settings** — Enables you to modify the parameters for calculating the channel measurements.

  ### Channel Settings for Occupied BW

  - Select the frequency span of the channel, `Span(Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, `FStart(Hz)`, and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.
  - **CF (Hz)** — The center frequency of the channel.
  - **Occupied BW (%)** — The percentage of the total integrated power of the spectrum centered on the selected channel frequency over which to compute the occupied bandwidth.
- **Channel Power** — The total power in the channel.
- **Occupied BW** — The bandwidth containing the specified **Occupied BW (%)** of the total power of the spectrum. This setting is available only if you select `Occupied BW` as the **Measurement** type.
- **Frequency Error** — The difference between the center of the occupied band and the center frequency (**CF**) of the channel. This setting is available only if you select `Occupied BW` as the **Measurement** type.

When you select `ACPR` as the **Measurement**, the following fields appear.



- **Channel Settings** — Enables you to modify the parameters for calculating the channel measurements.

  ### Channel Settings for ACPR

  - Select the frequency span of the channel, `Span (Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, `FStart(Hz)`, and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.
  - **CF (Hz)** — The center frequency of the channel.
  - **Number of Pairs** — The number of pairs of adjacent channels.
  - **Bandwidth (Hz)** — The bandwidth of the adjacent channels.
  - **Filter** — The filter to use for both main and adjacent channels. Available filters are `None`, `Gaussian`, and `RRC` (root-raised cosine).
- **Channel Power** — The total power in the channel.

- **Offset (Hz)** — The center frequency of the adjacent channel with respect to the center frequency of the main channel. This setting is available only if you select ACPR as the **Measurement** type.
- **Lower (dBc)** — The power ratio of the lower sideband to the main channel. This setting is available only if you select ACPR as the **Measurement** type.
- **Upper (dBc)** — The power ratio of the upper sideband to the main channel. This setting is available only if you select ACPR as the **Measurement** type.

Distortion Measurements Panel

The **Distortion Measurements** panel displays harmonic distortion and intermodulation distortion measurements. You can choose to hide or display this panel in the Scope menu by selecting **Tools > Measurements > Distortion Measurements**. Alternatively, in the Scope toolbar, click the Distortion Measurements  button.

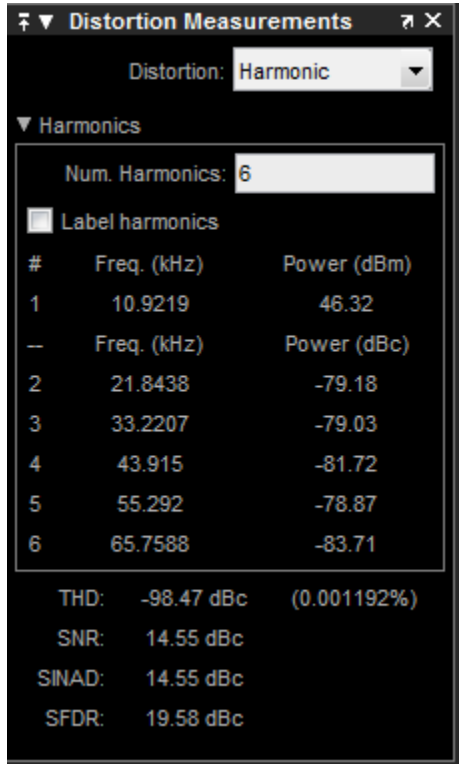
The **Distortion Measurements** panel has an expandable **Harmonics** pane, which shows measurement results for the specified number of harmonics.

Note: For an accurate measurement, ensure that the fundamental signal (for harmonics) or primary tones (for intermodulation) is larger than any spurious or harmonic content. To do so, you may need to adjust the resolution bandwidth (RBW) of the spectrum analyzer. Make sure that the bandwidth is low enough to isolate the signal and harmonics from spurious and noise content. In general, you should set the RBW so that there is at least a 10dB separation between the peaks of the sinusoids and the noise floor. You may also need to select a different spectral window to obtain a valid measurement.

- **Distortion** — The type of distortion measurements to display. Available options are `Harmonic` or `Intermodulation`. Select `Harmonic` if your system input is a single sinusoid. Select `Intermodulation` if your system input is two equal amplitude sinusoids. Intermodulation can help you determine distortion when only a small portion of the available bandwidth will be used.

  See “Algorithms” on page 1-1843 for information on how distortion measurements are calculated.

When you select `Harmonic` as the **Distortion**, the following fields appear.

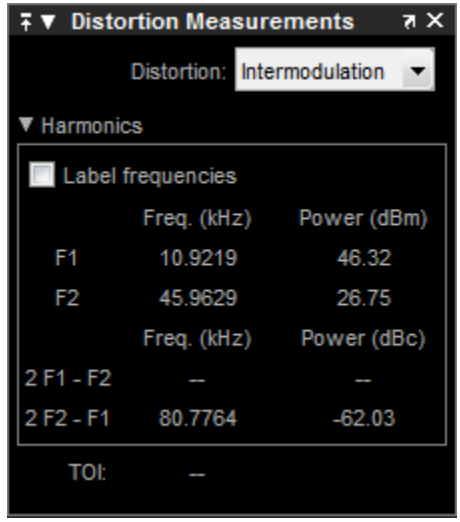


The harmonic distortion measurement automatically locates the largest sinusoidal component (fundamental signal frequency). It then computes the harmonic frequencies and power in each harmonic in your signal. Any DC component is ignored. Any harmonics that are outside the spectrum analyzer's frequency span are not included in the measurements. Adjust your frequency span so that it includes all the desired harmonics.

Note: To best view the harmonics, make sure that your fundamental frequency is set high enough to resolve the harmonics. However, this frequency should not be so high that aliasing occurs. For the best display of harmonic distortion, your plot should not show skirts, which indicate frequency leakage. Additionally, the noise floor should be visible. Using a Kaiser window with a large sidelobe attenuation may help to reduce the skirts.

- **Num. Harmonics** — Number of harmonics to display, including the fundamental frequency. Valid values of **Num. Harmonics** are from 2 to 10. The default value is 6.
- **Label Harmonics** — Select **Label Harmonics** to add numerical labels to each harmonic in the spectrum display.
- **1** — The fundamental frequency, in hertz, and its power, in decibels of the measured power referenced to one milliwatt (dBm).
- **2, 3, ...** — The harmonics frequencies, in hertz, and their power in decibels relative to the carrier (dBc). If the harmonics are at the same level or exceed the fundamental frequency, reduce the input power.
- **THD** — The total harmonic distortion. This value represents the ratio of the power in the harmonics, *D*, to the power in the fundamental frequency, *S*. If the noise power is too high in relation to the harmonics, the THD value is not accurate. In this case, lower the resolution bandwidth or select a different spectral window. $THD = 10\log_{10}(D/S)$.
- **SNR** — Signal-to-noise ratio (SNR). This value represents the ratio of power in the fundamental frequency, *S*, to the power of all nonharmonic content, *N*, including spurious signals, in decibels relative to the carrier (dBc). $SNR = 10\log_{10}(S/N)$. If you see – – as the reported SNR, your signal's total non-harmonic content is less than 30% of the total signal.
- **SINAD** — Signal-to-noise-and-distortion. This value represents the ratio of the power in the fundamental frequency, *S* to all other content (including noise, *N*, and harmonic distortion, *D*), in decibels relative to the carrier (dBc). $SINAD = 10\log_{10}(S/(N+D))$.
- **SFDR** — Spurious free dynamic range (SFDR). This value represents the ratio of the power in the fundamental frequency, *S*, to power of the largest spurious signal, *R*, regardless of where it falls in the frequency spectrum. The worst spurious signal may or may not be a harmonic of the original signal. SFDR represents the smallest value of a signal that can be distinguished from a large interfering signal. SFDR includes harmonics. $SFDR = 10\log_{10}(S/R)$.

When you select `Intermodulation` as the **Distortion**, the following fields appear.



The intermodulation distortion measurement automatically locates the fundamental, first-order frequencies (F1 and F2). It then computes the frequencies of the third-order intermodulation products (2*F1-F2 and 2*F2-F1).

- **Label frequencies** — Select **Label frequencies** to add numerical labels to the first-order intermodulation product and third-order frequencies in the spectrum analyzer display.
- **F1** — Lower fundamental first-order frequency
- **F2** — Upper fundamental first-order frequency
- **2F1 - F2** — Lower intermodulation product from third-order harmonics
- **2F2 - F1** — Upper intermodulation product from third-order harmonics
- **TOI** — Third-order intercept point. If the noise power is too high in relation to the harmonics, the TOI value will not be accurate. In this case, you should lower the resolution bandwidth or select a different spectral window. If the TOI has the same amplitude as the input two-tone signal, reduce the power of that input signal.

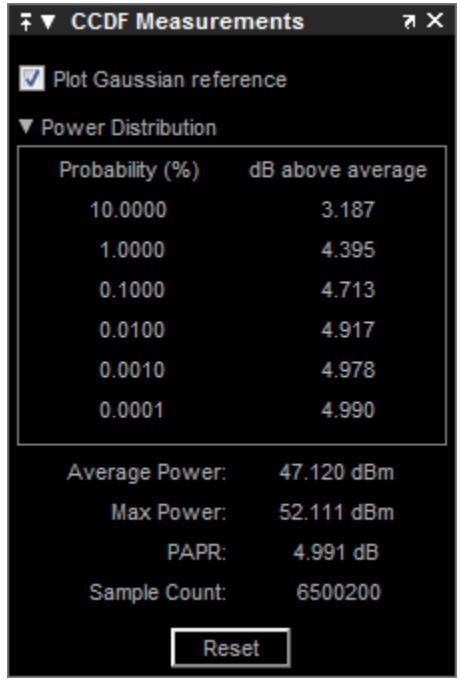
CCDF Measurements Panel

The **CCDF Measurements** panel displays complimentary cumulative distribution function measurements. CCDF measurements in this scope show the probability of a

signal's instantaneous power being a specified level above the signal's average power. These measurements are useful indicators of a signal's dynamic range.

To compute the CCDF measurements, each input sample is quantized to 0.01 dB increments. Using a histogram 100 dB wide (10,000 points at 0.01 dB increments), the largest peak encountered is placed in the last bin of the histogram. If a new peak is encountered, the histogram shifts to make room for that new peak.

You can choose to hide or display this panel in the Scope menu by selecting **Tools > Measurements > CCDF Measurements**. Alternatively, in the Scope toolbar, click the CCDF Measurements button.



- **Plot Gaussian reference** — Select **Plot Gaussian reference** to show the Gaussian white noise reference signal on the plot.
- **Probability (%)** — The percentage of the signal that contains the power level above the value listed in the **dB above average** column
- **dB above average** — The expected minimum power level at the associated **Probability (%)**.

- **Average Power** — The average power level of the signal since the start of simulation or from the last reset.

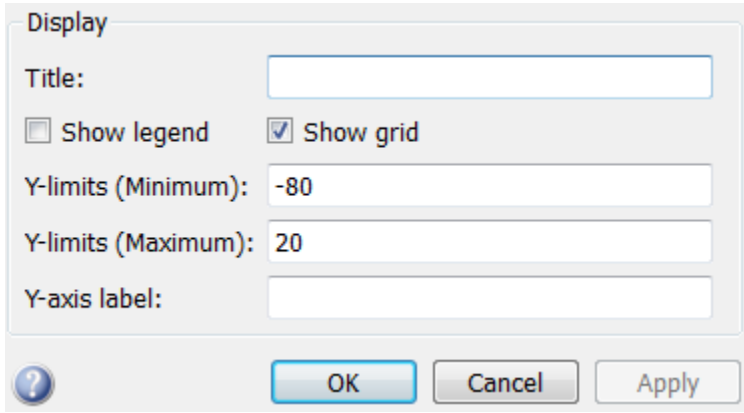
  Max Power — The maximum power level of the signal since the start of simulation or from the last reset.
- **PAPR** — The ratio of the peak power to the average power of the signal. PAPR should be less that 100 dB to obtain accurate CCDF measurements. If PAPR is above 100 dB, only the highest 100 dB power levels are plotted in the display and shown in the distribution table.
- **Sample Count** — The total number of samples used to compute the CCDF.
- **Reset** — Clear all current CCDF measurements and restart.

Visuals — Spectrum Properties

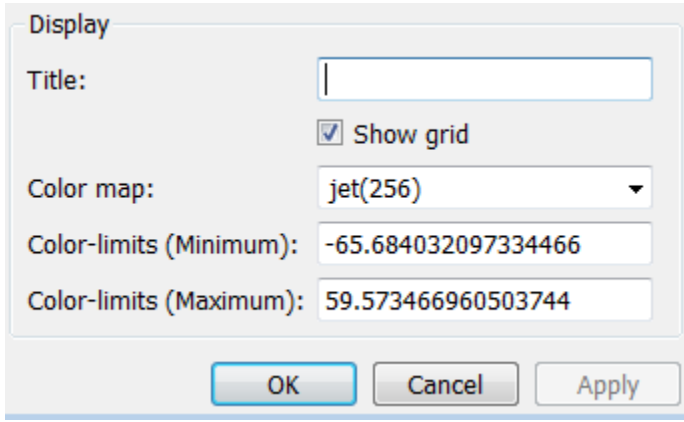
The Visuals—Spectrum Properties dialog box controls the visual configuration settings of the Spectrum Analyzer display. From the Spectrum Analyzer menu, select **View** > **Configuration Properties** to open this dialog box. Alternatively, in the Spectrum Analyzer toolbar, click the Configuration Properties button.

Display Pane

When the Spectrum **Type** is `Power` or `Power density`, the **Display** pane of the Visuals —Spectrum Properties dialog box appears as follows:



When the Spectrum **Type** is `Spectrogram` the **Display** pane of the Visuals—Spectrum Properties dialog box appears as follows:



Title

Specify the display title as a string. Enter `%<SignalLabel>` to use the signal labels in the Simulink Model as the axes titles. This property is Tunable.

By default, the display has no title.

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 off the legend, clear the **Show legend** check box. This parameter applies only when the Spectrum **Type** is `Power` or `Power density`. 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

Y-limits (Minimum)

Specify the minimum value of the *y*-axis. Tunable

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Regardless of this property, Spectrum Analyzer always displays power units after this text as one of `'dBm'`, `'dBW'`, `'Watts'`, `'dBm/Hz'`, `'dBW/Hz'`, or `'Watts/Hz'`. Tunable.

Color map

Select the color map for the spectrogram, or enter a 3-column matrix expression for the color map. See `colormap` for information. Tunable.

Color-limits (Minimum)

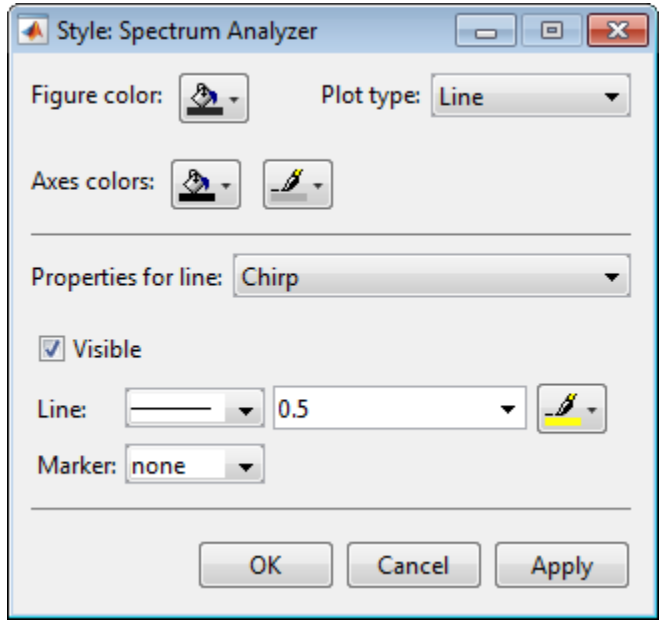
Set the signal power for the minimum color value of the spectrogram. Tunable.

Color-limits (Maximum)

Set the signal power for the maximum color value of the spectrogram. Tunable.

Style Dialog Box

In the **Style** dialog box, you can customize the style of power and power density displays. This dialog box is not available in spectrogram view. You are able to change the color of the figure, the background and foreground colors of the axes, and properties of the lines. From the Spectrum Analyzer menu, select **View > Style** to open this dialog box.



Properties

The **Style** dialog box allows you to modify the following properties of the Spectrum Analyzer 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 whether to display a `Line` or `Stem` plot.

Axes colors

Specify the color that you want to apply to the background of the axes.

Properties for line

Specify the channel for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected channel 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 channel.

Marker

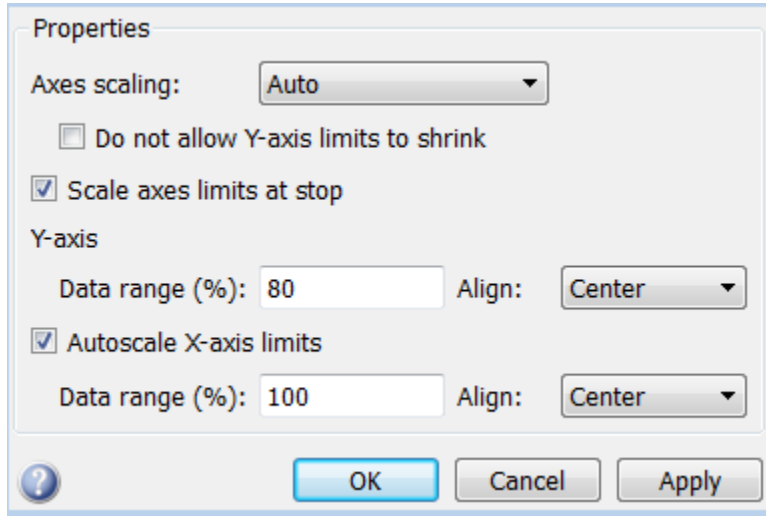
Specify marks for the selected channel to show at its data points. This parameter is similar to the `Marker` property for the MATLAB Handle Graphics® plot objects. You can choose any of the marker symbols from the dropdown.

Tools — Axes Scaling Properties

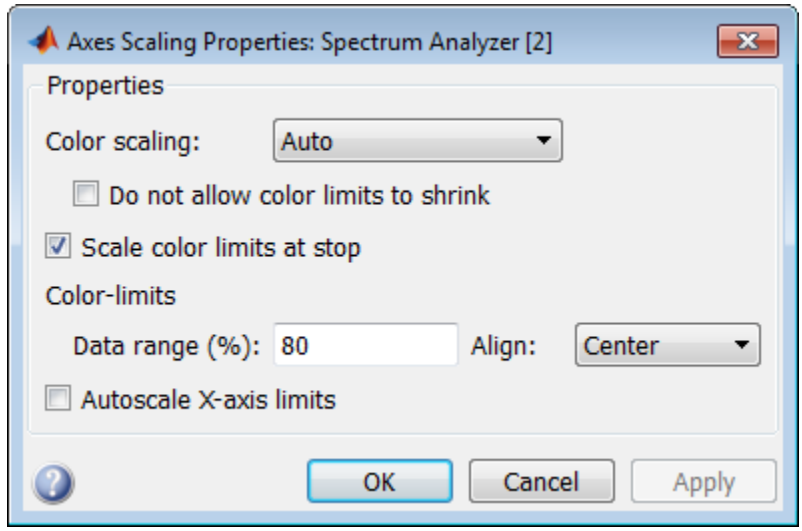
The Tools — Axes Scaling Properties dialog box allows you to automatically zoom in on and zoom out of your data. You can also scale the axes and color of the Spectrum Analyzer. In the Spectrum Analyzer menu, select **Tools > Scaling Properties** to open this dialog box.

Properties

The Tools—Axes Scaling Properties dialog box appears as follows for power and power density views.



For spectrogram view, the Tools—Axes Scaling Properties dialog box appears as follows.



Axes scaling/Color scaling

Specify when the scope automatically scales the axes. If the spectrogram is displayed, specify when the scope automatically scales the color. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes or color. You can manually scale the axes or color in any of the following ways:
  - Select **Tools > Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes or color as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** or **Do not allow color limits to shrink** .
- `After N Updates` — Selecting this option causes the scope to scale the axes or color after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this parameter is set to `Auto`, and the scope does not shrink the *y*-axis limits when scaling the axes or color. Tunable.

Do not allow Y-axis limits to shrink / Do not allow color limits to shrink

When you select this property, the *y*-axis are allowed to grow during axes scaling operations. If the spectrogram is displayed, selecting this property allows the color limits to grow during axis scaling. If you clear this check box, the *y*-axis or color limits can shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** or **Color scaling** property. When you set the **Axes scaling** or **Color scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits can shrink. Tunable.

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. If the spectrogram is displayed, this property specifies the number of updates after which to scale the color. This property appears only when you select `After N Updates` for the **Axes scaling** or **Color scaling** property. Tunable.

Scale axes limits at stop/Scale color limits at stop

Select this check box to scale the axes when the simulation stops. If the spectrogram is displayed, select this check box to scale the color when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%) / Color-limits Data range

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. If the spectrogram is displayed, set the percentage of the power values range within the colormap. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range or color such that your data uses only 30% of the *y*-axis range or color. Tunable.

Y-axis Align / Color-limits Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. If the spectrogram is displayed, specify where the scope aligns your data along the *y*-axis when it scales the color. You can select `Top`, `Center`, or `Bottom`. Tunable.

Autoscale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable.

X-axis Data range (%)

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this property 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** property to specify data placement along the *x*-axis.

This property appears only when you select the **Scale X-axis limits** check box. Tunable.

X-axis Align

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable.

Algorithms

Spectrum Analyzer uses the `RBW` or the `Window Length` setting in the **Spectrum Settings** panel to determine the data window length. The value of the `FrequencyResolutionMethod` property determines whether RBW or window length is used. Then, it partitions the input signal into a number of windowed data segments. Finally, Spectrum Analyzer uses the modified periodogram method to compute spectral updates, averaging the windowed periodograms for each segment.

1. Spectrum Analyzer requires that a minimum number of samples have been provided before it computes a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to resolution bandwidth, *RBW*, by the following equation or to the window length, by the equation shown in step 1b.

$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

   The normalized effective noise bandwidth, *NENBW*, is a factor that depends on the windowing method. Spectrum Analyzer shows NENBW in the **Window Options** pane of the **Spectrum Settings** panel. Overlap percentage, O_p, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** panel. F_s is the sample rate of the input signal. Spectrum Analyzer shows sample rate in the **Main Options** pane of the **Spectrum Settings** panel.

   a. When in RBW mode, the window length required to compute one spectral update, N_{window}, is directly related to the resolution bandwidth and normalized effective noise bandwidth by the following equation.

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

      When in WindowLength mode, the window length is used as specified.

   b. The number of input samples required to compute one spectral update, $N_{samples}$, is directly related to the window length and the amount of overlap by the following equation.

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$$

When you increase the overlap percentage, fewer new input samples are needed to compute a new spectral update. For example, if the window length is 100, then the number of input samples required to compute one spectral update is given as shown in the following table.

O_p	$N_{samples}$
0%	100
50%	50
80%	20

c The normalized effective noise bandwidth, *NENBW*, is a window parameter determined by the window length, N_{window}, and the type of window used. If $w(n)$ denotes the vector of N_{window} window coefficients, then *NENBW* is given by the following equation.

$$NENBW = N_{window} \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$$

d When in RBW mode, you can set the resolution bandwidth using the value of the **RBW** parameter on the **Main options** pane of the **Spectrum Settings** panel. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two, as given in the following equation.

$$\frac{span}{RBW} > 2$$

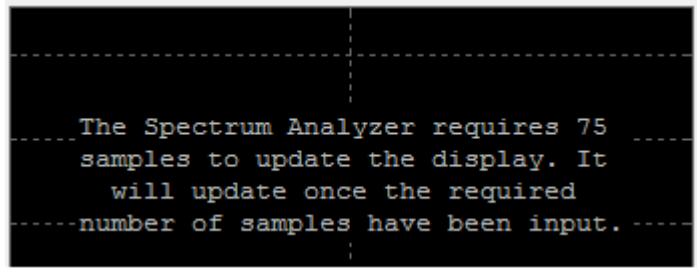
By default, the **RBW** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency

span. Thus, when you set **RBW** to `Auto`, it is calculated by the following equation. $RBW_{auto} = \frac{span}{1024}$

e When in window length mode, you specify N_{window} and the resulting RBW is

$$\frac{NENBW * Fs}{N_{window}}$$

In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer produces a message on the display, as shown in the following figure.



Spectrum Analyzer removes this message and displays a spectral estimate as soon as enough data has been input. Notice that this behavior differs from the Spectrum Scope block in versions R2012b and earlier. If the **Buffer input** check box was selected, the Spectrum Scope block computed a spectral update using the number of samples given by the **Buffer size** parameter. Otherwise, the Spectrum Scope block computed a spectral update using the number of samples in each frame.

2 Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, or spectrogram computed by the modified *Periodogram* estimator. For more information about the Periodogram method, see `periodogram` in the Signal Processing Toolbox documentation.

Power Spectral Density — The power spectral density (PSD) is given by the following equation.

$$PSD(f) = \frac{1}{P}\sum_{p=1}^{P}\frac{\left|\sum_{n=1}^{N_{FFT}} x^{(p)}[n]e^{-j2\pi f(n-1)T}\right|^2}{F_s \times \sum_{n=1}^{N_{window}} w^2[n]}$$

In this equation, $x[n]$ is the discrete input signal. On every input signal frame, Spectrum Analyzer generates as many overlapping windows as possible, each window denoted as $x^{(p)}[n]$, and computes their periodograms. Spectrum Analyzer displays a running average of the P most current periodograms.

Power Spectrum — The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by the following equation.

$$P_{spectrum}(f) = PSD(f) \times RBW = PSD(f) \times \frac{F_s \times NENBW}{N_{window}} = \frac{1}{P}\sum_{p=1}^{P}\frac{\left|\sum_{n=1}^{N_{FFT}} x^{(p)}[n]e^{-j2\pi f(n-1)T}\right|^2}{\left[\sum_{n=1}^{N_{window}} w[n]\right]^2}$$

Spectrogram — Each line of the spectrogram is one periodogram. The time resolution of each line is 1/RBW, which is the minimum attainable resolution. Achieving the resolution you want may require combining several periodograms may be combined. You then useinterpolation to calculate noninteger values of 1/RBW. In the spectrogram display, time scrolls from bottom to top, so the most recent data is shown at the bottom of the display. The offset shows the time value at which the center of the most current spectrogram line occurred.

Note: The number of FFT points (N_{fft}) is independent of the window length (N_{window}). You can set them to different values provided that N_{fft} is greater than or equal to N_{window}.

The *Occupied BW* is calculated as follows.

- Calculate the total power in the measured frequency range.

- Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, the power distributed in each frequency is summed until this sum is $\frac{100 - Occupied\ BW\ \%}{2}$ of the total power.
- Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, the power distributed in each frequency is summed until it reaches $\frac{100 - Occupied\ BW\ \%}{2}$ of the total power.
- The bandwidth between the lower and upper power frequency values is the occupied bandwidth.
- The frequency halfway between the lower and upper frequency values is the center frequency.

The *Distortion Measurements* are computed as follows.

1. Spectral content is estimated by finding peaks in the spectrum. When the algorithm detects a peak, it ignores all adjacent content that decreases monotonically from the peak. After recording the width of the peak, it clears all monotonically decreasing values (that is, it treats all of these values as if they belong to the peak). Using this method, all spectral content centered at DC (0 Hz) is removed from the spectrum and the amount of bandwidth cleared (W_0) is recorded.
2. The fundamental power (P_1) is determined from the remaining maximum value of the displayed spectrum. A local estimate (Fe_1) of the fundamental frequency is made by computing the central moment of the power in the vicinity of the peak. The bandwidth of the fundamental power content (W_1) is recorded. Then, the power associated from the fundamental is removed as in step 1.
3. The power and width of the second, and higher order harmonics (P_2, W_2, P_3, W_3, etc.) are determined in succession by examining the frequencies closest to the appropriate multiple of the local estimate (Fe_1). Any spectral content that decreases in a monotonically about the harmonic frequency is removed from the spectrum first before proceeding to the next harmonic.
4. Once the DC, fundamental, and harmonic content is removed from the spectrum, the power of the remaining spectrum is examined for its sum ($P_{remaining}$) peak value ($P_{maxspur}$), and its median value ($P_{estnoise}$).
5. The sum of all the removed bandwidth is computed as $W_{sum} = W_0+W_1+W_2+...+W_n$.

The sum of powers of the second and higher order harmonics are computed as $P_{harmonic} = P_2+P_3+P_4+...+P_n$.

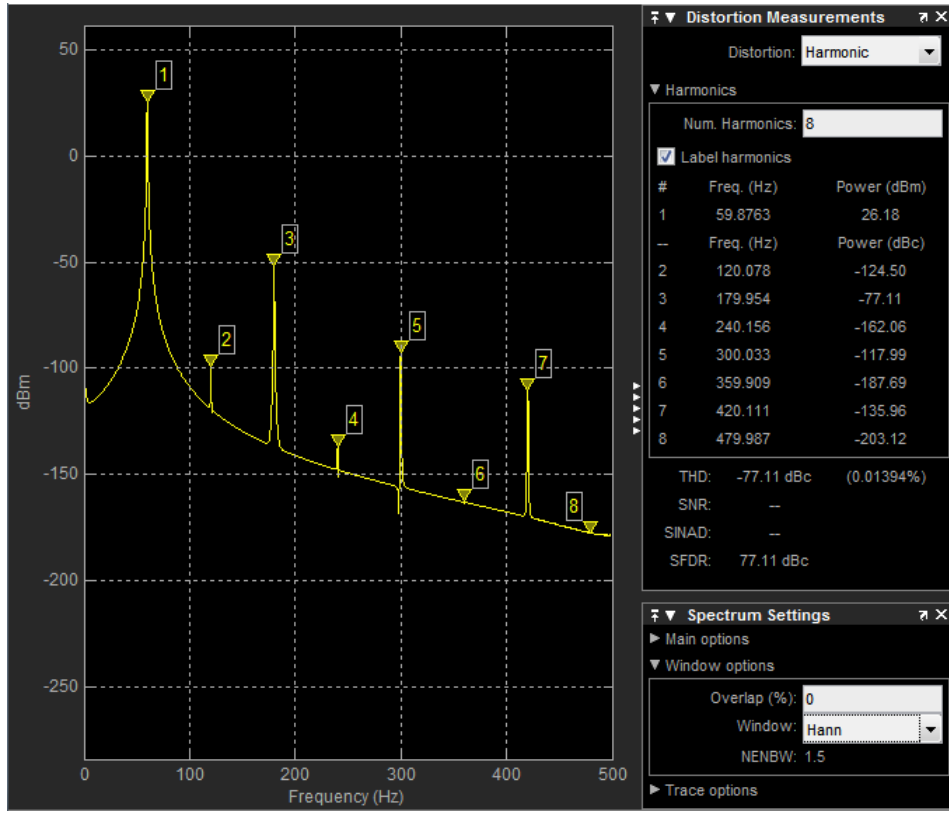
6 The sum of the noise power is then estimated as $P_{noise} = (P_{remaining}*dF + P_{estnoise}*W_{sum})/RBW$, where dF is the absolute difference between frequency bins, and RBW is the resolution bandwidth of the window.

7 The metrics for SNR, THD, SINAD, and SFDR are then computed from the estimates.

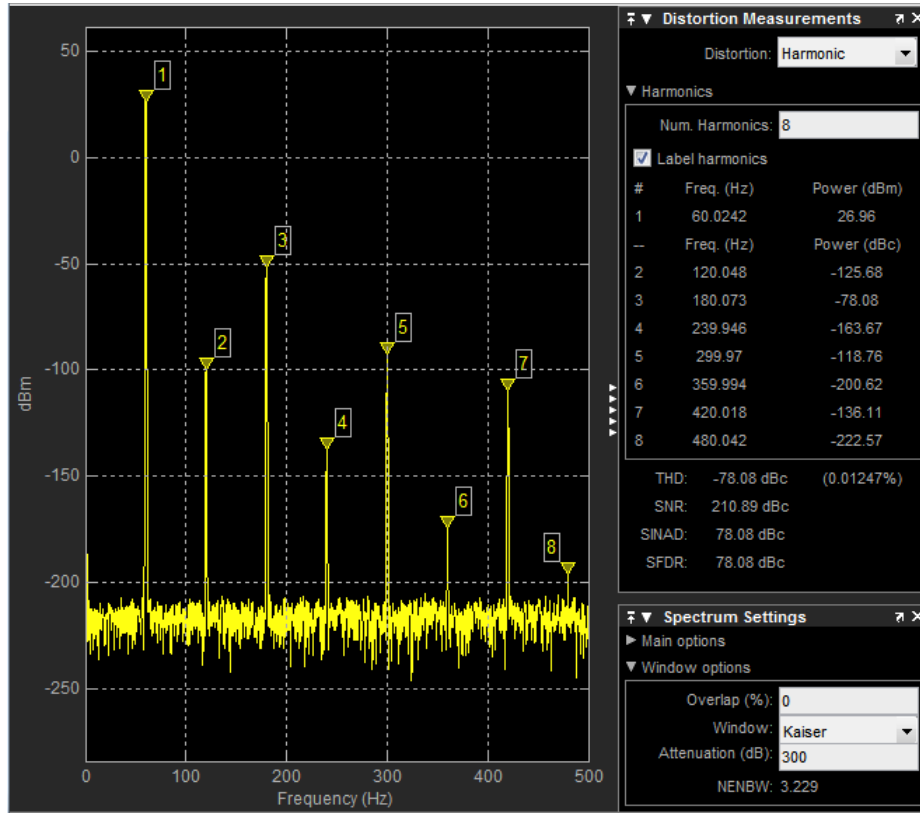
- THD = $10*\log10(P_{harmonic}/P_1)$
- SINAD = $10*\log_{10}(P_1/(P_{harmonic} + P_{noise})$
- SNR = $10*\log_{10}(P_1/P_{noise})$
- SFDR = $10*\log_{10}(P_1/\max(P_{maxspur}, \max(P_2,P_3,...,P_n))$

The following considerations apply to *Distortion Measurements*.

- The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default `Hann` window setting of the Spectrum Analyzer may exhibit leakage that can completely mask the noise floor of the measured signal.



The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (–) reported for **SNR** and **SINAD**. Consider using a Kaiser window with a high attenuation (up to 330dB) to minimize spectral leakage if your application can tolerate the increased equivalent noise bandwidth (ENBW) of the Kaiser window.



- The DC component is ignored.
- After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.
- *N*th order intermodulation products occur at

  $A*F1 + B*F2$

  where $F1$ and $F2$ are the sinusoid input frequencies and $|A| + |B| = N$. A and B are integer values.

- For intermodulation measurements, the third-order intercept (TOI) point is computed as follows, where P is power in decibels of the measured power referenced to one milliwatt (dBm).:
  - $TOI_{lower} = P_{F1} + (P_{F2} - P_{(2F1-F2)})/2$
  - $TOI_{upper} = P_{F2} + (P_{F1} - P_{(2F2-F1)})/2$
  - $TOI = + (TOI_{lower} + TOI_{upper})/2$

Differences from Spectrum Scope Block

All Simulink models containing Spectrum Scope blocks load with Spectrum Analyzer blocks in R2013a or later. Several options that were available on the Parameters dialog box of the Spectrum Scope block are no longer available or have changed. The parameters of Spectrum Scope map to Spectrum Analyzer parameters in the following manner.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Scope Properties	**Buffer input** check box	R2013a Spectrum Analyzer does not require that input signals are buffered. Spectrum Analyzer determines the number of samples needed using the value of the **RBW** parameter. Regardless of whether the input is a frame-based or sample-based signal, Spectrum Analyzer calculates the spectrum once it has acquired the requisite number of samples.	For Spectrum Scope blocks in R2012b or earlier models, the equivalent R2013a Spectrum Analyzer **RBW** value is given by the equation: $$RBW = \frac{NENBW \times F_s}{N_{window}}$$ In the preceding equation, $NENBW$ is the window constant calculated for a window length of 1000, F_s is the sample rate of the block,

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
			and N_{window} is the buffer length. If the input signal to the R2012b Spectrum Scope block was frame-based and the **Buffer input** check box was cleared, then the R2013a Spectrum Analyzer computes the **RBW** value with N_{window} set to the frame size of the input signal.
Scope Properties	**Buffer size** parameter	R2013a Spectrum Analyzer uses the **RBW** parameter to determine the requisite number of samples to calculate the spectrum, instead of using the buffer size or frame length.	For Spectrum Scope blocks in R2012b or earlier models, if the input signal was frame-based and the **Buffer input** check box was selected, then the R2013a Spectrum Analyzer computes the **RBW** value with N_{window} set to the value of the **Buffer size** parameter.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Scope Properties	**Buffer Overlap** parameter	R2013a Spectrum Analyzer has an **Overlap %** parameter that is directly related to buffer overlap.	R2013a Spectrum Analyzer will compute its **Overlap %** using the equation: $$O_p = {O_l}/{N_{window}} \times 100$$ In the preceding equation, O_p is **Overlap %** parameter value, O_l is the R2012b Spectrum Scope **Buffer overlap** parameter value, and N_{window} is the buffer length.
Scope Properties	**Treat Mx1 and unoriented sample-based signals as**	R2013a Spectrum Analyzer defaults to treating Mx1 and unoriented sample-based signals as one channel.	Spectrum Scope blocks in R2012b or earlier models with **Treat Mx1 and unoriented sample-based signals as** set to `M Channels` will have the Spectrum Analyzer property `TreatMby1SignalAsOneChannel` set to `false`. This property is available only via the Scope Configuration object.
Scope Properties	**Window** parameter	R2013a Spectrum Analyzer does not have the `Bartlett`, `Blackman`, `Triang`, or `Hanning` settings.	Spectrum Scope blocks in R2012b or earlier models with a window parameter set to any of these values will have their **Window** parameter set to `Hann` in the R2013a Spectrum Analyzer.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Scope Properties	**Window Sampling** parameter	R2013a Spectrum Analyzer does not have a `Periodic` option. All window sampling is now symmetric in the R2013a Spectrum Analyzer.	n/a
Display Properties	**Persistence** check box — this setting would execute the equivalent of the MATLAB `hold on` command, adding another line for each spectrum computation on the display.	This option is not available in the R2013a Spectrum Analyzer, which has replaced this feature with the trace options, **Normal Trace**, **Max Hold Trace**, and **Min Hold Trace**.	Spectrum Scope blocks in R2012b or earlier models with persistence enabled will have their **Max Hold Trace** check box selected in the R2013a Spectrum Analyzer.
Display Properties	**Compact Display** check box	There is no equivalent capability in the R2013a Spectrum Analyzer.	n/a
Axis Properties	**Inherit Sample time from input** check box	R2013a Spectrum Analyzer always uses the sample time of the input signal.	n/a

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Axis Properties	**Frequency display limits** parameter	R2013a Spectrum Analyzer determines the range of frequencies calculated based on the **Full Span**, **FStart (Hz)**, and **FStop (Hz)** parameters.	If this parameter was set to: • `Auto` — R2013a Spectrum Analyzer selects the **Full Span** check box on the Spectrum Settings panel, Main options pane. • `User-defined` — R2013a Spectrum Analyzer clears the **Full Span** check box on the Spectrum Settings panel Main options pane.
Axis Properties	**Minimum frequency (Hz)** parameter	R2013a Spectrum Analyzer determines the range of frequencies calculated based on the **Full Span**, **FStart (Hz)**, and **FStop (Hz)** parameters.	If the `User-defined` parameter was chosen, then this parameter maps to the R2013a Spectrum Analyzer **FStart (Hz)** parameter.
Axis Properties	**Maximum frequency (Hz)** parameter	R2013a Spectrum Analyzer determines the range of frequencies calculated based on the **Full Span**, **FStart (Hz)**, and **FStop (Hz)** parameters.	If the `User-defined` parameter was chosen, then this parameter maps to the R2013a Spectrum Analyzer **FStop (Hz)** parameter.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Line Properties	**Line visibilities**, **Line styles**, **Line markers**, and **Line colors** parameters	There are no equivalent capabilities in the R2013a Spectrum Analyzer.	Once the simulation has started, you can modify the line styles, markers, and colors using the Style dialog box.

The R2012b Spectrum Scope allowed you to retain the axes limits over multiple simulations by selecting **Axes > Save Axes Settings**. There is no equivalent capability in the R2013a Spectrum Analyzer. However, you can automatically scale the axes to a specified range using the Tools — Axes Scaling Properties dialog box.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned)

Supported Simulation Modes

You can use the scope block in models running the following supported simulation modes.

Mode	Supported	Notes and Limitations
Normal	Yes	

Mode	Supported	Notes and Limitations
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 code generation and the build process. For more information about External mode, see "Set Up and Use Host/Target Communication Channel" in the Simulink Coder documentation. The scope does not support data archiving. See "Set External Mode Data Archiving Parameters" in the Simulink Desktop Real-Time™ documentation.

For more information about these modes, see "How Acceleration Modes Work" in the Simulink documentation.

See Also

dsp.SpectrumAnalyzer | `Array Plot` | `sptool` | `Time Scope`

Related Examples

- "Display Frequency-Domain Data in Spectrum Analyzer"

Introduced in R2014b

Slider Switch

Set on/off values to tune parameters or variables

Library

Dashboard

Description

The `Slider Switch` block enables you to control tunable parameters and variables in your model during simulation. The block has two states that can be set to two different values.

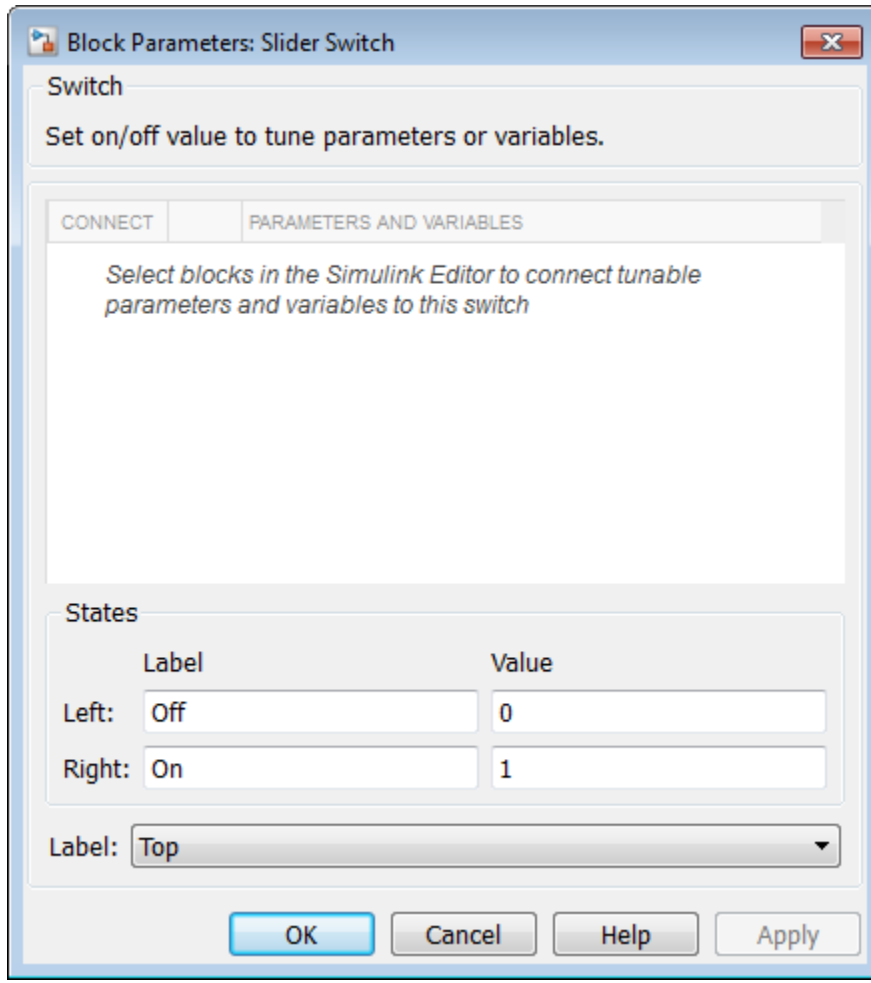
To control a tunable parameter or variable using the `Slider Switch` block, double-click the `Slider Switch` block to open the dialog box. Select a block in the model canvas. The tunable parameter or variable appears in the dialog box **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable to the block.

Limitations

The `Slider Switch` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.
Parameters that index a variable array do not appear in the **Connection** table.	For example, a block parameter specified using the variable `engine(1)` will not appear in the table because the parameter uses an index of the variable `engine`, which is not a scalar variable. To make the parameter appear in the **Connection** table, change the block parameter field to a scalar variable, such as `engine_1`.

Parameters and Dialog Box



Connection

Select a block to connect and control a tunable parameter or variable.

To control a tunable parameter or variable, select a block in the model. The tunable parameter or variable appears in the **Connection** table. Select the option button next

to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable.

Settings

The table has a row for the tunable parameter or variable connected to the block. If there are no tunable parameters or variables selected in the model or the block is not connected to any tunable parameters or variables, then the table is empty.

States

Switch values and labels.

Settings

Default Labels: `Off` and `On`

Default Values: `0` and `1`

By default, the **Off** state label corresponds to the set value of `0`, and the **On** state label corresponds to the set value of `1`.

The state labels appear on the outside of the switch. You can change the state labels to another text string. You can change the state values to any real value that is between negative `realmax` and positive `realmax`.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top
: Show the label at the top of the block.

Bottom
: Show the label at the bottom of the block.

Hide
: Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

Introduced in R2015a

Sqrt, Signed Sqrt, Reciprocal Sqrt

Calculate square root, signed square root, or reciprocal of square root

Library

Math Operations

Description 

You can select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
`sqrt`	Square root of the input	$u^{0.5}$	`sqrt`
`signedSqrt`	Square root of the absolute value of the input, multiplied by the sign of the input	$\mathrm{sign}(u)*\lvert u\rvert^{0.5}$	—
`rSqrt`	Reciprocal of the square root of the input	$u^{-0.5}$	—

The block 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:

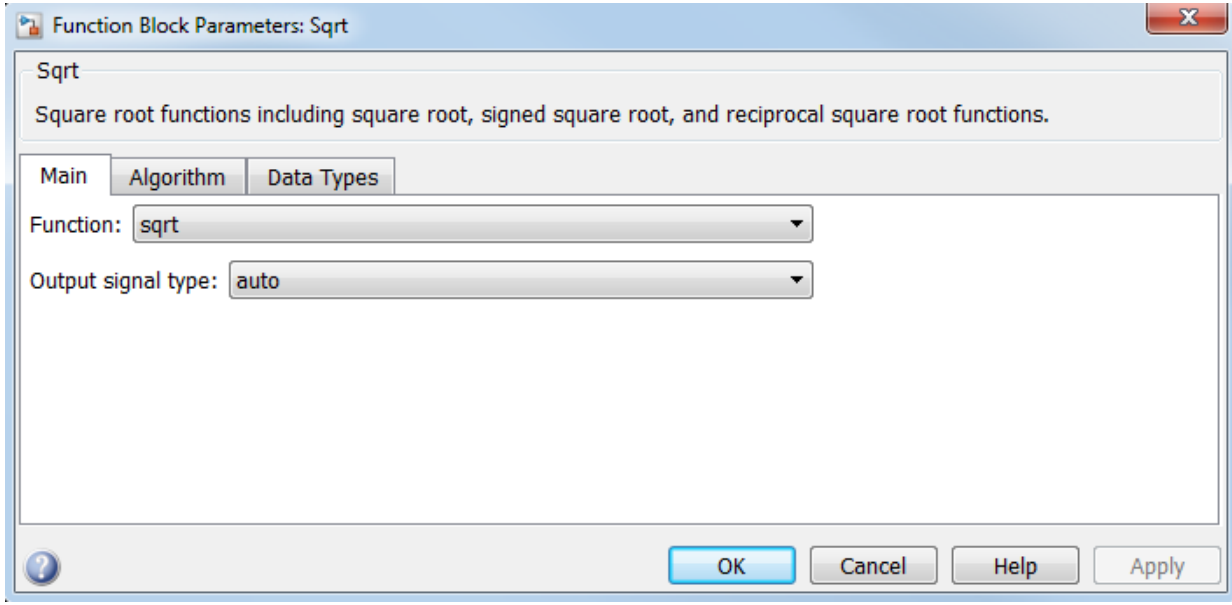
Function	Types of Real Inputs	Types of Complex Inputs
`sqrt`	Any, except for fixed-point inputs that are negative or have nontrivial slope and nonzero bias	Any, except for fixed-point inputs
`signedSqrt`		None
`rSqrt`		None

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**

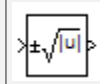
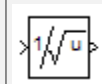
Parameters and Dialog Box

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



Function

Specify the mathematical function. The block icon changes to match the function you select.

Function	Block Icon
`sqrt`	
`signedSqrt`	
`rSqrt`	

Output signal type

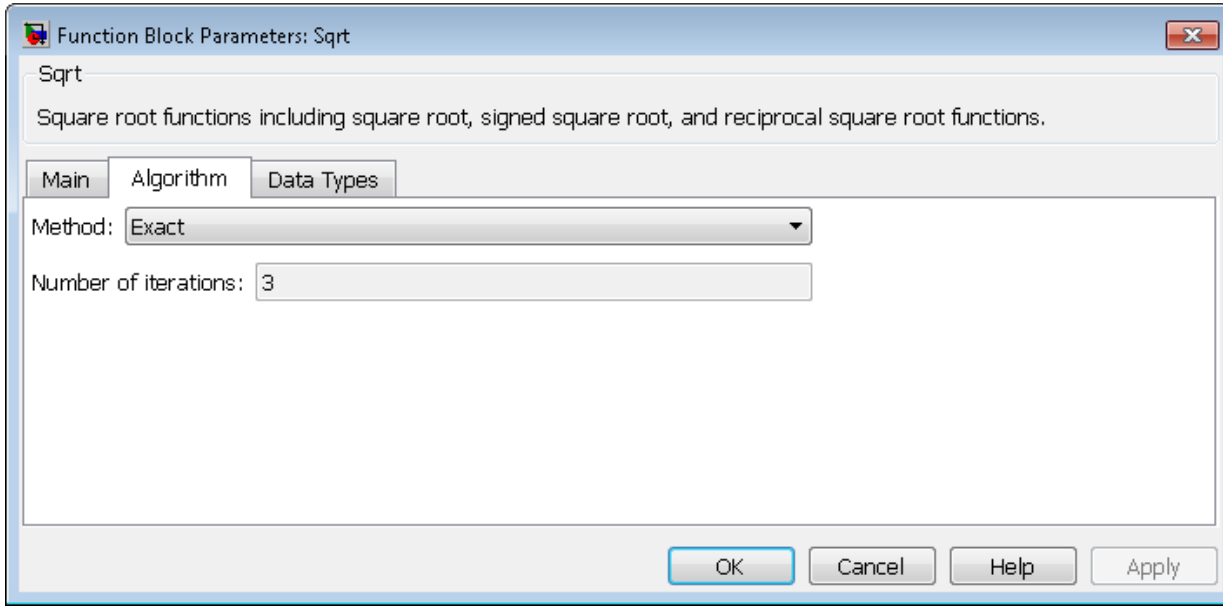
Specify the output signal type of the block as `auto`, `real`, or `complex`.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
`sqrt`	`real`	`real` for nonnegative inputs `NaN` for negative inputs	`real` for nonnegative inputs `NaN` for negative inputs	`complex`
	`complex`	`complex`	`error`	`complex`
`signedSqrt`	`real`	`real`	`real`	`complex`
	`complex`	`error`	`error`	`error`
`rSqrt`	`real`	`real`	`real`	`error`
	`complex`	`error`	`error`	`error`

Sample time

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

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



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.

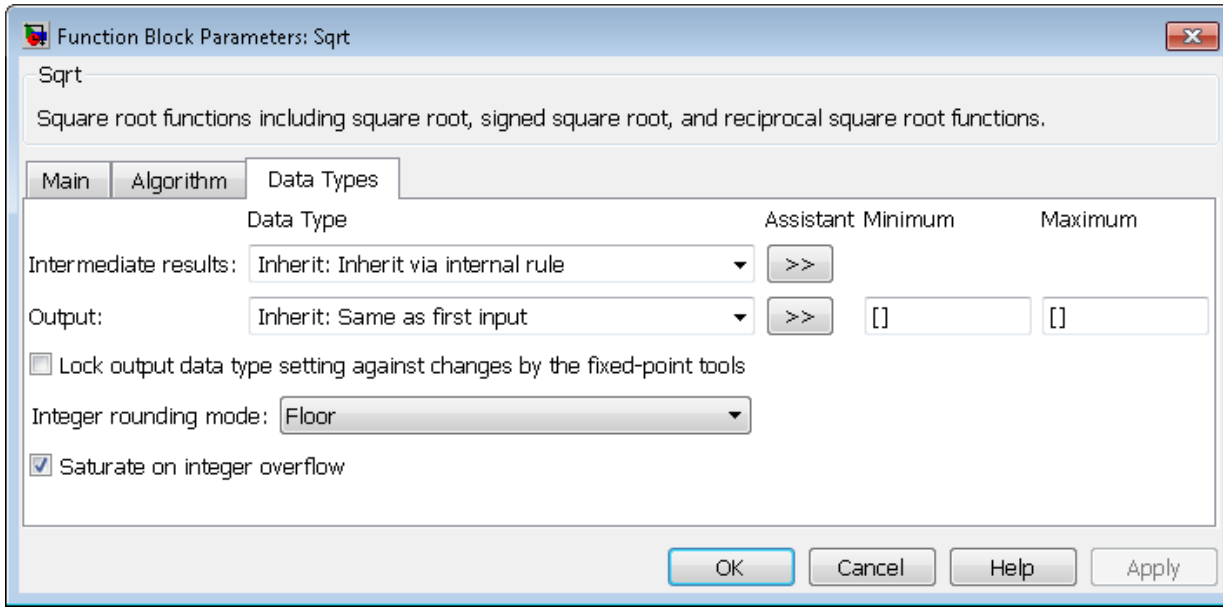
Number of iterations

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:



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.

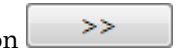
Under these conditions, selecting `Inherit:Inherit from output` yields suboptimal performance and produces an error.

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 "Control Signal Data Types" in the Simulink User's Guide for more information.

Output minimum

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

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

Output maximum

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

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

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

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

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

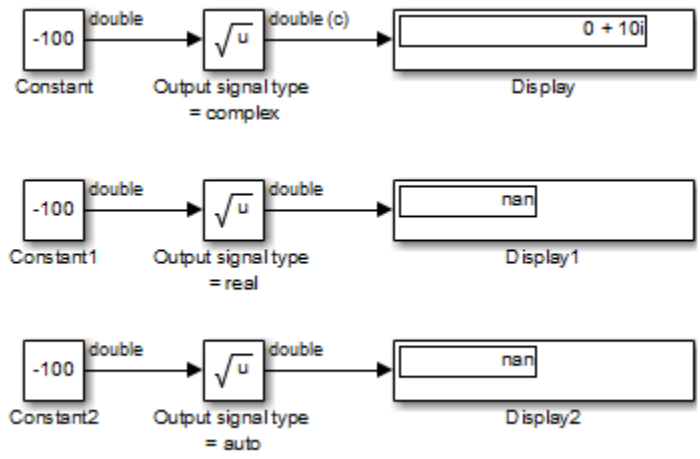
Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

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

Examples

sqrt Function

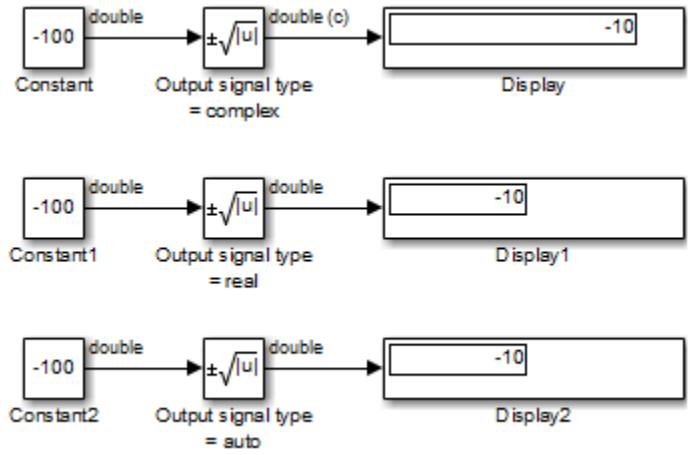
Suppose that you have the following model:



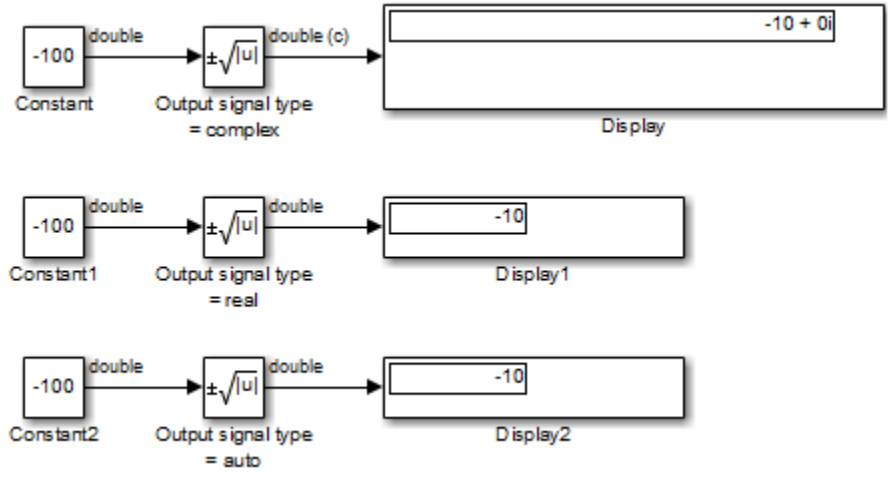
When the input to the Sqrt block is negative and the **Output signal type** is `auto` or `real`, the `sqrt` function outputs `NaN`. However, setting **Output signal type** to `complex` produces the correct answer.

signedSqrt Function

Suppose that you have the following model:

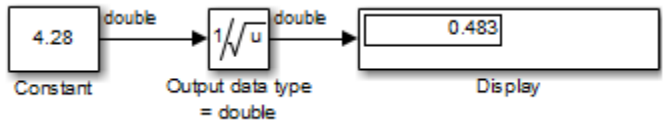


When the input to the Sqrt block is negative, the block output is the same for any **Output signal type** setting. If you change the first `Display` block format from `short` to `decimal (Stored Integer)`, you see the value of the imaginary part for the complex output.



rSqrt Function with Floating-Point Inputs

Suppose that you have the following model:



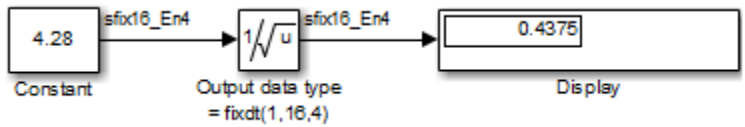
In the Sqrt block dialog box, assume that the following parameter settings apply:

Parameter	Setting
Method	`Newton-Raphson`
Number of iterations	1
Intermediate results data type	`Inherit:Inherit from input`

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0004 of the final value (0.4834).

rSqrt Function with Fixed-Point Inputs

Suppose that you have the following model:



In the Sqrt block dialog box, assume that the following parameter settings apply:

Parameter	Setting
Method	`Newton-Raphson`
Number of iterations	1
Intermediate results data type	`Inherit:Inherit from input`

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0459 of the final value (0.4834).

Characteristics

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

See Also

`Math Function`, `Trigonometric Function`

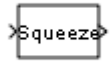
Introduced in R2010a

Squeeze

Remove singleton dimensions from multidimensional signal

Library

Math Operations



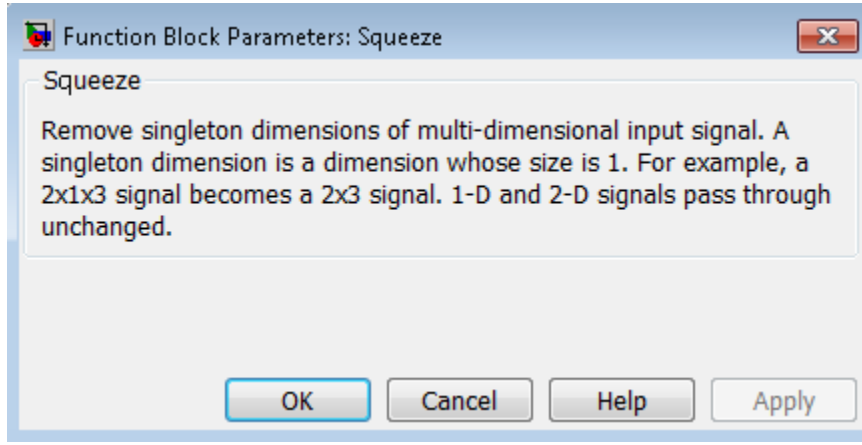
Description

The Squeeze block removes singleton dimensions from its multidimensional input signal. A singleton dimension is any dimension whose size is one. The Squeeze block operates only on signals whose number of dimensions is greater than two. Scalar, one-dimensional (vector), and two-dimensional (matrix) signals pass through the Squeeze block unchanged.

Data Type Support

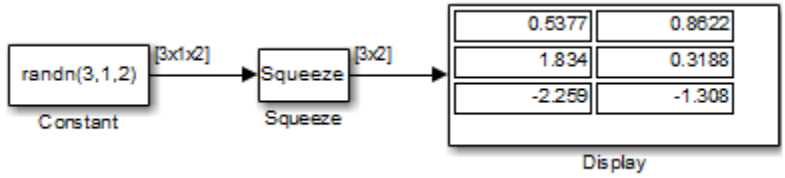
The Squeeze block accepts input signals of any dimension and of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Examples

In the following model, the Squeeze block converts a multidimensional array of size 3-by-1-by-2 into a 3-by-2 signal:



Because the Constant block supplies a signal with random values to the Squeeze block, the values in the Display block vary from simulation to simulation.

Characteristics

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

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

See Also

`Reshape`

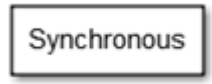
Introduced in R2007b

State Control

Specify synchronous reset and enable behavior for blocks with state

Library

HDL Coder / HDL Subsystems

Description 

Note: This page outlines the behavior of the block in Simulink. For information about the behavior of this block in HDL Coder, see `State Control` in the HDL Coder documentation.

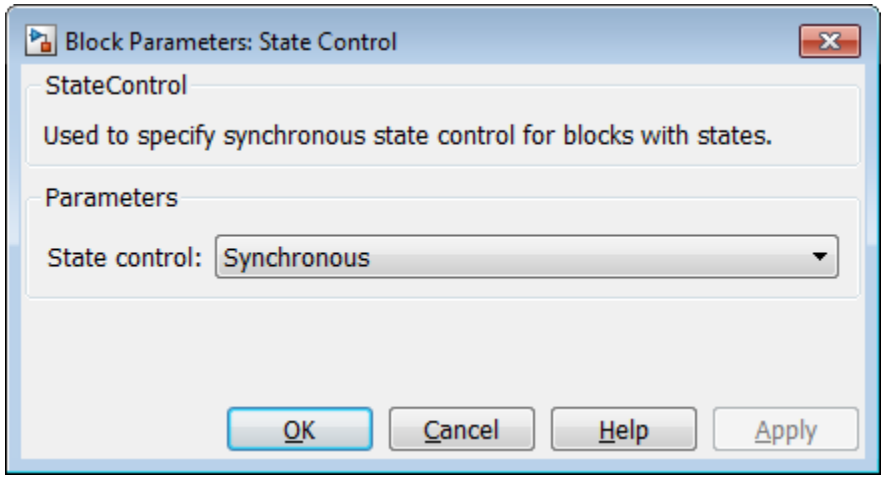
The `State Control` block enables you to choose between synchronous or classic simulation semantics for the subsystem it is placed in.

State Control Block Modes

Functionality	Synchronous Semantics	Classic Semantics
`State Control` block setting	Default block setting when you add the block from the HDL Subsystems block library.	The simulation behavior is the same as a subsystem that does not use the `State Control` block.
Simulink simulation behavior: • Update method: Updates states. • Output method: Computes output values.	When the subsystem is disabled, the update method is disabled and the states of the subsystem are held. The output method still computes the output based on the input signal and held states.	The output and update methods are disabled when the subsystem is disabled. For example, when you have enabled subsystems, the output value stays the same or resets to the initial condition, based on the settings of the

Functionality	Synchronous Semantics	Classic Semantics
		`Outport` block. The subsystem states can be held or reset based on the settings of the `Enable` block.

Dialog Box and Parameters



State control

Specify whether to use synchronous or classic semantics. The default is `Synchronous`.

Limitations

The following limitations apply to using the State Control block in Simulink. For information about this block in HDL Coder, see `State Control` in the HDL Coder documentation.

Block-Level Limitations

- For synchronous semantics in S-function blocks, set the method `ssSetStateSemanticsClassicAndSynchronous` to `true`.

- `Discrete-Time Integrator` blocks with a reset port do not support synchronous semantics.
- For synchronous semantics in `MATLAB System` blocks, set the method `getExecutionSemanticsImpl` to `Synchronous'` or `{'Classic', 'Synchronous'}`. If unset, the default value is `'Classic'`.
- All action subsystems connected to `If` and `Switch Case` blocks must have the same semantics, either classic or synchronous.
- The following blocks are not allowed in synchronous mode:
  - Continuous time blocks and blocks with continuous rate
  - Simulink blocks with **Input processing** set to `Column as channels (frame based)`, where this parameter applies.
  - `Trigger` block
  - `From Workspace` block
  - The set of unit delay blocks in the **Additional Math & Discrete** > **Additional Discrete** sublibrary in Simulink, such as the `Unit Delay Resettable` and `Unit Delay External IC` blocks

Subsystem-level Limitations

- Conditional subsystems using classic semantics cannot have subsystems with synchronous semantics inside them.
- `Synchronous Enabled Subsystem` cannot contain reset subsystems or a reset parameter port. For example, you cannot have a `Delay` block with an external reset port inside the subsystem.
- The following subsystem constructs are not allowed in synchronous mode:
  - Conditional subsystems with blocks having different sample times
  - `For Iterator Subsystem`, `While Iterator Subsystem`, `Function-Call Subsystem`, and `Triggered Subsystem`

Model-Level Limitations

- Variable-size signals are not supported with synchronous semantics.
- Synchronous semantics do not propagate across model boundaries. If your parent model has synchronous semantics, any referenced model must have synchronous

semantics explicitly specified. At the root level of each referenced model, add a `State Control` block with the **State control** parameter set to `Synchronous`.

See Also

`Enable` | `Enabled Subsystem` | `Enabled Synchronous Subsystem` | `Synchronous Subsystem`

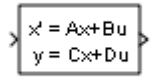
Introduced in R2016a

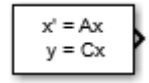
State-Space

Implement linear state-space system

Library

Continuous





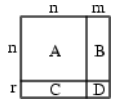
Description

The State-Space block implements a system whose behavior you define as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$
$$x\big|_{t=t_0} = x_0,$$

where x is the state vector, u is the input vector, y is the output vector and $x0$ is the initial condition of the state vector. The matrix coefficients must have these characteristics:

- A must be an n-by-n matrix, where n is the number of states.
- B must be an n-by-m matrix, where m is the number of inputs.
- C must be an r-by-n matrix, where r is the number of outputs.
- D must be an r-by-m matrix.



In general, the block has one input port and one output port. The number of rows in C or D matrix is the same as the width of the output port. The number of columns in

the B or D matrix are the same as the width of the input port. If you want to model an autonomous linear system with no inputs, set the B and D matrices to empty. In this case, the block acts as a source block with no input port and one output port, and implements the following system:

$$\dot{x} = Ax$$
$$y = Cx$$
$$x\big|_{t=t_0} = x_0.$$

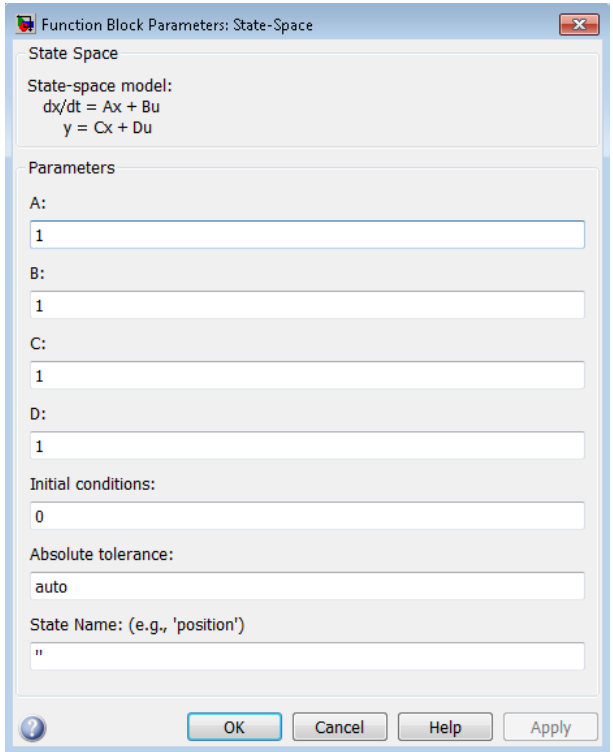
Simulink software converts a matrix containing zeros to a sparse matrix for efficient multiplication.

Data Type Support

A State-Space block accepts and outputs real signals of type `double`.

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

Parameters and Dialog Box



A

Specify the n-by-n matrix coefficient, where n is the number of states.

Settings

Default: 1

Command-Line Information

Parameter: A
Type: matrix
Value: `'1'`
Default: `'1'`

B

Specify the n-by-m matrix coefficient, where n is the number of states and m is the number of inputs.

Settings

Default: 1

Command-Line Information

Parameter: B
Type: matrix
Value: '1'
Default: '1'

C

Specify the r-by-n matrix coefficient, where r is the number of outputs and n is the number of states.

Settings

Default: 1

Command-Line Information

Parameter: C
Type: matrix
Value: '1'
Default: '1'

D

Specify the r-by-m matrix coefficient, where r is the number of outputs and m is the number of inputs.

Settings

Default: 1

Command-Line Information

Parameter: D
Type: matrix
Value: '1'
Default: '1'

Initial conditions

Specify the initial state vector.

Settings

Default: `0`

The initial conditions of this block cannot be `inf` or `NaN`.

Command-Line Information

Parameter: `X0`
Type: vector
Value: `'0'`
Default: `'0'`

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, –1, a positive real scalar or vector.
- If you enter `auto` or –1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Command-Line Information

Parameter: `AbsoluteTolerance`
Type: string, scalar, or vector
Value: `'auto' | '-1' |` any positive real scalar or vector
Default: `'auto'`

State Name (e.g., 'position')

Assign a unique name to each state.

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

Parameter: `ContinuousStateAttributes`
Type: string
Value: `' '` | user-defined
Default: `' '`

Examples

The following Simulink examples show how to use the State-Space block:

- `sldemo_dblcart1`
- `aero_vibrati`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Only if **D** ≠ 0
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Discrete State-Space`

Introduced before R2006a

Step

Generate step function

Library

Sources

Description

The Step block provides a step between two definable levels at a specified time. If the simulation time is less than the **Step time** parameter value, the block's output is the **Initial value** parameter value. For simulation time greater than or equal to the **Step time**, the output is the **Final value** parameter value.

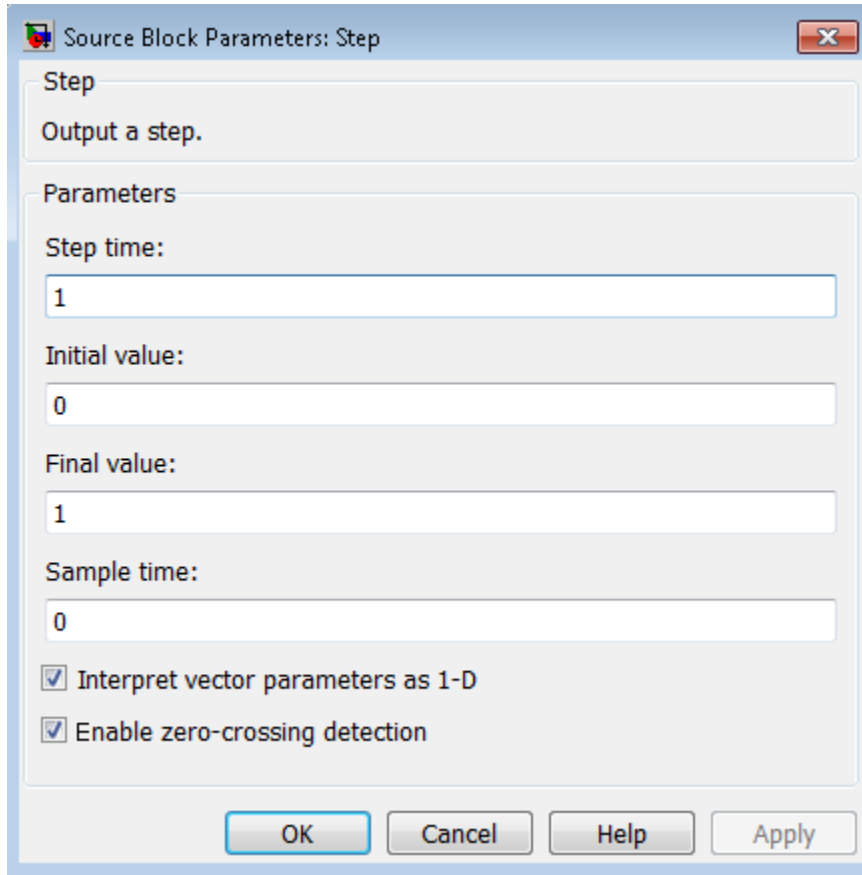
The numeric block parameters must be of the same dimensions after scalar expansion. If the **Interpret vector parameters as 1-D** option is off, the block outputs a signal of the same dimensions and dimensionality as the parameters. If the **Interpret vector parameters as 1-D** option is on and the numeric parameters are row or column vectors (that is, single row or column 2-D arrays), the block outputs a vector (1-D array) signal. Otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support

The Step block outputs real signals of type `double`.

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

Parameters 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

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

See Also

Ramp

Introduced before R2006a

Stop Simulation

Stop simulation when input is nonzero

Library

Sinks

Description

The Stop Simulation block stops the simulation when the input is nonzero. The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop.

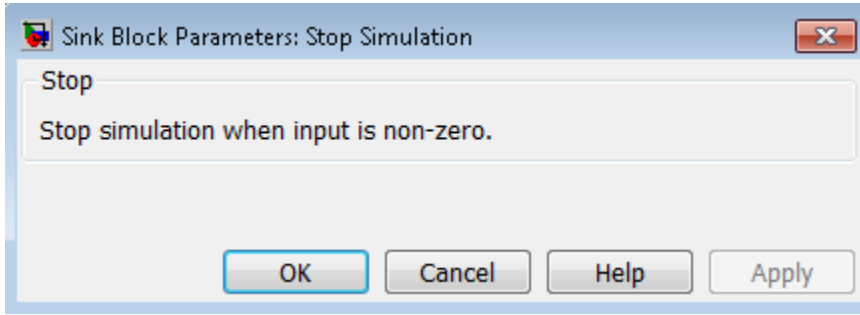
When you use the Stop Simulation block in a `For Iterator` subsystem, the stop action occurs after execution of *all* the iterations in the subsystem during a time step. The stop action does not interrupt execution until the start of the next time step.

You cannot use the Stop Simulation block to pause the simulation. To create a block that pauses the simulation, see “Pause Simulation Using Assertion Blocks” in the Simulink documentation.

Data Type Support

The Stop Simulation block accepts real signals of type `double` or `Boolean`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

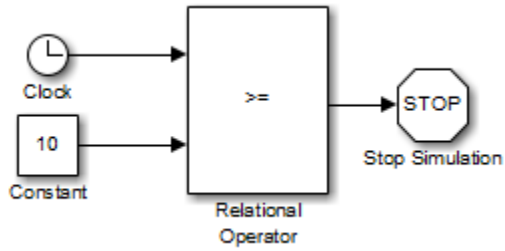
Parameters and Dialog Box



Examples

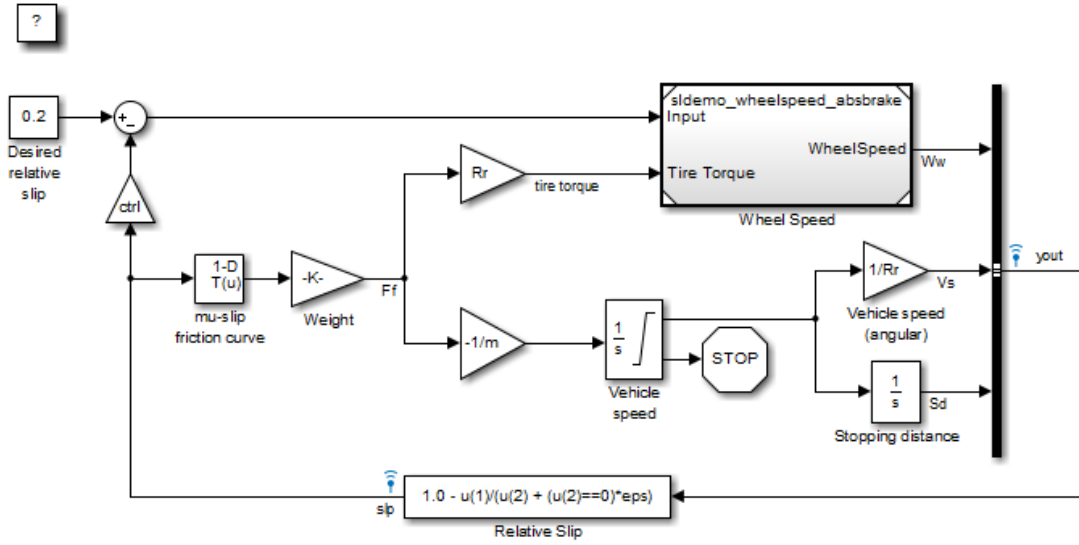
Usage with the Relational Operator Block

You can use the Stop Simulation block with the `Relational Operator` block to control when a simulation stops. For example, the following model stops simulation when the simulation time reaches 10.



Usage with the Integrator Block

You can use the Stop Simulation block with the `Integrator` block to control when a simulation stops. For example, the `sldemo_absbrake` model stops simulation when the saturation port of the Integrator block outputs a value of 1 or –1.



Characteristics

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

Introduced before R2006a

Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

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. Virtual subsystems do not have checksums.

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 1-1916 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.

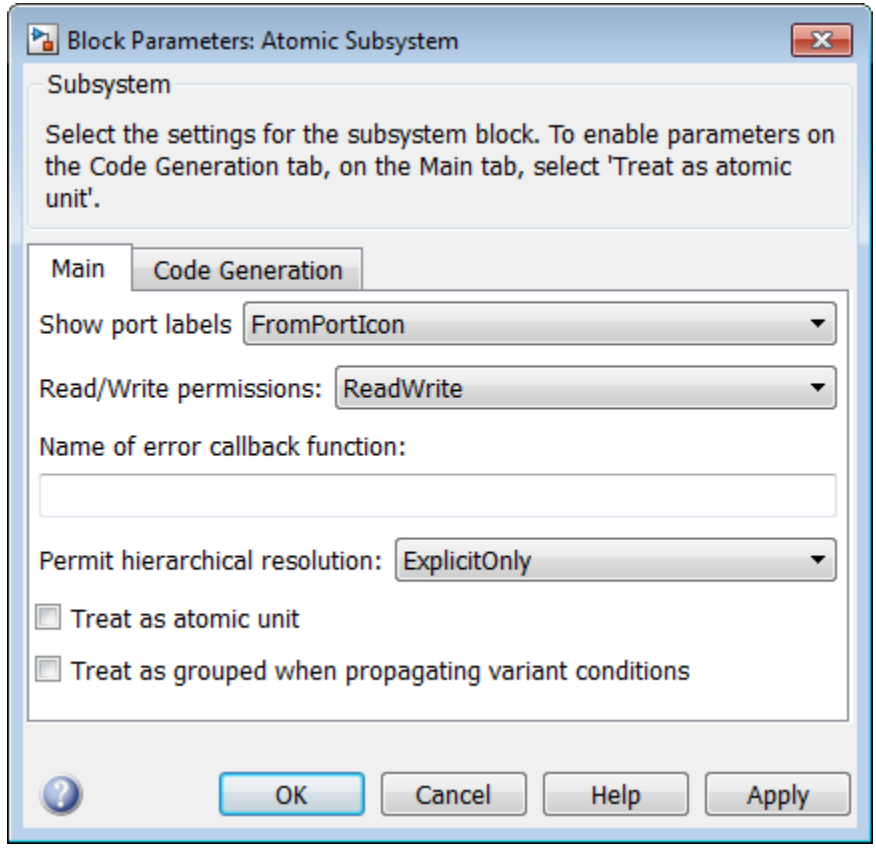
See “Create a Subsystem” for more information.

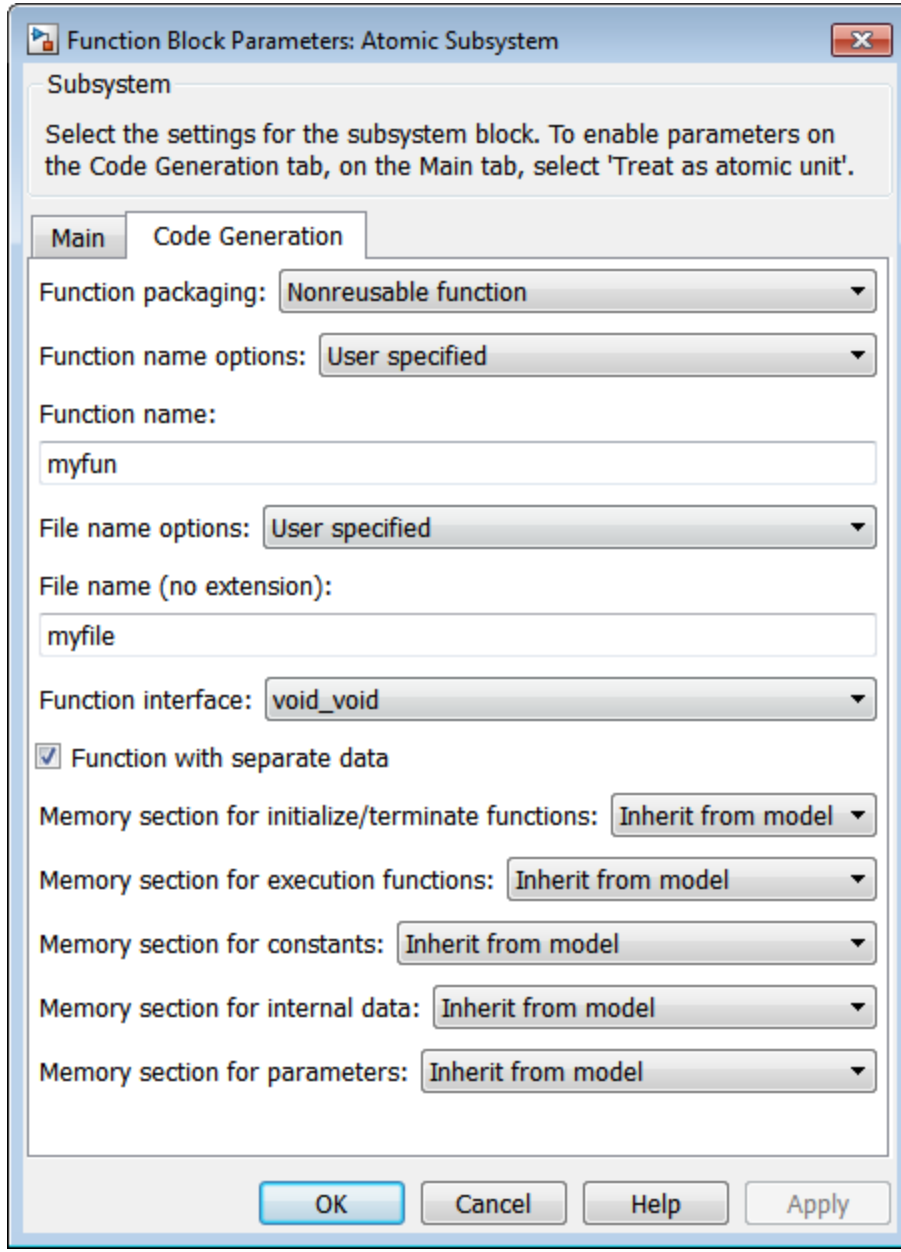
Data Type Support

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
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 6-101 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 6-101 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 6-101 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 6-101 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 1-1911.
- "Sample time (-1 for inherited)" on page 1-1914
- "Function packaging" on page 1-1916 (requires a Simulink Coder license)

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from `Variant Source` blocks or to `Variant Sink` blocks.

Settings

Default: On

 On

Simulink treats the subsystem as a unit when propagating variant conditions from `Variant Source` blocks or to `Variant Sink` blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

 Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 1-1909 enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 6-101 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 6-101 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 1-1909 enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Variant control

Enter the variant activation condition or the variant control that contains the expression for variant activation.

The variant control can be a boolean condition expression or a `Simulink.Variant` object representing a boolean condition expression. If you want to generate code for your model, define control variables as `Simulink.Parameter` objects.

Settings

Default: `Variant`

Dependency

Adding a `Subsystem` block inside a `Variant Subsystem` block 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`

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 1-1925 parameter setting. You can name the generated function and file using parameters “Function name” on page 1-1920 and “File name (no extension)” on page 1-1923. 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 for each subsystem instance.
  - `Reusable function` yields a separate function with arguments for each subsystem instance in the model.
- If you select `Reusable function` while your generated code is under source control, set **File name options** to `Use subsystem name`, `Use function name`, or `User specified`. Otherwise, the names of your code files change whenever you modify your model, which prevents source control on your files.

Dependencies

- This parameter requires a Simulink Coder license.
- "Treat as atomic unit" on page 1-1909 enables this parameter.
- Setting this parameter to `Nonreusable function` or `Reusable function` enables the following parameters:
  - "Function name options" on page 1-1918
  - "File name options" on page 1-1921
  - "Memory section for initialize/terminate functions" on page 1-1926 (requires a license for Embedded Coder software and an ERT-based system target file)
  - "Memory section for execution functions" on page 1-1927 (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 1-1924 (requires a license for Embedded Coder software and an ERT-based system target file).

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Function name options

Specify how Simulink Coder software is to name the function it generates for the subsystem.

If you have an Embedded Coder license, you can control function names with options on the Configuration Parameter **Code Generation** > **Symbols** pane.

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. Be default, the function name uses the naming convention *`model`*`, _`*`subsystem`*.

Note When a subsystem is a library block and the subsystem parameter "Function packaging" on page 1-1916 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 1-1916 to `Nonreusable function` or `Reusable function` enables this parameter.

- Setting this parameter to `User specified` enables the "Function name" on page 1-1920 parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 1-1918 to `User specified` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 1-1916 to `Nonreusable function` or `Reusable function` enables this parameter.
- Setting this parameter to `User specified` enables the "File name (no extension)" on page 1-1923 parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 1-1921 to `User specified` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 1-1916 to `Nonreusable function` enables this parameter.
- Selecting this check box enables these parameters:
  - "Memory section for constants" on page 1-1928
  - "Memory section for internal data" on page 1-1930
  - "Memory section for parameters" on page 1-1932

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 1-1916 to `Nonreusable function` enables this parameter.

Command-Line Information

For the command-line information, see "Block-Specific Parameters" on page 6-101.

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 “Control Data and Function Placement in Memory by Inserting Pragmas” 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 1-1916 to `Nonreusable function` or `Reusable function` enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 "Control Data and Function Placement in Memory by Inserting Pragmas" 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 1-1916 to `Nonreusable function` or `Reusable function` enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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

- The memory section that you specify applies to the corresponding global data structures in the generated code. For basic information about the global data structures generated for atomic subsystems, see "Default Data Structures in the Generated Code".
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See "Control Data and Function Placement in Memory by Inserting Pragmas" 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 1-1916 to `Nonreusable function` and selecting the "Function with separate data" on page 1-1924 check box enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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

- The memory section that you specify applies to the corresponding global data structures in the generated code. For basic information about the global data structures generated for atomic subsystems, see "Default Data Structures in the Generated Code".
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See "Control Data and Function Placement in Memory by Inserting Pragmas" 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 1-1916 to `Nonreusable function` and selecting the "Function with separate data" on page 1-1924 check box enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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

- The memory section that you specify applies to the corresponding global data structure in the generated code. For basic information about the global data structures generated for atomic subsystems, see "Default Data Structures in the Generated Code".
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See "Control Data and Function Placement in Memory by Inserting Pragmas" 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 1-1916 to `Nonreusable function` and selecting the "Function with separate data" on page 1-1924 check box enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Depends on the blocks in the subsystem
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, for enable and trigger ports if present
Code Generation	Yes

Introduced in R2007a

Sum, Add, Subtract, Sum of Elements

Add or subtract inputs

Library

Math Operations

Description

The Sum block performs addition or subtraction on its inputs. This block can add or subtract scalar, vector, or matrix inputs. It can also collapse the elements of a signal.

You specify the operations of the block with the **List of signs** parameter. Plus (+), minus (-), and spacer (|) characters indicate the operations to be performed on the inputs:

- If there are two or more inputs, then the number of + and - characters must equal the number of inputs. For example, "+-+" requires three inputs and configures the block to subtract the second (middle) input from the first (top) input, and then add the third (bottom) input.
- All nonscalar inputs must have the same dimensions. Scalar inputs will be expanded to have the same dimensions as the other inputs.
- A spacer character creates extra space between ports on the block's icon.
- For a round Sum block, the first input port is the port closest to the 12 o'clock position going in a counterclockwise direction around the block. Similarly, other input ports appear in counterclockwise order around the block.
- If only addition of all inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of "+" characters.
- If only one input port is required, a single "+" or "-" collapses the element via the specified operation.

The Sum block first converts the input data type(s) to its accumulator data type, then performs the specified operations. The block converts the result to its output data type using the specified rounding and overflow modes.

Calculation of Block Output

Output calculation for the Sum block depends on the number of block inputs and the sign of input ports:

If the Sum block has...	And...	The formula for output calculation is...	Where...
One input port	The input port sign is +	y = e[0] + e[1] + e[2] ... + 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

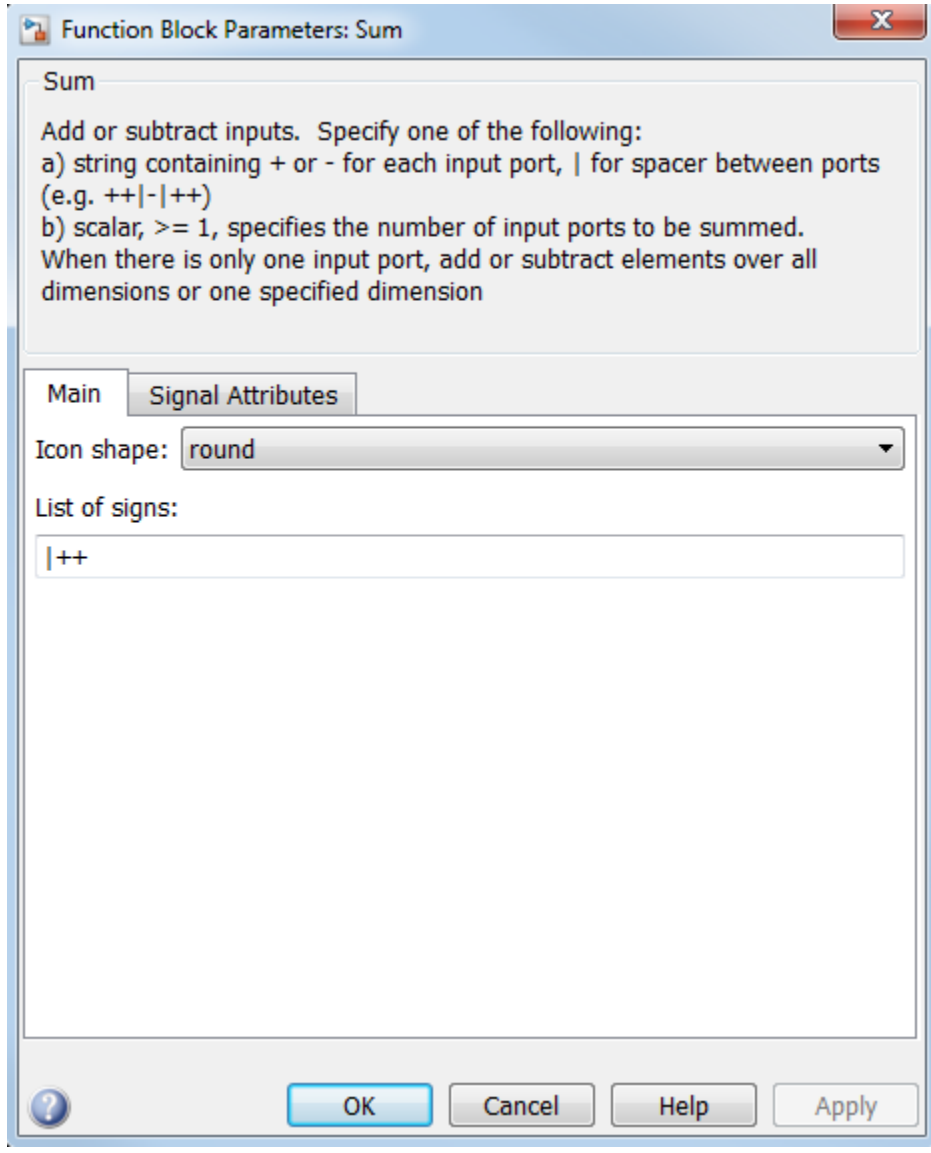
The Sum block accepts real or complex signals of the following data types:

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

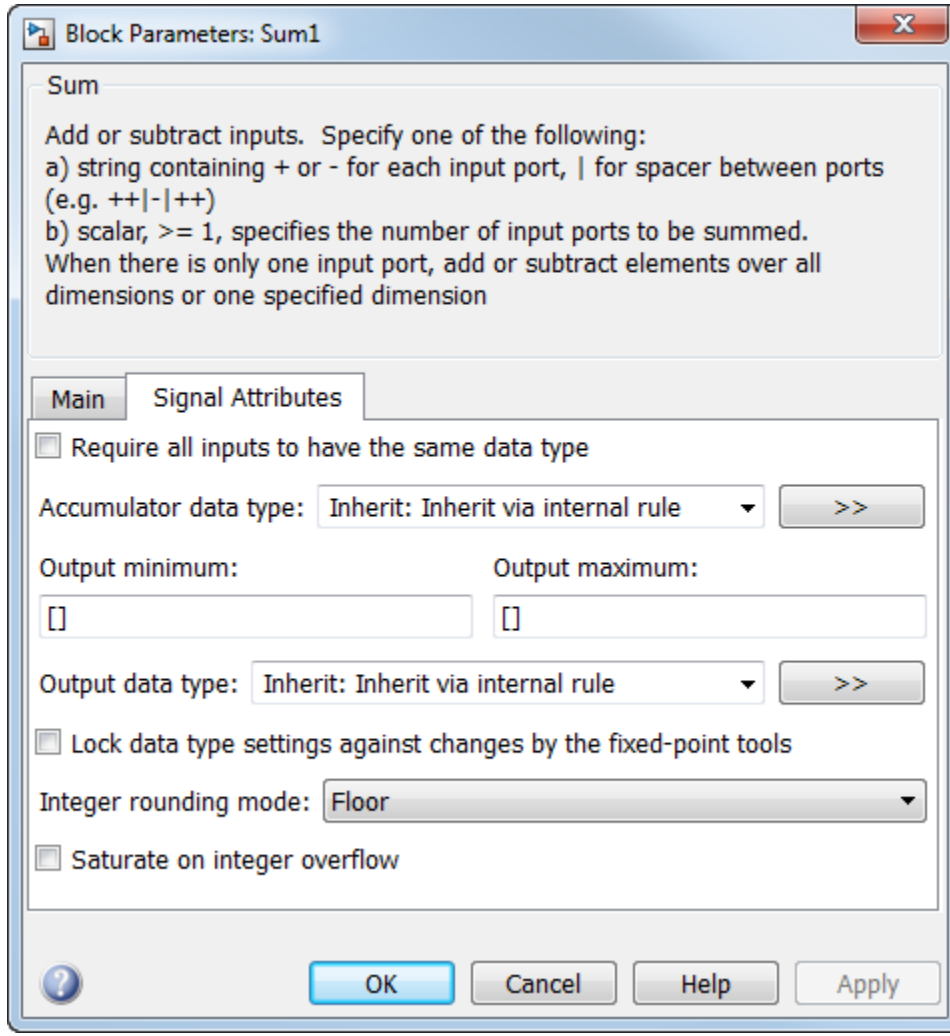
The inputs can be of different data types, unless you select the **Require all inputs to have the same data type** parameter. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

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



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



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 “Control Signal Data Types”.

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 6-101 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 6-101 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 6-101 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 6-101 for the command-line information.

Sample time

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

Require all inputs to have the same data type

Require that all inputs have the same data type.

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 6-101 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 6-101 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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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 6-101 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 6-101 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 6-101 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 6-101 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 6-101 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 6-101 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 6-101 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 6-101 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.

Simulink uses the minimum to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

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

Simulink uses the maximum value to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

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 6-101 for the command-line information.

See Also

For more information, see "Control Signal 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 6-101 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.

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.

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 6-101 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.

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.

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.

See Also

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

Examples

How the Sum Block Reorders Inputs

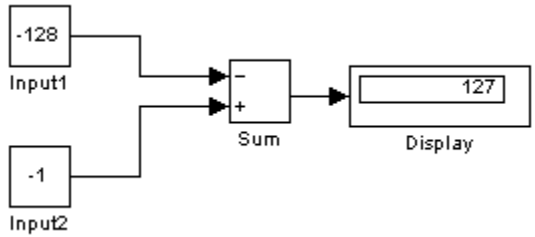
If you use `-` on the first input port, the Sum block reorders the inputs so that, if possible, the first input uses a `+` operation. For example, in the expression `output = -a-b+c`, the

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`

Step	Block Operation
4	Calculates the block output: `Output = int8(127) = 127`

If the Sum block does *not* reorder the inputs, the following operations occur instead and you get the nonideal result of 126.

Step	Block Operation
1	Initializes the accumulator by using the first input port: `Accumulator = int8(-(-128)) = 127` Because saturation is on, the initial value of the accumulator saturates at 127 and does not wrap.
2	Continues to accumulate values: `Accumulator = Accumulator + int8(-1) = 126`
3	Calculates the block output: `Output = int8(126) = 126`

Characteristics

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

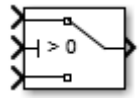
Introduced before R2006a

Switch

Switch output between first input and third input based on value of second input

Library

Signal Routing

Description 

Types of Block Inputs

The Switch block passes through the first input or the third input based on the value of the second input. The first and third inputs are called *data inputs*. The second input is called the *control input*. Specify the condition under which the block passes the first input by using the **Criteria for passing first input** and **Threshold** parameters.

To immediately back propagate a known output data type to the first and third input ports, set the **Output data type** parameter to `Inherit: Inherit via internal rule` and select the **Require all data port inputs to have the same data type** check box.

Limitations on Data Inputs

The sizes of the two data inputs can be different if you select **Allow different data input sizes** on the block dialog box. However, this block does not support variable-size input signals. Therefore, the size of each input cannot change during simulation.

If the data inputs to the Switch block are buses, the element names of both buses must be the same. Using the same element names ensures that the output bus has the same element names no matter which input bus the block selects. To ensure that your model meets this requirement, use a bus object to define the buses and set the **Element name mismatch** diagnostic to `error`. See “Connectivity Diagnostics Overview” for more information.

Block Icon Appearance

The block icon helps you identify **Criteria for passing first input** and **Threshold** without having to open the block dialog box.

For information about port order for various block orientations, see "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.

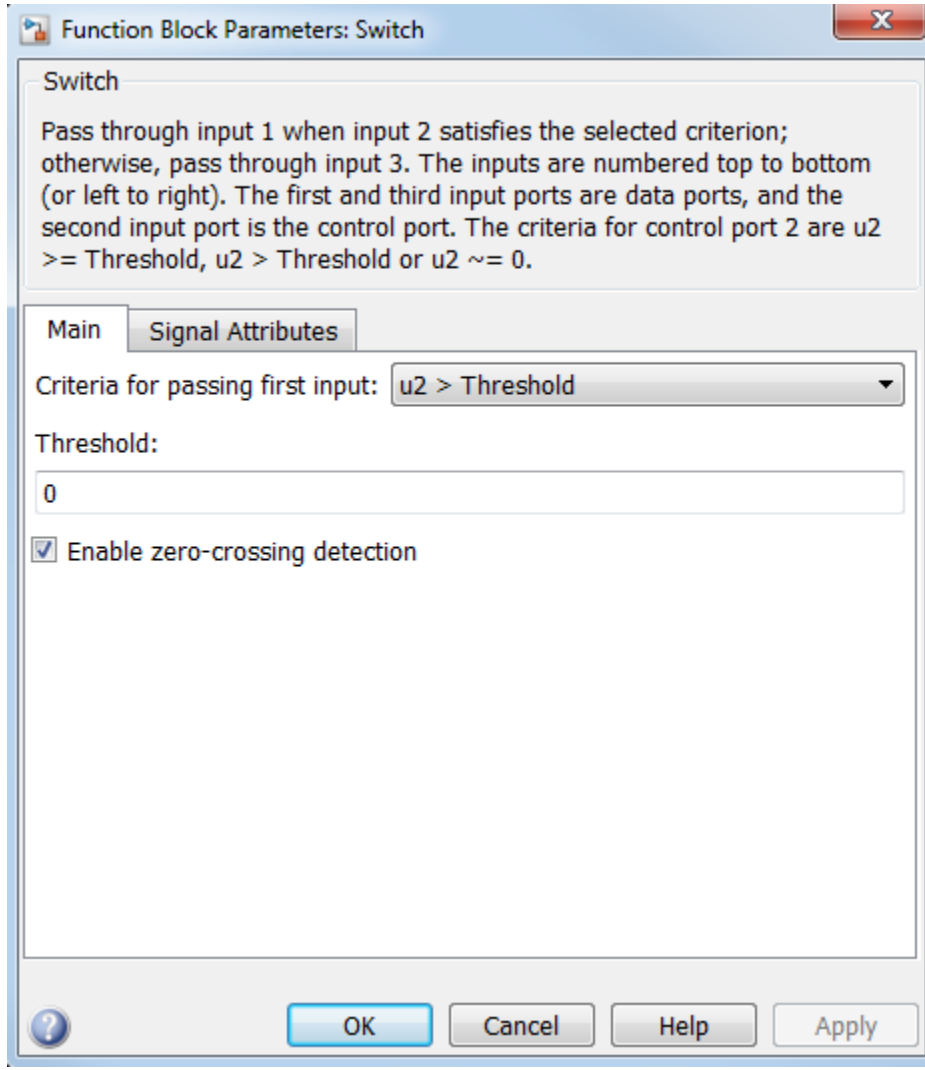
The data inputs can be of any data type that Simulink supports. If either data input is of an enumerated type, the other must be of the same enumerated type.

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 Switch block dialog box appears as follows:



- “Criteria for passing first input” on page 1-1977
- “Threshold” on page 1-1979
- “Enable zero-crossing detection” on page 1-1555
- “Sample time” on page 1-298
- “Require all data port inputs to have the same data type” on page 1-1982
- “Lock output data type setting against changes by the fixed-point tools” on page 1-235
- “Integer rounding mode” on page 1-295
- “Saturate on integer overflow” on page 1-297
- “Allow different data input sizes” on page 1-1987
- “Output minimum” on page 1-299
- “Output maximum” on page 1-300
- “Output data type” on page 1-1990
- “Mode” on page 1-1992
- “Data type override” on page 1-230
- “Signedness” on page 1-231
- “Word length” on page 1-232
- “Scaling” on page 1-225
- “Fraction length” on page 1-233
- “Slope” on page 1-234
- “Bias” on page 1-234

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 6-101 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 6-101 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

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Sample time

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

Require all data port inputs to have the same data type

Require all data inputs to have the same data type.

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 6-101 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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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: `AllowDiffInputSizes`
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 "Control Block Parameter Value Ranges") 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 6-101 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 “Control Block Parameter Value Ranges”) 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 6-101 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 6-101 for the command-line information.

See Also

See "Control Signal 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 6-101 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 6-101 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 6-101 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**

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 6-101 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 6-101 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 6-101 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.

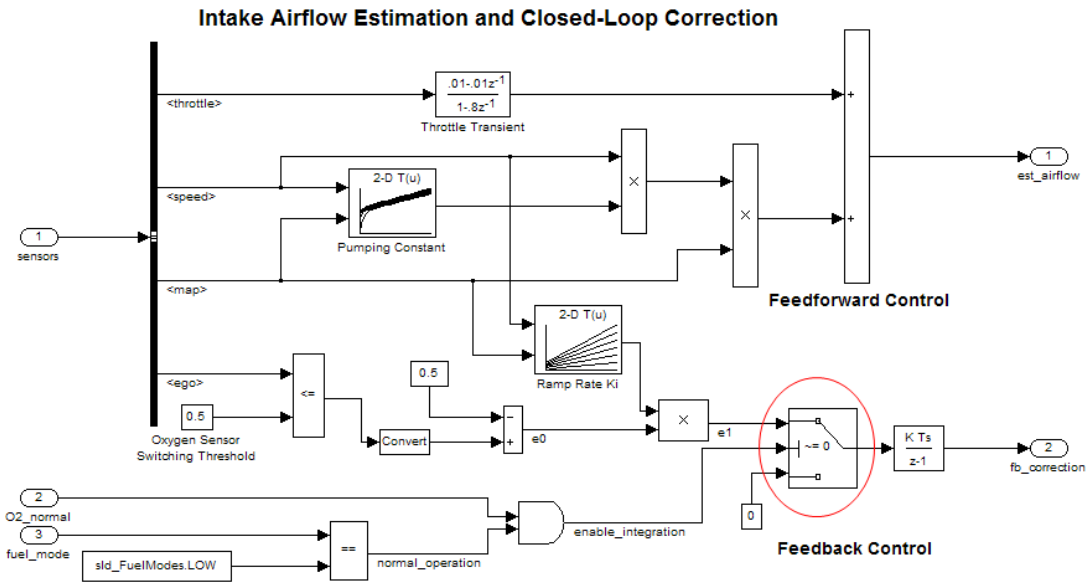
You can use a `Rate Transition` block to change the sample time of an individual signal, or of all signals in a bus. See "Composite Signals" and Bus-Capable Blocks for more information.

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:

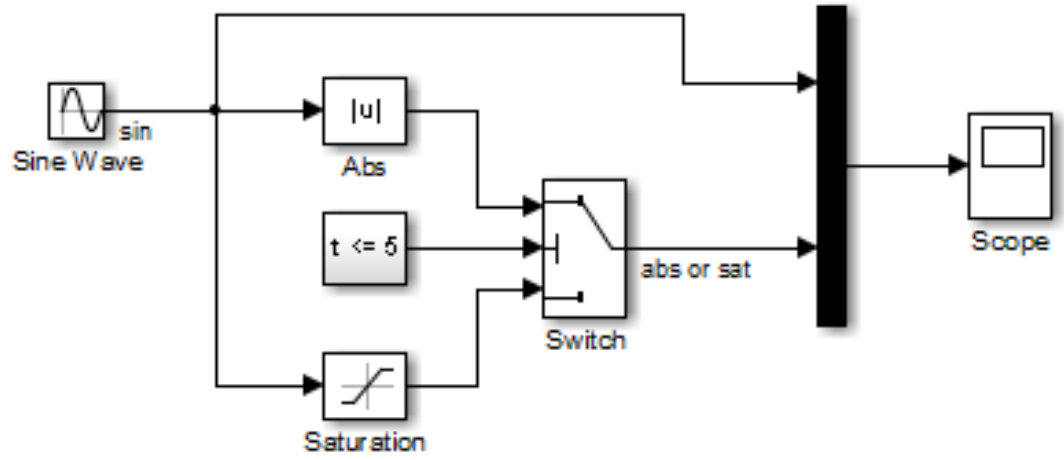


The value of the control port on the Switch block determines whether or not feedback correction occurs. The control port value depends on the output of the Logical Operator block.

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

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

See Also

`Multiport Switch`

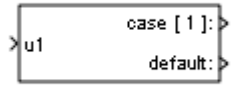
Introduced before R2006a

Switch Case

Implement C-like `switch` control flow statement

Library

Ports & Subsystems

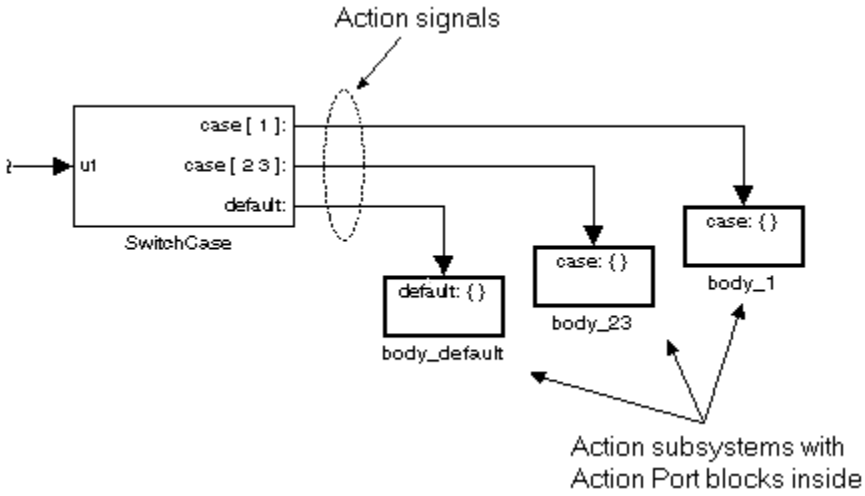


Description

A Switch Case block receives a single input. Each output port is attached to a `Switch Case Action Subsystem`. Data outputs are action signals to Switch Case Action subsystems, which you create with `Action Port` blocks and subsystems.

The Switch Case block uses its input value to select a case condition that determines which subsystem to execute. The cases are evaluated top down starting with the first case. If a case value (in brackets) corresponds to the value of the input, its `Switch Case Action Subsystem` is executed.

If a `default` case exists, it executes if none of the other cases executes. Providing a `default` case is optional, even if the other case conditions do not exhaust every possible value. The following diagram shows a completed Simulink `switch` control flow statement:



Cases for the Switch Case block contain an implied break after their Switch Case Action subsystems are executed. Thus there is no fall-through behavior for the Simulink `switch` control flow statement as found in standard C `switch` statements. The following pseudocode represents generated code for the preceding `switch` control example:

```
switch (u1) {
 case [u1=1]:
  body_1;
  break;
 case [u1=2 or u1=3]:
  body_23;
  break;
 default:
  body_default;
}
```

To construct the Simulink `switch` control flow statement shown in the above example:

1. Place a Switch Case block in the current system and attach the input port labeled `u1` to the source of the data you are evaluating.
2. Open the Switch Case block dialog box and update parameters:
   a. Populate the **Case conditions** field with the individual cases.
   b. To show a default case, select the **Show default case** check box.

3 Create a `Switch Case Action Subsystem` for each case port you added to the Switch Case block.

   These consist of subsystems with Action Port blocks inside them. When you place the Action Port block inside a subsystem, the subsystem becomes an atomic subsystem with an input port labeled `Action`.

4 Connect each case output port and the default output port of the Switch Case block to the Action port of an Action subsystem.

   Each connected subsystem becomes a case body. This is indicated by the change in label for the `Switch Case Action Subsystem` block and the Action Port block inside of it to the name `case{}`.

   During simulation of a `switch` control flow statement, the Action signals from the Switch Case block to each `Switch Case Action Subsystem` turn from solid to dashed.

5 In each `Switch Case Action Subsystem`, enter the Simulink logic appropriate to the case it handles.

Control Algorithm Execution Using Enumerated Signal

This example shows how to use a signal of an enumerated data type to control the execution of a block algorithm. For basic information about using enumerated data types in models, see "Use Enumerated Data in Simulink Models".

When you use enumerated data in a `Switch Case` block, follow these best practices:

- Use the same enumerated type for the input `u1` and all of the case condition values.
- Use a different underlying integer for each of the enumerated values that you specify in the **Case conditions** box.

Define Enumerated Type

Copy the enumerated type definition `ex_SwitchCase_MyColors` into a script file in your current folder.

```
classdef ex_SwitchCase_MyColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
```

```
    Mauve(3)
  end
end
```

Alternatively, you can use the function `Simulink.defineIntEnumType` to define the type.

```
Simulink.defineIntEnumType('ex_SwitchCase_MyColors',...
{'Red','Yellow','Blue','Mauve'},[0;1;2;3])
```

Explore Example Model

1. Open the example model `ex_enum_switch_case`.
2. Open the `Enumerated Constant` block dialog box. The constant output value is `ex_SwitchCase_MyColors.Blue`.
3. Open the `Switch Case` block dialog box. The **Case conditions** box is set to a cell array containing three of the four possible enumeration members. The block has four outputs corresponding to the three specified enumeration members and a default case.
4. Open the `Switch Case Action Subsystem` blocks. The subsystems each contain a `Constant` block that uses a unique constant value.

Control Execution During Simulation

1. In the Simulink Editor, set the simulation stop time to `Inf`.
2. Simulate the model. The `Display` block shows the value `5`, which corresponds to the case `ex_SwitchCase_MyColors.Blue`.
3. In the `Enumerated Constant` block dialog box, set **Value** to `ex_SwitchCase_MyColors.Red` and click **Apply**. The Display block shows `19`.
4. Set **Value** to `ex_SwitchCase_MyColors.Mauve` and click **Apply**. The `Display` block shows `3`, which corresponds to the default case.

Data Type Support

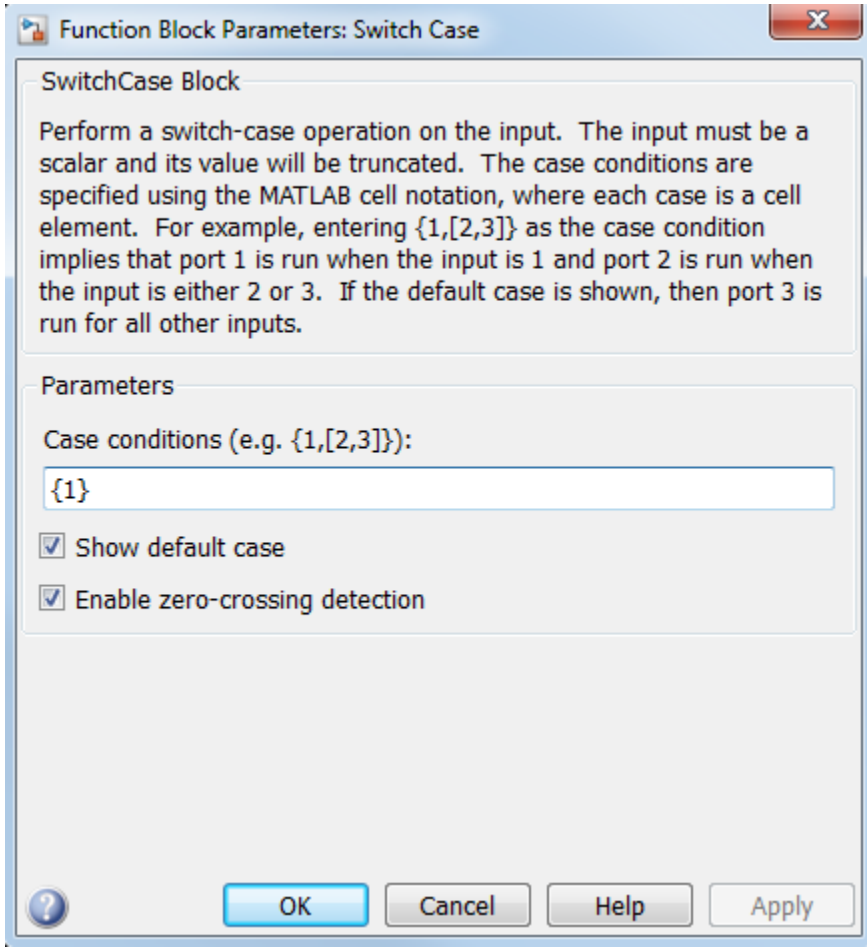
The input to the port labeled `u1` of a Switch Case block can be:

- A scalar value having a built-in data type that Simulink supports. The block does not support Boolean or fixed-point data types and truncates the numeric inputs to 32-bit signed integers.

- A scalar value of any enumerated data type, as described in "Control Algorithm Execution Using Enumerated Signal" on page 1-2006.

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

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

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

Characteristics

Data Types	Double | Single | Base Integer | Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No

Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

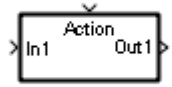
Introduced before R2006a

Switch Case Action Subsystem

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 “Create an Action Subsystem”, `Switch Case` block and “Use Control Flow Logic” in the “Creating a Model” chapter of the Simulink documentation.

Characteristics

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

Introduced before R2006a

Synchronous Subsystem

Represent subsystem that has synchronous reset and enable behavior

Library

HDL Coder / HDL Subsystems

Description 

A `Synchronous Subsystem` is a subsystem that uses the `Synchronous` mode of the `State Control` block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create a `Synchronous Subsystem`, add the block to your Simulink model from the HDL Subsystems block library. You can also add a `State Control` block with **State control** set to `Synchronous` inside a subsystem. For more information about the `State Control` block, see `State Control`.

Data Type Support

See `Inport` for information on the data types accepted by a subsystem's input ports. See `Outport` for information on the data types output by a subsystem's output ports.

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

Parameters

Show port labels

Cause Simulink software to display labels for the subsystem's ports on the subsystem's icon.

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 6-101 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 6-101 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 6-101 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 6-101 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 1-1911.
- "Sample time (-1 for inherited)" on page 1-1914
- "Function packaging" on page 1-1916 (requires a Simulink Coder license)

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from `Variant Source` blocks or to `Variant Sink` blocks.

Settings

Default: On

 On

Simulink treats the subsystem as a unit when propagating variant conditions from `Variant Source` blocks or to `Variant Sink` blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

 Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

Dependency

"Treat as grouped when propagating variant conditions" on page 1-1910 enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 1-1925 parameter setting. You can name the generated function and file using parameters "Function name" on page 1-1920 and "File name (no extension)" on page 1-1923. 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.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Characteristics

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

See Also

`Enabled Synchronous Subsystem` | `State Control`

Introduced in R2016a

Tapped Delay

Delay scalar signal multiple sample periods and output all delayed versions

Library

Discrete

Description

The Tapped Delay block delays an input by the specified number of sample periods and outputs all the delayed versions. Use this block to discretize a signal in time or resample a signal at a different rate.

The block accepts one scalar input and generates an output vector that contains each delay. Specify the order of the delays in the output vector with the **Order output vector starting with** parameter:

- `Oldest` orders the output vector starting with the oldest delay version and ending with the newest delay version.
- `Newest` orders the output vector starting with the newest delay version and ending with the oldest delay version.

Specify the output vector for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior.

Specify the time between samples with the **Sample time** parameter. Specify the number of delays with the **Number of delays** parameter. A value of `-1` instructs the block to inherit the number of delays by back propagation. Each delay is equivalent to the z^{-1} discrete-time operator, which the `Unit Delay` block represents.

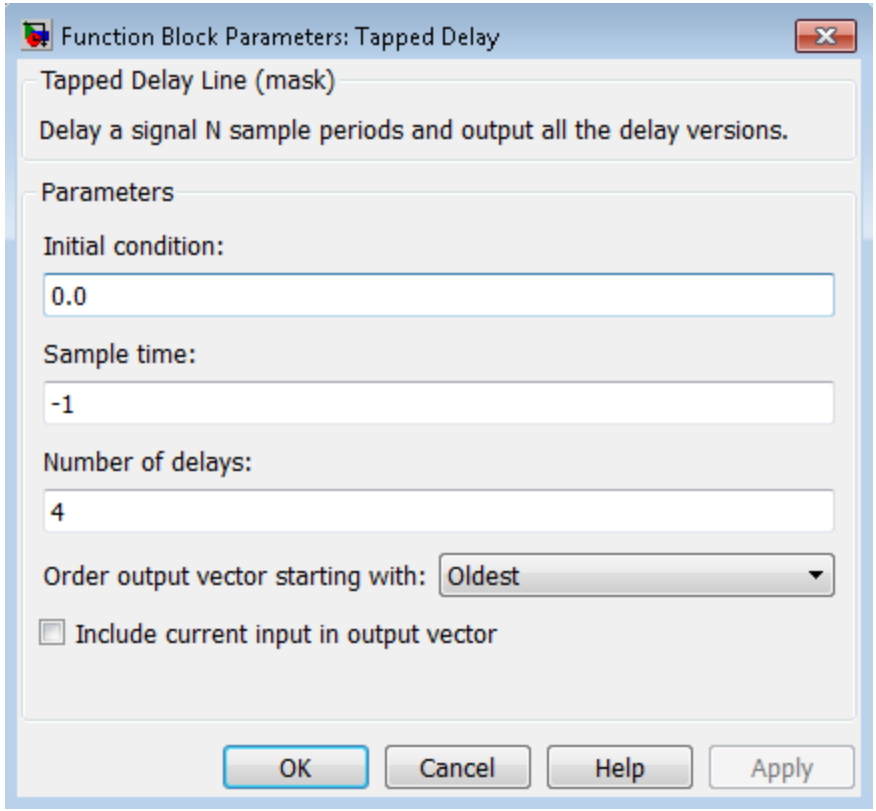
Data Type Support

The Tapped Delay block accepts signals of the following data types:

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

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Initial condition

Specify the initial output of the simulation. The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation. Simulink software does not allow you to set the initial condition of this block to `inf` or `NaN`.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the online Simulink documentation for more information.

Number of delays

Specify the number of discrete-time operators.

Order output vector starting with

Specify whether to output the oldest delay version first, or the newest delay version first.

Include current input in output vector

Select to include the current input in the output vector.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, when **Include current input in output vector** check box is selected. No, otherwise.
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

`Delay` | `Resettable Delay` | `Unit Delay` | `Variable Integer Delay`

Introduced before R2006a

Terminator

Terminate unconnected output port

Library

Sinks

Description

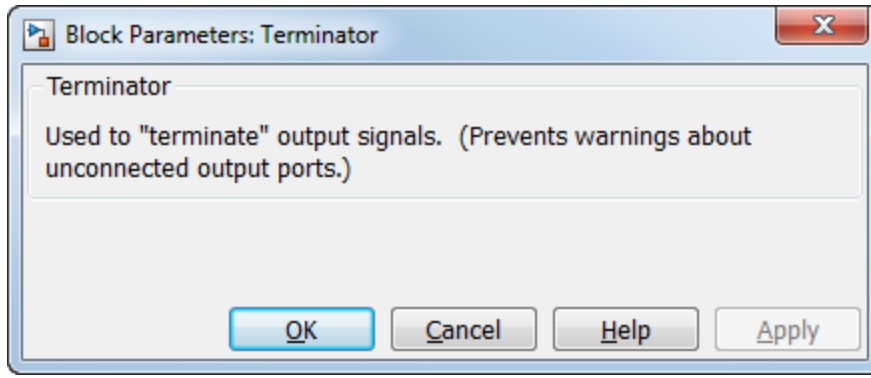
Use the Terminator block to cap blocks whose output ports do not connect to other blocks. If you run a simulation with blocks having unconnected output ports, Simulink issues warning messages. Using Terminator blocks to cap those blocks helps prevent warning messages.

Data Type Support

The Terminator block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Examples

The following Simulink examples show how to use the Terminator block:

- `sldemo_bounce`
- `sldemo_fuelsys`
- `aero_six_dof`

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Timed-Based Linearization

Generate linear models in base workspace at specific times

Library

Model-Wide Utilities

Description

This block calls `linmod` or `dlinmod` to create a linear model for the system when the simulation clock reaches the time specified by the **Linearization time** parameter. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.

The block sets the following model parameters to the indicated values:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`

The name of the structure used to save the snapshots is the name of the model appended by `_Timed_Based_Linearization`, for example, `vdp_Timed_Based_Linearization`. The structure has the following fields:

Field	Description
a	The A matrix of the linearization
b	The B matrix of the linearization
c	The C matrix of the linearization
d	The D matrix of the linearization

Field	Description
`StateName`	Names of the model's states
`OutputName`	Names of the model's output ports
`InputName`	Names of the model's input ports
`OperPoint`	A structure that specifies the operating point of the linearization. The structure specifies the operating point time (`OperPoint.t`). The states (`OperPoint.x`) and inputs (`OperPoint.u`) fields are not used.
`Ts`	The sample time of the linearization for a discrete linearization

Use the `Trigger-Based Linearization` block if you need to generate linear models conditionally.

You can use state and simulation time logging to extract the model states and inputs at operating points. For example, suppose that you want to get the states of the f14 example model at linearization times of 2 seconds and 5 seconds.

1. Open the model and drag an instance of this block from the Model-Wide Utilities library and drop the instance into the model.
2. Open the block's parameter dialog box and set the **Linearization time** to 2 and 5.
3. Open the model's **Model Configuration Parameters** dialog box.
4. Select the **Data Import/Export** pane.
5. Check **States** and **Time** on the **Save to Workspace** control panel
6. Select OK to confirm the selections and close the dialog box.
7. Simulate the model.

   At the end of the simulation, the following variables appear in the MATLAB workspace: `f14_Timed_Based_Linearization`, `tout`, and `xout`.
8. Get the indices to the operating point times by entering the following at the MATLAB command line:

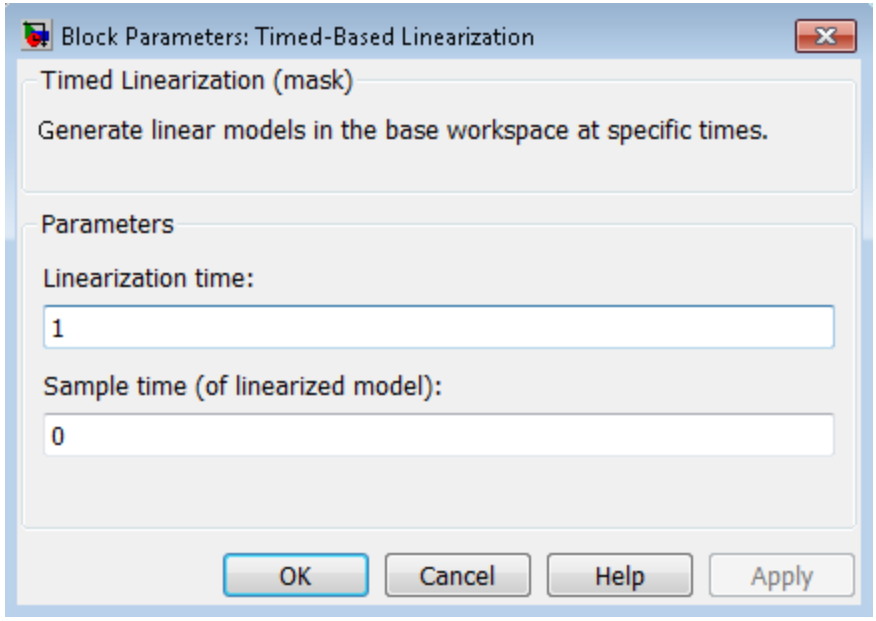
   ```
   ind1 = find(f14_Timed_Based_Linearization(1).OperPoint.t==tout);
   ind2 = find(f14_Timed_Based_Linearization(1).OperPoint.t==tout);
   ```
9. Get the state vectors at the operating points.

   ```
   x1 = xout(ind1,:);
   x2 = xout(ind2,:);
   ```

Data Type Support

Not applicable.

Parameters 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 2-34).

Characteristics

Data Types	Not applicable

Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	No

See Also

`Trigger-Based Linearization`

Introduced in R2010a

To File

Write data to file

Library

Sinks

Description

The `To File` block inputs a signal and writes the signal data into a MAT-file. Use the `To File` block to log signal data.

The `To File` block icon shows the name of the output file.

The block writes to the output file incrementally, with minimal memory overhead during simulation. If the output file exists when the simulation starts, the block overwrites the file. The file automatically closes when simulation is complete or paused. If simulation terminates abnormally, the `To File` block saves the data it has logged up until the point of the abnormal termination.

Tip If MATLAB encounters memory issues when you log a large number of signals in a long simulation that has many time steps, consider logging to persistent storage. When you log to persistent storage, the `Dataset` format logging data is stored in a MAT-file. Compared to logging to persistent storage, connecting a To File block to signals:

- Is a per-signal approach that can clutter a model with several `To File` blocks attached to individual signals.
- Creates a separate MAT-file for each To File block, compared to the one MAT-file that logging to persistent storage uses.

For details, see “Log Data Using Persistent Storage”.

Specifying the Format for Writing Data

Use the **Save format** parameter to specify the format for writing data:

- `Timeseries` (default)
- `Array`

Use the `Array` format only for vector, double, noncomplex signals. To save bus data, use the `Timeseries` format.

For the `Timeseries` format, the `To File` block:

- Writes data in a MATLAB `timeseries` object
- Supports writing multidimensional, real or complex output values
- Supports writing output values that have any built-in data type, including `Boolean`, enumerated (`enum`), and fixed-point data with a word length of up to 32 bits
- For bus input signals, creates a MATLAB structure that matches the bus hierarchy. Each leaf of the structure is a MATLAB `timeseries` object.

For the `Array` format, the `To File` block:

- Writes data into a matrix containing two or more rows. The matrix has the following form:

  $$\begin{bmatrix} t_1 & t_2 & \dots & t_{final} \\ u1_1 & u1_2 & \dots & u1_{final} \\ \dots & & & \\ un_1 & un_2 & \dots & un_{final} \end{bmatrix}$$

  Simulink writes one column to the matrix for each data sample. The first element of the column contains the time stamp. The remainder of the column contains data for the corresponding output values.
- Supports writing data that is one-dimensional, double, and noncomplex.

The following table shows how simulation mode support depends on the **Save format** value.

Simulation Mode	`Timeseries`	`Array`
Normal	Supported.	Supported.

Simulation Mode	`Timeseries`	`Array`
Accelerator	Supported.	Supported.
Rapid Accelerator	Supported.	Supported.
Software-in-the-Loop (SIL)	Not supported.	Supported if MAT-file logging is enabled.
Processor-in-the-Loop (PIL)	Not supported.	Supported if MAT-file logging is available and enabled.
External	Not supported.	Supported if MAT-file logging is enabled.
RSim target	Supported.	Supported if MAT-file logging is enabled.

Controlling When Data Is Written to the File

The `To File` block **Decimation** and **Sample Time** parameters control when data is written to the file.

The To File block does not log data outside of the intervals specified by the **Model Configuration Parameters > Data Import/Export > Logging intervals** parameter. The block stores the logged data in the file associated with the block instead of storing the data in the variable that you specify for the **Single simulation output** parameter.

Saving Data for Use by a From File Block

The `From File` block can use data written by a `To File` block in any format (`Timeseries` or `Array`) without any modifications to the data or other special provisions.

Saving Data for Use by a From Workspace Block

The `From Workspace` block can read data that is in the `Array` format and is the transposition of the data written by the `To File` block. To provide the required format, use MATLAB commands to load and transpose the data from the MAT-file.

Simulation Stepper Interaction with To File Block

If you pause using the Simulation Stepper, the `To File` block captures the simulation data up to the point of the pause. When you step back, the `To File` data file no longer contains any simulation data past the new reduced time of the last output.

Limitations of To File blocks in a Referenced Model

When a `To File` block is in a referenced model, that model must be a single-instance model. Only one instance of such a model can exist in a model hierarchy. See "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 copy of the file.

Generating Code

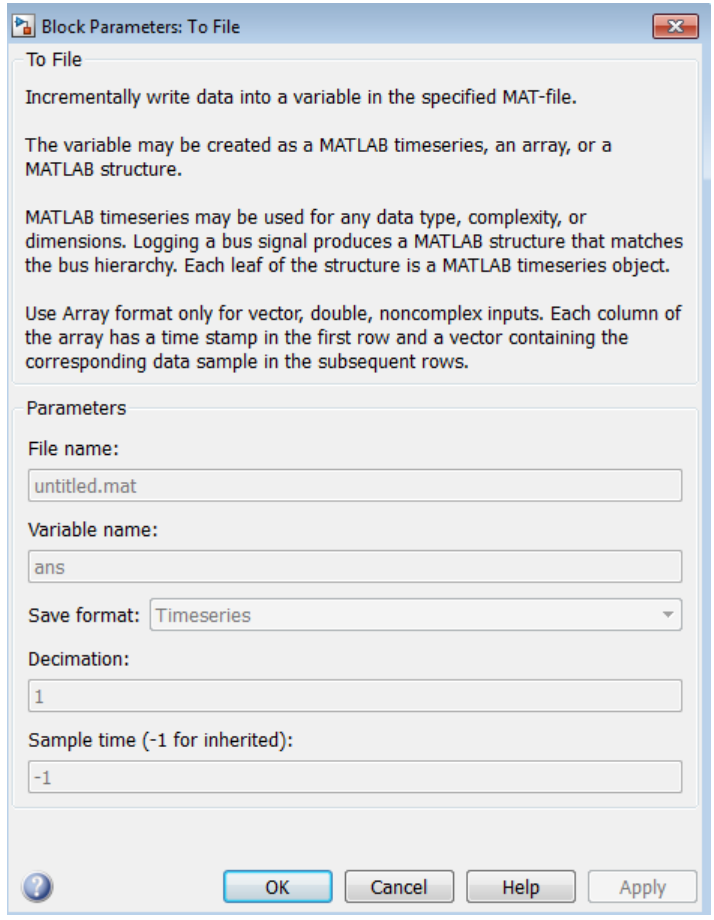
To generate code for a `To File` block, on the **Code Generation** > **Interface** pane, you must select the configuration parameter "MAT-file logging".

Data Type Support

The `To File` block accepts real or complex signal data of any data type that Simulink supports, with the exception that the word length for fixed-point data must be 32 bits or less.

The `To File` block accepts bus data.

Parameters 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 that the `To File` block uses for writing data:

- `Timeseries` (default)
- `Array`

Decimation

The decimation factor, `n`, where `n` specifies writing data at every `n`th time that the block executes. The default decimation is `1`, which writes data at every time step.

Sample time

Specifies the sample period and offset at which to collect points. This parameter is useful when you are using a variable-step solver where the interval between time steps might not be constant. The default is `-1`, which inherits the sample time from the driving block. See “Specify Sample Time” for more information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus

Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

See Also

"Save Runtime Data from Simulation", "Convert Logged Data to Dataset Format", `From File`, `From Workspace`, `To Workspace`

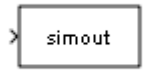
Introduced before R2006a

To Workspace

Write data to workspace

Library

Sinks



Description

The To Workspace block inputs a signal and writes the signal data to a workspace. During the simulation, the block writes data to an internal buffer. When the simulation is completed or paused, that data is written to the workspace. Data is not available until the simulation is stopped or paused.

- For menu-based simulation, data is written in the MATLAB base workspace.
- A `sim` command in a MATLAB function sends data logged with the To Workspace block to the workspace of the calling function, not to the MATLAB (base) workspace. To send the logged data to the base workspace, use an `assignin` command in the function. For example:

```
function myfunc
    a = sim('mTest','SimulationMode','normal');
    b = a.get('simout')
    assignin('base','b',b);
end
```

The block icon shows the name of the variable to which the data is written. To specify the name of the workspace variable to which the To Workspace block writes the data, use the **Variable name** parameter.

To specify the data format of the variable, use the **Save format** parameter. You can specify to save the data in one of the following formats:

- A MATLAB timeseries object (or structure of MATLAB `timeseries` objects for bus data)
- An array
- Structure
- Structure with time

From one of these formats, you can convert the data to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format". You can also use signal logging with a variable-size signal exception.

Saving Data for Use by a From Workspace Block

To use a `From Workspace` block to read into Simulink the sample-based data that was saved by a To Workspace block in a previous simulation, in the To Workspace block, specify time information using the `Timeseries` or `Structure with time` format.

Controlling the Amount of Data Saved

If you specify intervals with the **Model Configuration Parameters > Data Import/Export > Logging intervals** parameter, the block does not log data outside of the intervals. For example, the block logs no data if the intervals are empty (`[]`).

For variable-step solvers, to control the amount of data available to the To Workspace block, use the **Model Configuration Parameters > Data Import/Export > Additional parameters > Output options** parameter. For example, to have Simulink write data at identical time points over multiple simulations, select the `Produce specified output only` option.

Then use To Workspace block parameters to control when and how much of this data the block writes:

- Use the **Limit data points to last** parameter to specify how many sample points to save. If the simulation generates more data points than the specified maximum, the simulation saves only the most recently generated samples. To capture all the data, set this value to `inf`.
- Use the **Decimation** parameter to have the To Workspace block write data at every `n`th sample, where `n` is the decimation factor. The default decimation, 1, writes data at every time hit.

- Use the **Sample time** parameter to specify a sampling interval at which to collect points. This parameter is useful when you are using a variable-step solver where the interval between time hits might not be the same. The default value of -1 causes the block to inherit the sample time from the driving block when determining the points to write. See "Specify Sample Time" in the online documentation for more information.

For example, suppose you have a simulation where the start time is 0, the **Limit data points to last** is 100, the **Decimation** is 1, and the **Sample time** is 0.5. The To Workspace block collects a maximum of 100 points, at time values of 0, 0.5, 1.0, 1.5, ..., seconds. Specifying a **Decimation** value of `1` directs the block to write data at each step.

In a similar example, the **Limit data points to last** is 100 and the **Sample time** is 0.5, but the **Decimation** is 5. In this example, the block collects up to 100 points, at time values of 0, 2.5, 5.0, 7.5, ..., seconds. Specifying a **Decimation** value of `5` directs the block to write data at every fifth sample. The sample time ensures that data is written at these points.

In another example, all parameters are as defined in the first example except that the **Limit data points to last** is 3. In this case, only the last three sample points collected are written to the workspace. If the simulation stop time is 100, data corresponds to times 99.0, 99.5, and 100.0 seconds (three points).

MAT-File Logging

When you enable the **Configuration Parameters > All Parameters > MAT-file logging** parameter, To Workspace logs its data to a MAT-file. For information about this parameter, in the Simulink Coder documentation, see "MAT-file logging".

Frame-Based Signals

By default, the To Workspace block treats input signals as sample-based.

To have the To Workspace block treat input signals as frame-based, set:

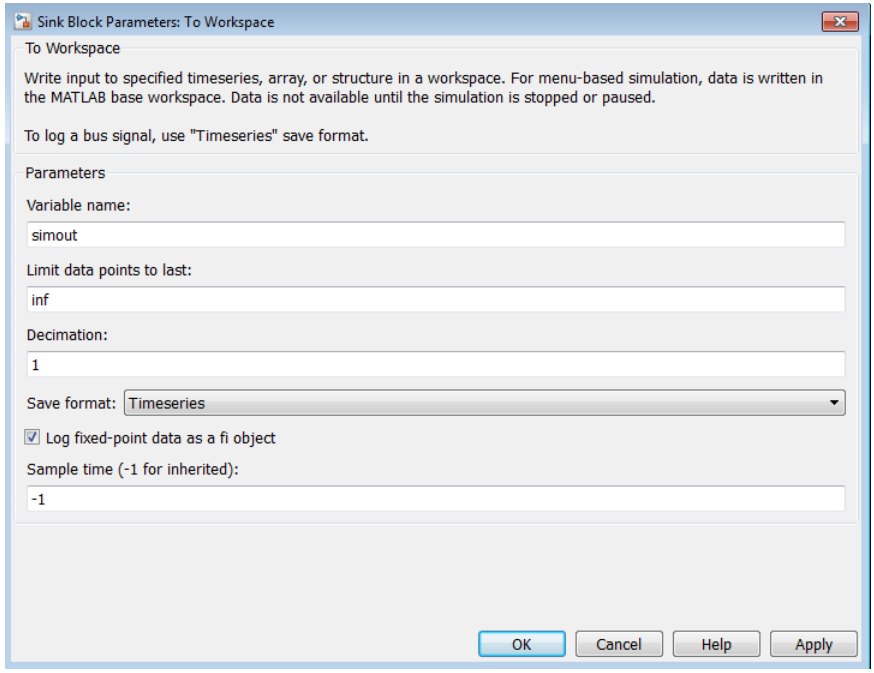
1. **Save format** to either `Array` or `Structure`
2. **Save 2-D signals as** to `2-D array (concatenate along first dimension)`

Data Type Support

The To Workspace block can save to the MATLAB workspace real or complex inputs of any data type that Simulink supports, including fixed-point and enumerated data types, as well as bus objects.

For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box



Variable name

Specify the name of the variable for the saved the data.

Limit data points to last

Specify the maximum number of input samples to save. The default is `inf`.

Decimation

Specify the decimation factor. The default is `1`.

Save format

Specify one of these formats for saving simulation output to the workspace:

- `Timeseries` (Default)

  Save non-bus signals as a MATLAB timeseries object and bus signals as a structure of MATLAB `timeseries` objects.

- `Array`

  Save the input as an N-dimensional array where N is one more than the number of dimensions of the input signal. For example, if the input signal is a vector, the resulting workspace array is two-dimensional. If the input signal is a matrix, then the array is three-dimensional.

  How Simulink stores samples in the array depends on whether the input signal is a scalar, vector, or matrix. If the input is a scalar or a vector, each input sample is output as a row of the array. For example, suppose that the name of the output array is `simout`. Then, `simout(1,:)` corresponds to the first sample, `simout(2,:)` corresponds to the second sample, and so on. If the input signal is a matrix, time corresponds to the third dimension. For example, suppose again that `simout` is the name of the resulting workspace array. Then, `simout(:,:,1)` is the value of the input signal at the first sample point; `simout(:,:,2)` is the value of the input signal at the second sample point; and so on.

  If you select `Array`, the **Save 2-D signals** as parameter appears.

  To treat input signals as frame-based, set **Save format** to either `Array` or `Structure` and set the **Save 2-D signals** parameter to `2-D array (concatenate along first dimension)`.

- `Structure`

  This format consists of a structure with three fields:

  - `time` — This field is empty for this format.

- `signals` — A structure with three fields: `values`, `dimensions`, and `label`. The `values` field contains the array of signal values. The `dimensions` field specifies the dimensions of the corresponding signals. The `label` field contains the label of the input line.
- `blockName` — Name of the To Workspace block

If you select `Structure`, the **Save 2-D signals** as parameter appears.

To treat input signals as frame-based, set **Save format** to either `Structure` or `Array` and set the **Save 2-D signals** parameter to `2-D array (concatenate along first dimension)`.

- `Structure With Time`

  This format is the same as `Structure`, except that the time field contains a vector of simulation time hits.

  To read To Workspace block output directly with a From Workspace block, use either the `Timeseries` or `Structure with Time` format. For details, see "Comparison of Signal Loading Techniques".

  `Structure with Time` format does not support frame-based signals. Use `Array` or `Structure` format instead.

The following table shows how simulation mode support depends on the **Save format** value.

Simulation Mode	`Timeseries`	`Array`, `Structure`, or `Structure With Time`
Normal	Supported.	Supported.
Accelerator	Supported.	Supported only in top model, not referenced models.
Rapid Accelerator	Not supported.	Supported only in top model, not referenced models.
Software-in-the-Loop (SIL)	Not supported.	If MAT-file logging is enabled, supported only in top model, not referenced models.

Simulation Mode	`Timeseries`	`Array`, `Structure`, or `Structure With Time`
Processor-in-the-Loop (PIL)	Not supported.	If MAT-file logging is available and enabled, supported only in top model, not referenced models.
External	Not supported.	Supported only in top model, not referenced models.
Simulink Coder Targets	Not supported.	If MAT-file logging is enabled, supported only in top model, not referenced models.

Save 2-D signal as

If you set **Save format** to `Array` or `Structure`, the **Save 2-D signals as** parameter appears.

Specify one of these formats for saving 2-D signals to the workspace:

- `3-D array (concatenate along third dimension)` (Default)

  This setting is well-suited for sample-based signals. Data is concatenated along the third dimension. For example, 2-by-4 matrix input for 10 samples is stored as a 2x4x10 array.

- `2-D array (concatenate along first dimension)`

  This setting is well-suited for frame-based signals. The data is concatenated along the first dimension. For example, 2-by-4 matrix input for 10 samples is stored as a 20x4 array

- `Inherit from input (this choice will be removed – see release notes)`

  This setting is for backward compatibility. To configure this block to treat input signals as frame-based in future releases, set this parameter to `2-D array (concatenate along first dimension)`. To configure this block to treat input signals as sample-based in future releases, set this parameter to `3-D array (concatenate along third dimension)`.

When the **Save format** is set to `Array` or `Structure`, the dimensions of the output depend on the input dimensions and the setting of the **Save 2-D signals as** parameter.

The following table summarizes the output dimensions under various conditions. In the table, *K* represents the value of the **Limit data points to last** parameter.

Input Signal Dimensions	Save 2-D Signals as ...	Signal To Workspace Output Dimension
M-by-*N* matrix	`2-D array (concatenate along first dimension)`	*K*-by-*N* matrix. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of samples acquired in each column by the end of simulation. This is equivalent to multiplying the input frame size (*M*) by the total number of *M*-by-*N* inputs acquired by the block.
M-by-*N* matrix	`3-D array (concatenate along third dimension)`	*M*-by-*N*-by-*K* array. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of *M*-by-*N* inputs acquired by the end of the simulation.
Length-*N* unoriented vector	Any setting	*K*-by-*N* matrix
N-dimensional array where $N > 2$	Any setting	Array with *N+1* dimensions, where the size of the last dimension is equal to *K*. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of *M*-by-*N* inputs acquired by the end of simulation

Log fixed-point data as a fi object

By default, the To Workspace block logs fixed-point data to the MATLAB workspace as a Fixed-Point Designer `fi` object. If you clear this parameter, fixed-point data is logged to the workspace as `double`.

Sample time

Specify the sample period and offset at which to collect data. This parameter is useful when you are using a variable-step solver where the interval between time hits might not be constant. The default is `-1`, which inherits the sample time from the driving block. See "Specify Sample Time" for more information.

Examples

The `sldemo_varsize_basic` example shows how to use the To Workspace block.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	No

See Also

"Export Simulation Data", "Convert Logged Data to Dataset Format", `From File`, `From Workspace`, `To File`

Introduced before R2006a

Toggle Switch

Set on/off values to tune parameters or variables

Library

Dashboard

Description

The `Toggle Switch` block enables you to control tunable parameters and variables in your model during simulation. The block has two states that can be set to two different values.

To control a tunable parameter or variable using the `Toggle Switch` block, double-click the `Toggle Switch` block to open the dialog box. Select a block in the model canvas. The tunable parameter or variable appears in the dialog box **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable to the block.

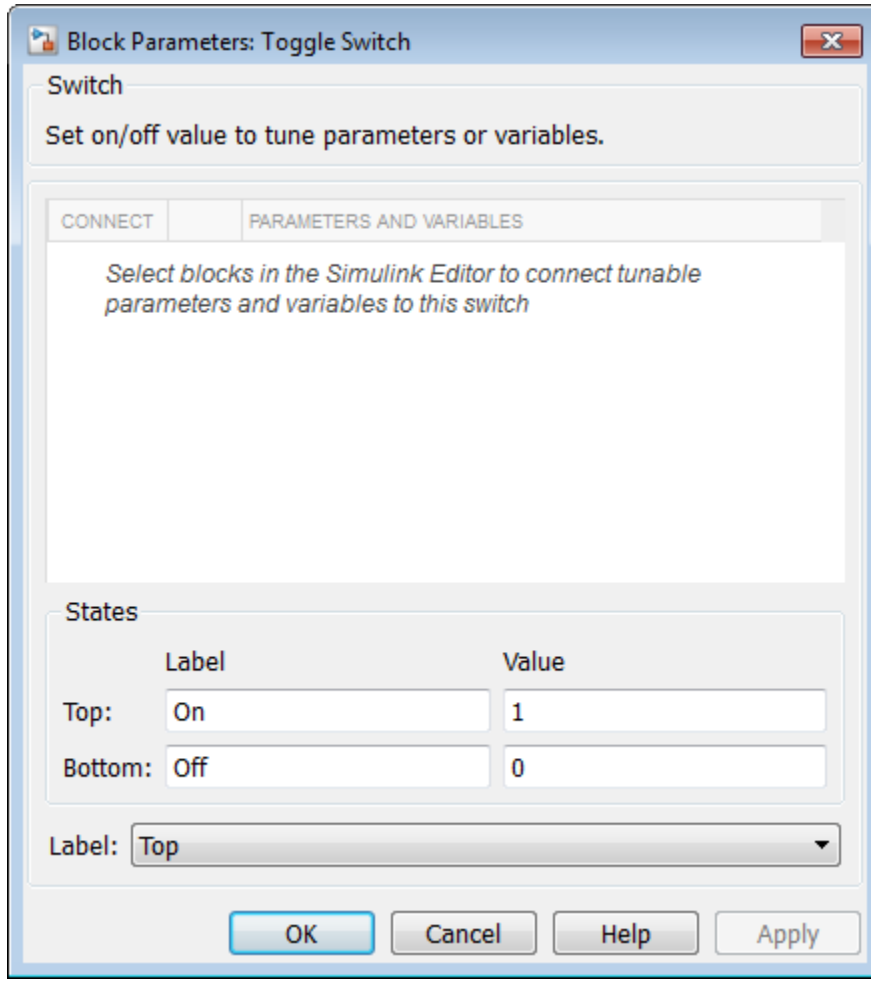
Limitations

The `Toggle Switch` block has these limitations, which you can work around.

Limitation	Workaround
You cannot save the block connections or properties in model files that use the `MDL` format.	Save the model file to `SLX` format to be able to save connections and properties.
Parameters that index a variable array do not appear in the **Connection** table.	For example, a block parameter specified using the variable `engine(1)` will not appear in the table because the parameter uses an index of the variable `engine`,

Limitation	Workaround
	which is not a scalar variable. To make the parameter appear in the **Connection** table, change the block parameter field to a scalar variable, such as `engine_1`.

Parameters and Dialog Box



Connection

Select a block to connect and control a tunable parameter or variable.

To control a tunable parameter or variable, select a block in the model. The tunable parameter or variable appears in the **Connection** table. Select the option button next to the tunable parameter or variable you want to control. Click **Apply** to connect the tunable parameter or variable.

Settings

The table has a row for the tunable parameter or variable connected to the block. If there are no tunable parameters or variables selected in the model or the block is not connected to any tunable parameters or variables, then the table is empty.

States

Switch values and labels.

Settings

Default Labels: `Off` and `On`

Default Values: `0` and `1`

By default, the **Off** state label corresponds to the set value of `0`, and the **On** state label corresponds to the set value of `1`.

The state labels appear on the outside of the switch. You can change the state labels to another text string. You can change the state values to any real value that is between negative `realmax` and positive `realmax`.

Label

Position of the block label or instructional text if the block is not connected.

Settings

Default: Top

Top

Show the label at the top of the block.

Bottom

Show the label at the bottom of the block.

Hide

Do not show the label or instructional text when the block is not connected.

Examples

For more information on using blocks from the Dashboard library, see “Tune and Visualize Your Model with Dashboard Blocks”.

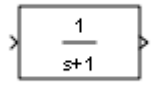
Introduced in R2015a

Transfer Fcn

Model linear system by transfer function

Library

Continuous



Description

The Transfer Fcn block models a linear system by a transfer function of the Laplace-domain variable `s`. The block can model single-input single-output (SISO) and single-input multiple output (SIMO) systems.

Conditions for Using This Block

The Transfer Fcn block assumes the following conditions:

- The transfer function has the form

$$H(s) = \frac{y(s)}{u(s)} = \frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + num(2)s^{nn-2} + \ldots + num(nn)}{den(1)s^{nd-1} + den(2)s^{nd-2} + \ldots + den(nd)},$$

  where u and y are the system input and outputs, respectively, nn and nd are the number of numerator and denominator coefficients, respectively. *num(s)* and *den(s)* contain the coefficients of the numerator and denominator in descending powers of s.
- The order of the denominator must be greater than or equal to the order of the numerator.
- For a multiple-output system, all transfer functions have the same denominator and all numerators have the same order.

Modeling a Single-Output System

For a single-output system, the input and output of the block are scalar time-domain signals. To model this system:

1. Enter a vector for the numerator coefficients of the transfer function in the **Numerator coefficients** field.
2. Enter a vector for the denominator coefficients of the transfer function in the **Denominator coefficients** field.

Modeling a Multiple-Output System

For a multiple-output system, the block input is a scalar and the output is a vector, where each element is an output of the system. To model this system:

1. Enter a matrix in the **Numerator coefficients** field.

   Each *row* of this matrix contains the numerator coefficients of a transfer function that determines one of the block outputs.
2. Enter a vector of the denominator coefficients common to all transfer functions of the system in the **Denominator coefficients** field.

Specifying Initial Conditions

A transfer function describes the relationship between input and output in Laplace (frequency) domain. Specifically, it is defined as the Laplace transform of the response (output) of a system with zero initial conditions to an impulse input.

Operations like multiplication and division of transfer functions rely on zero initial state. For example, you can decompose a single complicated transfer function into a series of simpler transfer functions. Apply them sequentially to get a response equivalent to that of the original transfer function. This will not be correct if one of the transfer functions assumes a non-zero initial state. Furthermore, a transfer function has infinitely many time domain realizations, most of whose states do not have any physical meaning.

For these reasons, Simulink presets the initial conditions of the Transfer Fcn block to zero. To specify initial conditions for a given transfer function, convert the transfer function to its controllable, canonical state-space realization using `tf2ss` . Then, use the State-Space block. The `tf2ss` utility provides the A, B, C, and D matrices for the system.

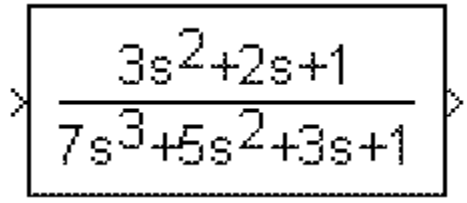
For more information, type `help tf2ss` or see the Control System Toolbox™ documentation.

Transfer Function Display on the Block

The Transfer Fcn block displays the transfer function depending on how you specify the numerator and denominator parameters.

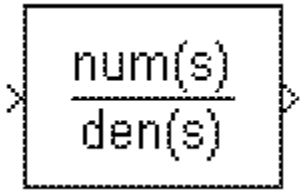
- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified coefficients and powers of *s*. If you specify a variable in parentheses, the block evaluates the variable.

  For example, if you specify **Numerator coefficients** as `[3,2,1]` and **Denominator coefficients** as `(den)`, where `den` is `[7,5,3,1]`, the block looks like this:



- If you specify each parameter as a variable, the block shows the variable name followed by `(s)`.

  For example, if you specify **Numerator coefficients** as `num` and **Denominator coefficients** as `den`, the block looks like this:

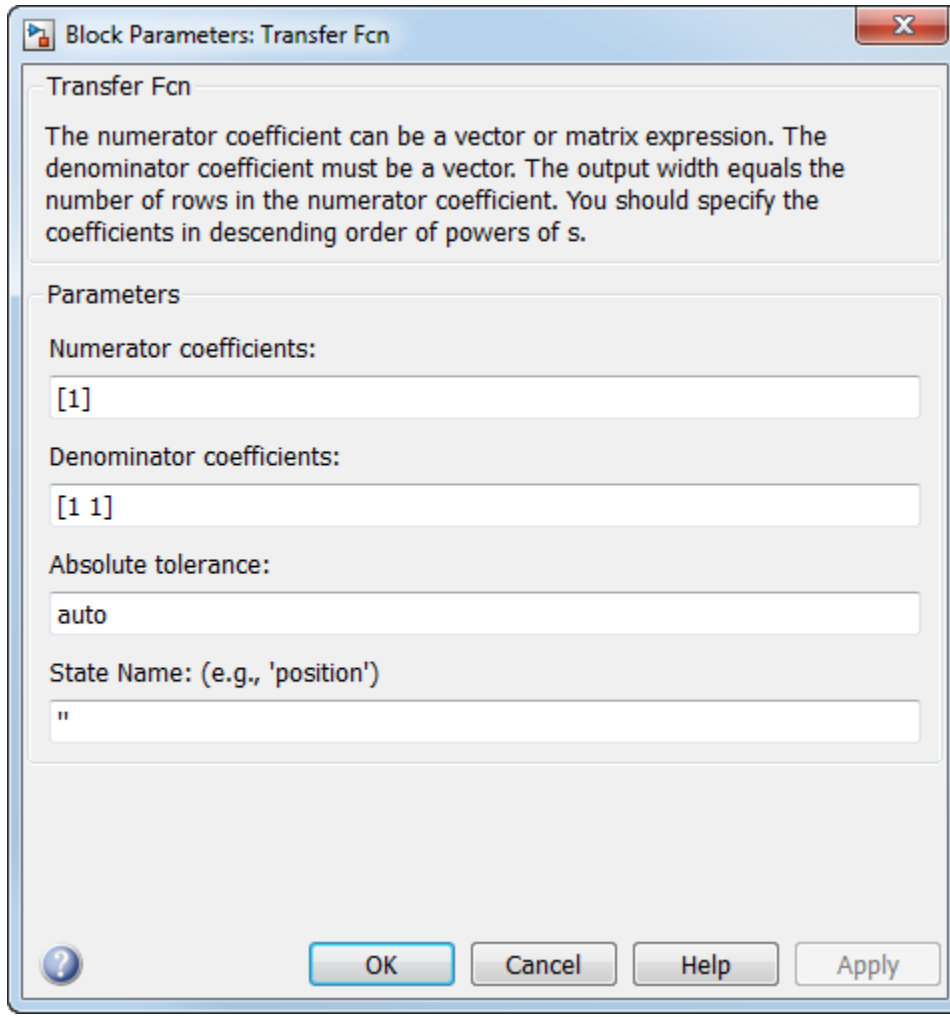


Data Type Support

The Transfer Fcn block accepts and outputs signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters 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

Parameter: `Numerator`
Type: vector or matrix
Value: `'[1]'`
Default: `'[1]'`

See Also

See the `Transfer Fcn` block reference page for more 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

Parameter: `Denominator`
Type: vector
Value: `'[1 1]'`
Default: `'[1 1]'`

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, –1, a positive real scalar or vector.
- If you enter `auto` or –1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Command-Line Information

Parameter: `AbsoluteTolerance`
Type: string, scalar, or vector
Value: `'auto' | '-1' |` any positive real scalar or vector
Default: `'auto'`

State Name (e.g., 'position')

Assign a unique name to each state.

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

Parameter: `ContinuousStateAttributes`
Type: string
Value: `' '` | user-defined
Default: `' '`

Examples

The following Simulink examples show how to use the `Transfer Fcn` block:

- `slexAircraftExample`
- `sldemo_absbrake`

- `penddemo`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Only if the lengths of the **Numerator coefficients** and **Denominator coefficients** parameters are equal
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Discrete Transfer Fcn`

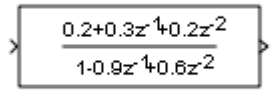
Introduced before R2006a

Transfer Fcn Direct Form II

Implement Direct Form II realization of transfer function

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.

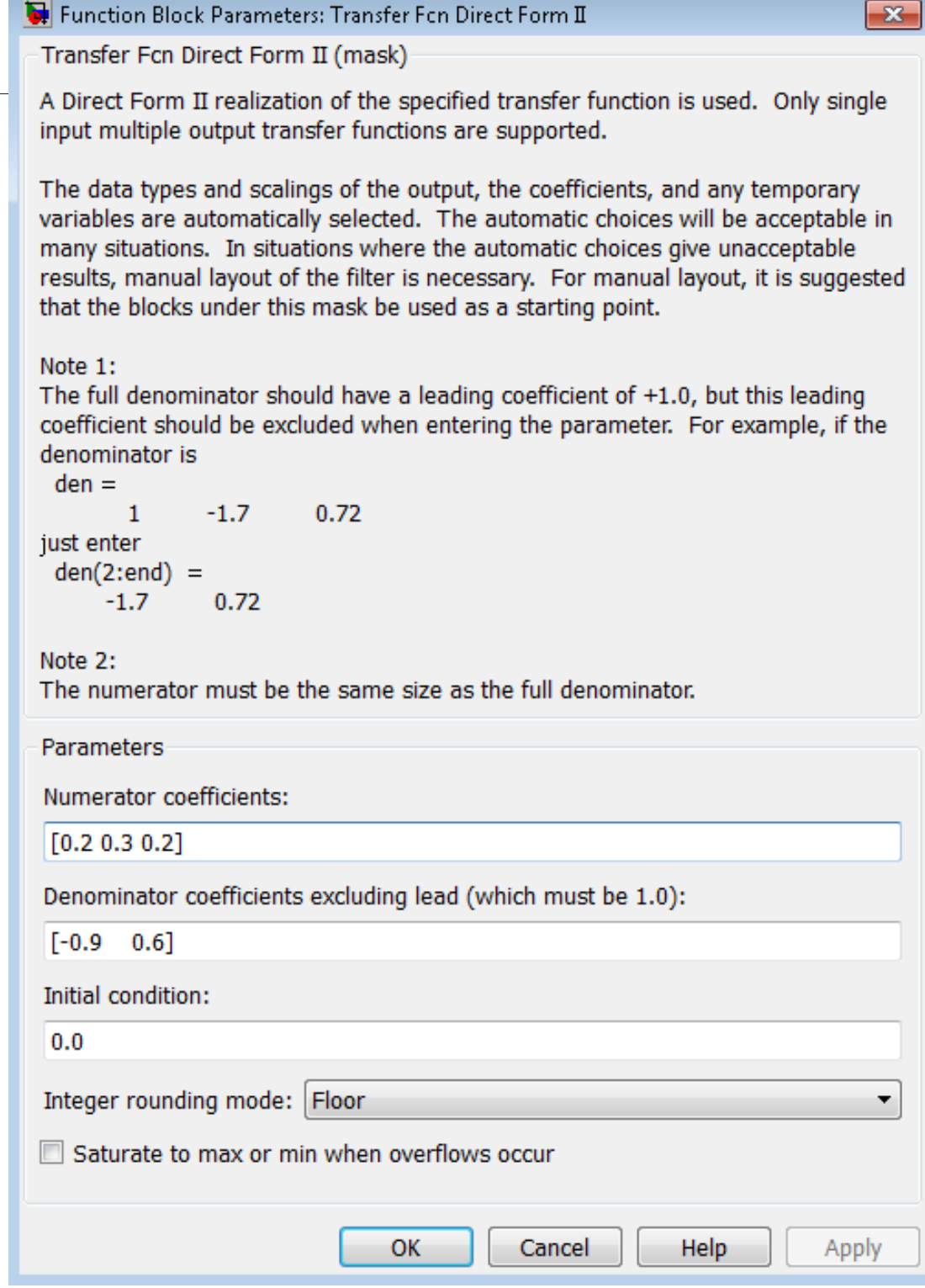
The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

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.


Function Block Parameters: Transfer Fcn Direct Form II
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
OK
Cancel
Help
Apply

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

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Transfer Fcn Direct Form II Time Varying`

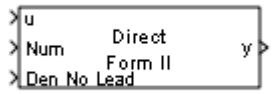
Introduced before R2006a

Transfer Fcn Direct Form II Time Varying

Implement time varying Direct Form II realization of transfer function

Library

Additional Math & Discrete / Additional Discrete



Description

The Transfer Fcn Direct Form II Time Varying block implements a Direct Form II realization of the specified transfer function. The block supports only single input-single output (SISO) transfer functions.

The input signal labeled `Den No Lead` contains the denominator coefficients of the transfer function. The full denominator has a leading coefficient of one, but it is excluded from the input signal. For example, a denominator of [1 -1.7 0.72] is represented by a signal with the value [-1.7 0.72]. The input signal labeled `Num` contains the numerator coefficients. The data types of the numerator and denominator coefficients can be different, but the length of the numerator vector and the full denominator vector must be the same. Pad the numerator vector with zeros, if needed.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

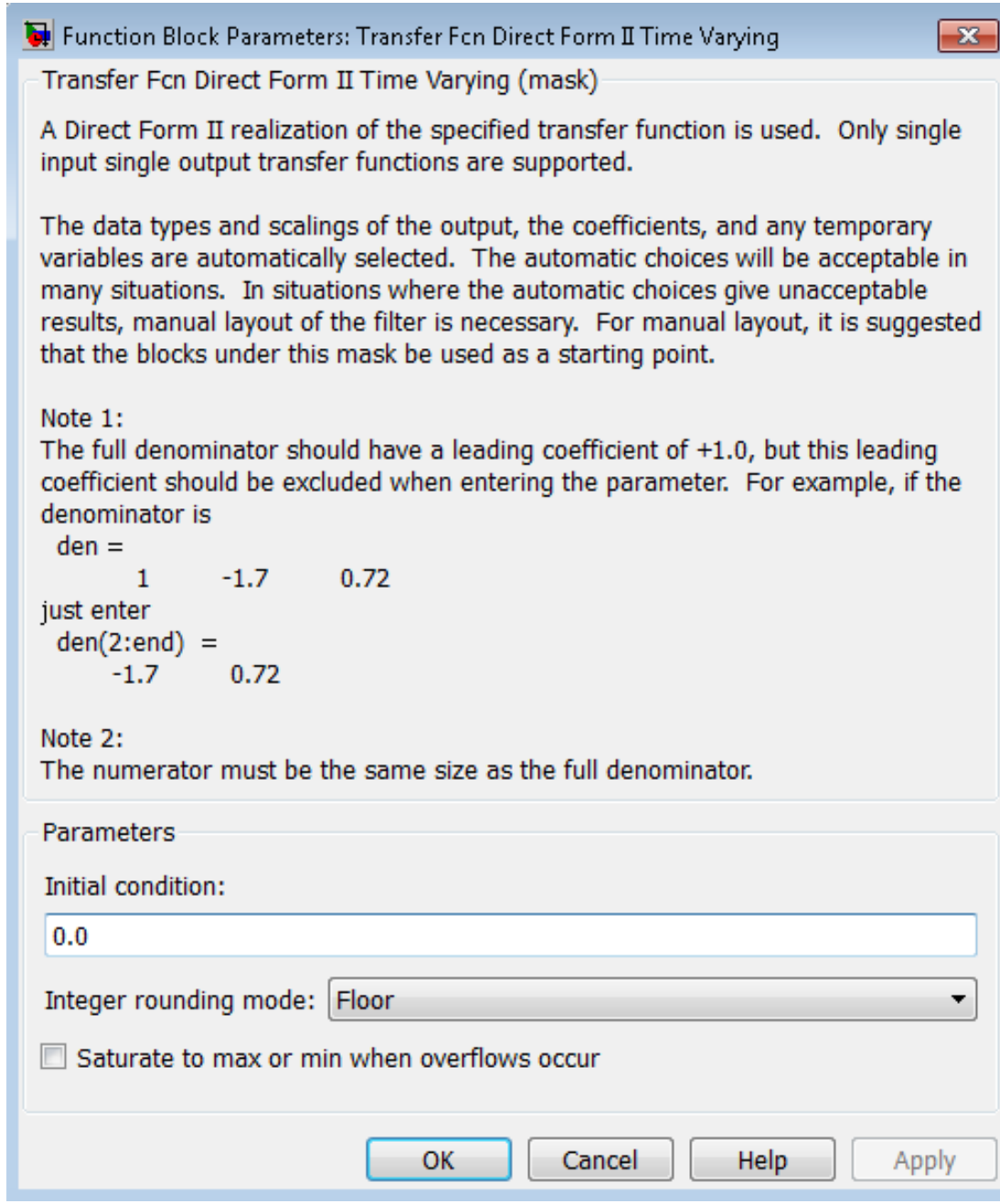
Data Type Support

The Transfer Fcn Direct Form II Time Varying block accepts signals of the following data types:

- Floating point

- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.


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
OK
Cancel
Help
Apply

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Transfer Fcn Direct Form II`

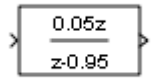
Introduced before R2006a

Transfer Fcn First Order

Implement discrete-time first order transfer function

Library

Discrete



Description

The Transfer Fcn First Order block implements a discrete-time first order transfer function of the input. The transfer function has a unity DC gain.

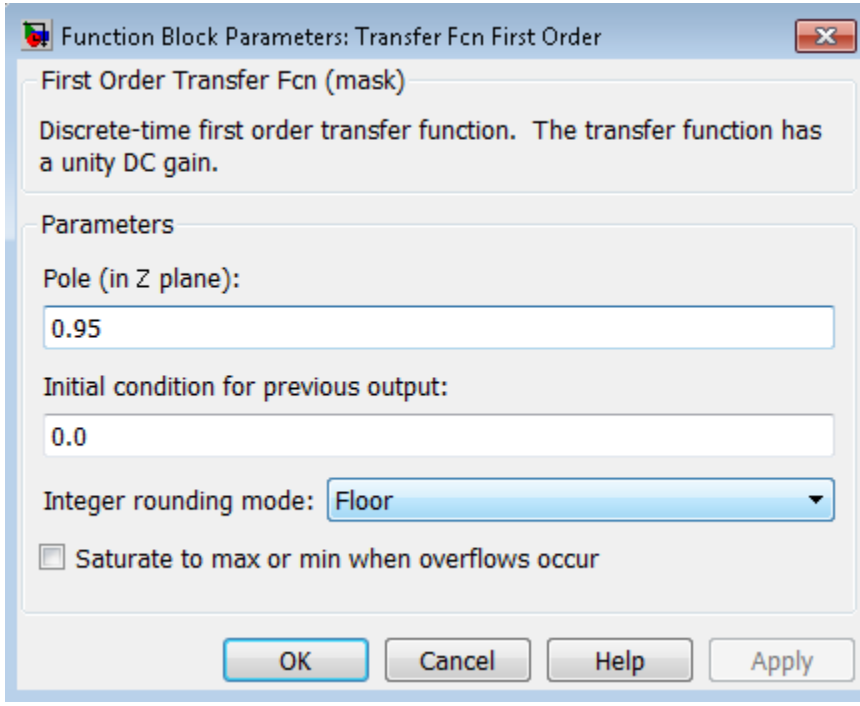
Data Type Support

The Transfer Fcn First Order block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters 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

Data Types	Double | Single | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

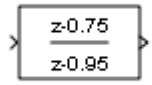
Introduced before R2006a

Transfer Fcn Lead or Lag

Implement discrete-time lead or lag compensator

Library

Discrete



Description

The Transfer Fcn Lead or Lag block implements a discrete-time lead or lag compensator of the input. The instantaneous gain of the compensator is one, and the DC gain is equal to (1-z)/(1-p), where z is the zero and p is the pole of the compensator.

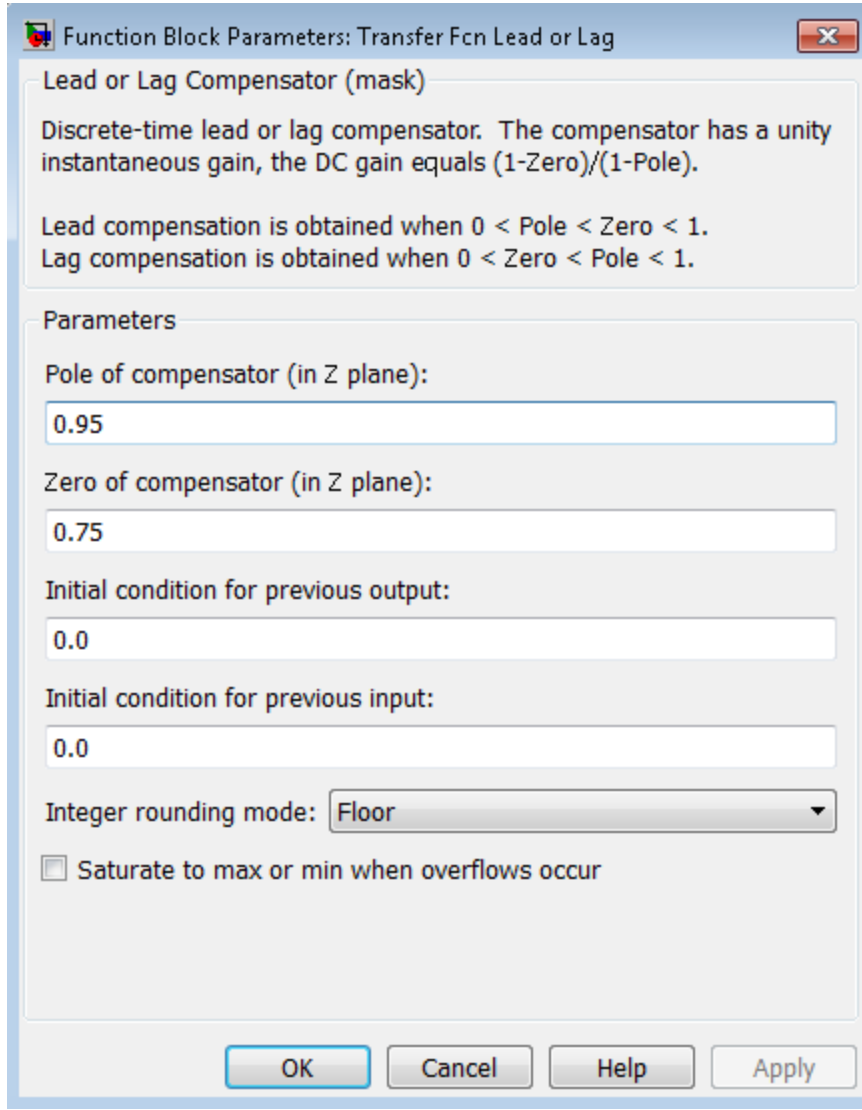
The block implements a lead compensator when $0 < z < p < 1$, and implements a lag compensator when $0 < p < z < 1$.

Data Type Support

The Transfer Fcn Lead or Lag block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters 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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

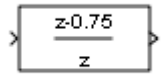
Introduced before R2006a

Transfer Fcn Real Zero

Implement discrete-time transfer function that has real zero and no pole

Library

Discrete



Description

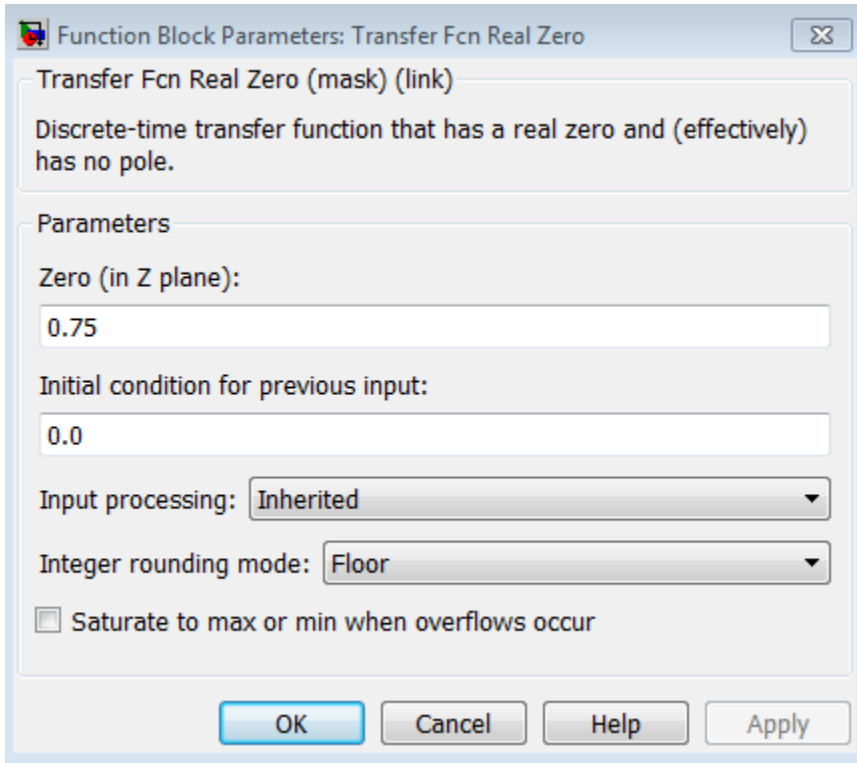
The Transfer Fcn Real Zero block implements a discrete-time transfer function that has a real zero and effectively no pole.

Data Type Support

The Transfer Fcn Real Zero block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters 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
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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Transport Delay

Delay input by given amount of time

Library

Continuous

Description

The Transport Delay block delays the input by a specified amount of time. You can use this block to simulate a time delay. The input to this block should be a continuous signal.

At the start of simulation, the block outputs the **Initial output** parameter until the simulation time exceeds the **Time delay** parameter. Then, the block begins generating the delayed input. During simulation, the block stores input points and simulation times in a buffer. You specify this size with the **Initial buffer size** parameter.

When you want output at a time that does not correspond to times of the stored input values, the block interpolates linearly between points. When the delay is smaller than the step size, the block extrapolates from the last output point, which can produce inaccurate results. Because the block does not have direct feedthrough, it cannot use the current input to calculate an output value. For example, consider a fixed-step simulation with a step size of 1 and the current time at t = 5. If the delay is 0.5, the block must generate a point at t = 4.5. Because the most recent stored time value is at t = 4, the block performs forward extrapolation.

The Transport Delay block does not interpolate discrete signals. Instead, the block returns the discrete value at the required time.

This block differs from the Unit Delay block, which delays and holds the output on sample hits only.

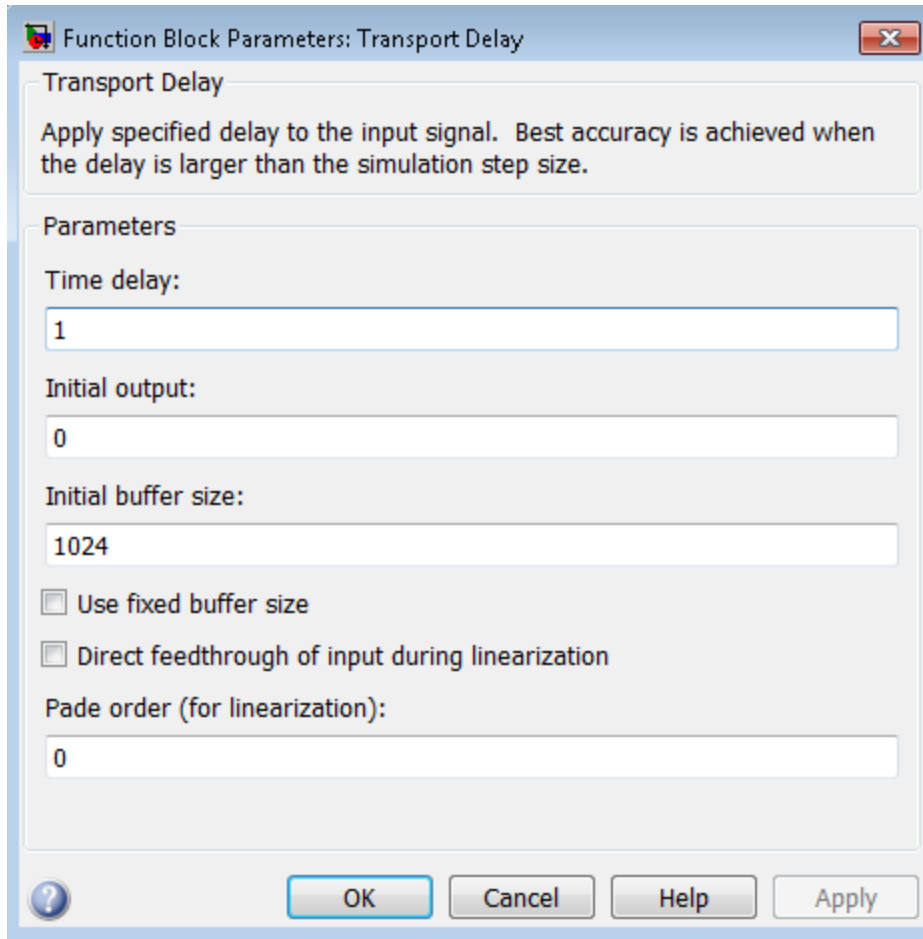
Tip Avoid using `linmod` to linearize a model that contains a Transport Delay block. For more information, see "Linearizing Models" in the Simulink documentation.

Data Type Support

The Transport Delay block accepts and outputs real signals of type `double`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters 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

Parameter: `DelayTime`
Type: scalar or vector
Value: `'1'`
Default: `'1'`

Initial output

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Settings

Default: `Run-to-run tunable parameter`

A `Run-to-run tunable parameter` cannot be changed during a simulation's run time. However, changing it before a simulation begins will not cause Accelerator or Rapid Accelerator to regenerate code.

Also, the initial output of this block cannot be `inf` or `NaN`.

Command-Line Information

Parameter: `InitialOutput`
Type: scalar or vector
Value: `'0'`
Default: `'0'`

Initial buffer size

Define the initial memory allocation for the number of input points to store.

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

Parameter: `BufferSize`
Type: scalar
Value: `'1024'`
Default: `'1024'`

Use fixed buffer size

Specify use of a fixed-size buffer to save input data from previous time steps.

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

Parameter: `FixedBuffer`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Direct feedthrough of input during linearization

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

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

Parameter: `TransDelayFeedthrough`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `PadeOrder`
Type: string
Value: `'0'`
Default: `'0'`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Variable Time Delay`

Introduced before R2006a

Trigger

Add trigger port to 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:

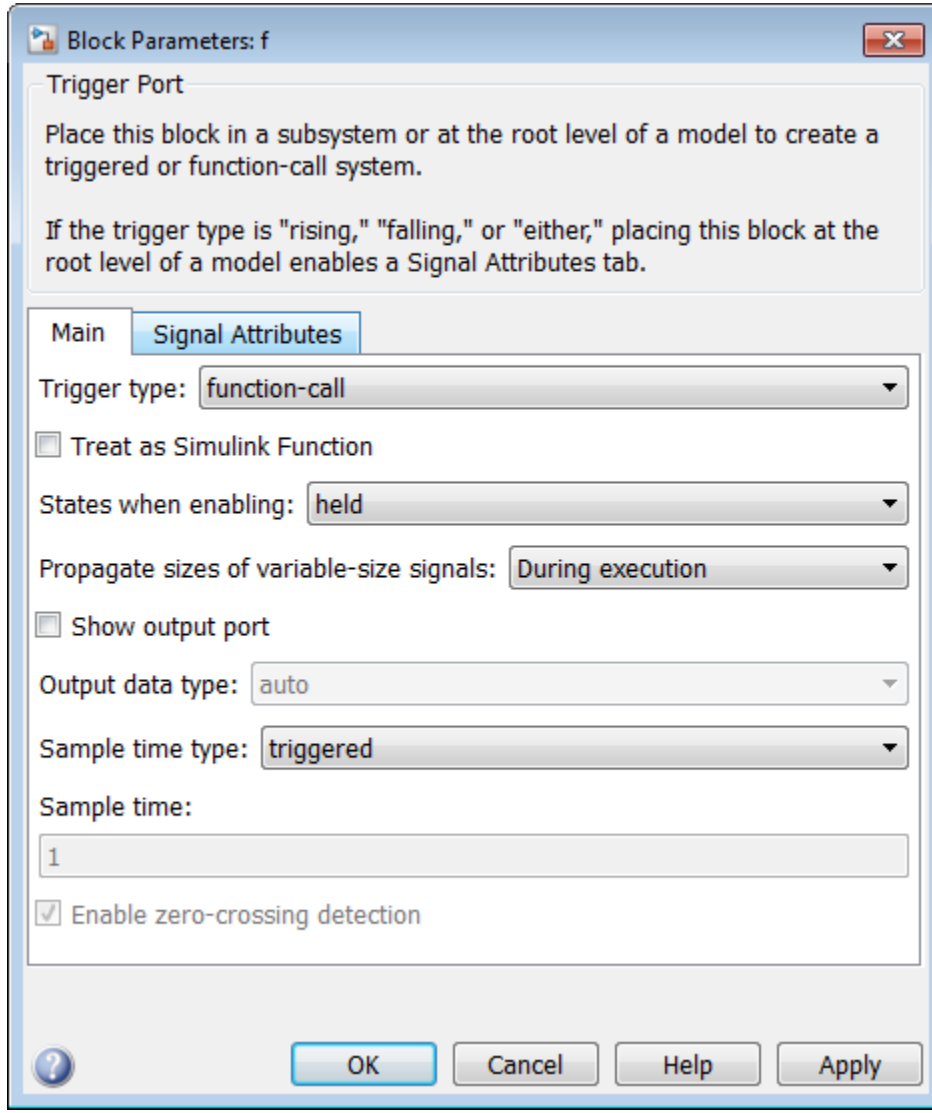
- “Create a Triggered Subsystem”
- “Create Conditional Models”
- “Function-Call Models”
- “Create a Function-Call Subsystem”

Data Type Support

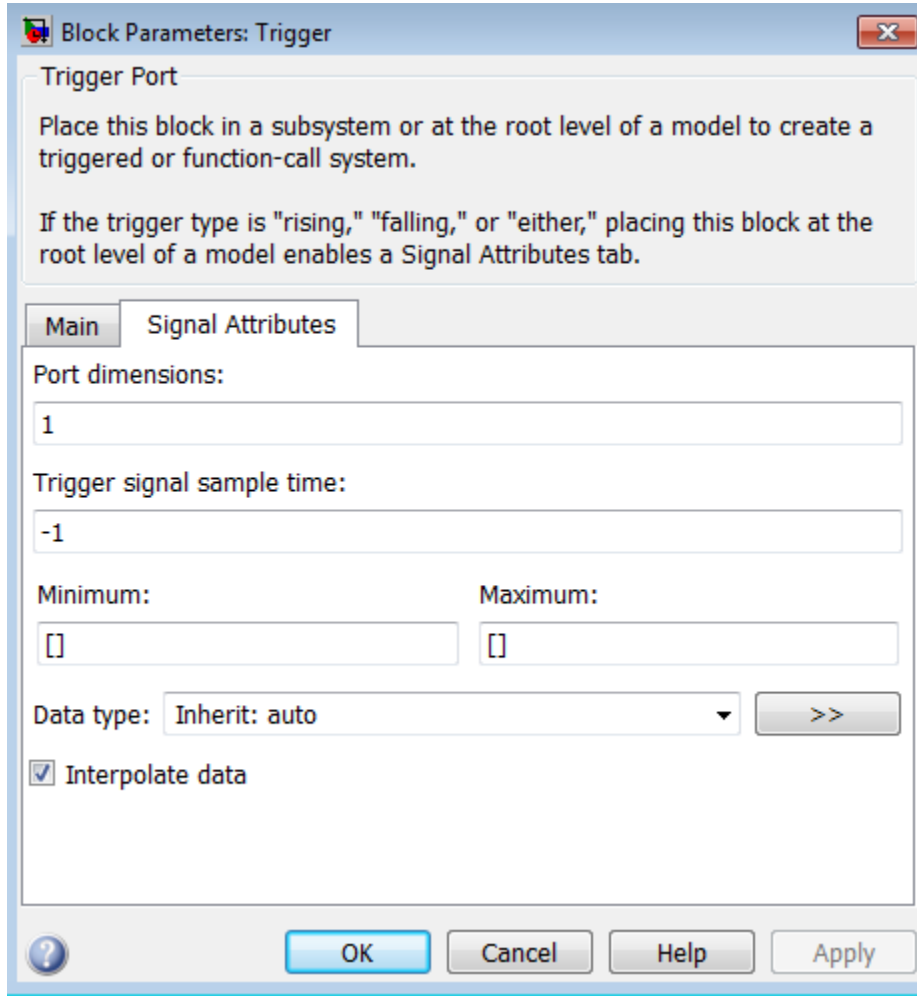
The Trigger block accepts signals of supported Simulink numeric data types, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

The **Main** tab of the Trigger block dialog box appears as follows:



The **Signal Attributes** tab of the Trigger block dialog box appears as follows:



- "Trigger type" on page 1-2092
- "Treat as Simulink Function" on page 1-2093
- "Function name" on page 1-2094
- "Enable variant condition" on page 1-2095
- "Variant control" on page 1-2096
- "Generate preprocessor conditionals" on page 1-2097

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 6-101 for the command-line information.

Treat as Simulink Function

Configure the Function-call subsystem to be a Simulink Function callable with arguments.

Settings

Default: `Off`

When you select the check box, a function prototype appears on the subsystem block icon, which you can edit to configure input and output arguments to the Simulink Function.

Tip

To use this check box to configure a subsystem as a Simulink Function, the Trigger block must reside in the subsystem.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Dependency

This parameter appears only when you select function-call as the **Trigger type** parameter.

Setting **Trigger type** to `function-call` and selecting the **Treat as a Simulink Function** check box enables the **Function name** and **Enable variant condition** parameters.

Function name

Specify the function name of Simulink Function.

Settings

Default: `f`

This parameter provides the function name in the function prototype of the Simulink Function.

Dependency

Setting **Trigger type** to `function-call` and selecting the **Treat as a Simulink Function** check box enables this parameter.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Enable variant condition

Enable a variant condition on the function-call port so that the port is activated when that condition is **true**.

Settings

Default: `Off`

When you select the check box, the function-call port of the Simulink Function block changes in appearance to indicate that variant conditions are enabled.

Dependency

Selecting this check box enables **Variant control** and **Generate preprocessor conditions** parameters.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Variant control

Variant control expression that activates this port when it evaluates to `true`. 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`

Dependency

Setting **Trigger type** to `function-call` and selecting the **Enable variant condition** check box enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Generate preprocessor conditionals

When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`).

Settings

Default: `off`

Dependency

Setting **Trigger type** to `function-call` and selecting the **Enable variant condition** check box enables this parameter.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 6-101 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 6-101 for the command-line information.

Show output port

Select this check box to output a signal that indicates the trigger type.

Settings

Default: On

 On

Shows the Trigger block output port and outputs the trigger type. Selecting this check box 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` in all other cases

 Off

Removes the output port.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 6-101 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. Set this parameter to `-1` to inherit the sample time from the incoming trigger signal.

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 6-101 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 6-101 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 6-101 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 6-101 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.

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 6-101 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.

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 6-101 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 6-101 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 6-101 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.

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 6-101 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.

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.

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.

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.

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 6-101 for the command-line information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
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.
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

- “Create Conditional Models”
- `Triggered Subsystem`
- `Function-Call Subsystem`
- `Simulink Function`

Introduced before R2006a

Trigger-Based Linearization

Generate linear models in base workspace when triggered

Library

Model-Wide Utilities

Description

When triggered, this block calls `linmod` or `dlinmod` to create a linear model for the system at the current operating point. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.

The block sets the following model parameters to the indicated values:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`

The name of the structure used to save the snapshots is the name of the model appended by `_Trigger_Based_Linearization`, for example, `vdp_Trigger_Based_Linearization`. The structure has the following fields:

Field	Description
a	The A matrix of the linearization
b	The B matrix of the linearization
c	The C matrix of the linearization

Field	Description
`d`	The D matrix of the linearization
`StateName`	Names of the model's states
`OutputName`	Names of the model's output ports
`InputName`	Names of the model's input ports
`OperPoint`	A structure that specifies the operating point of the linearization. The structure specifies the value of the model's states (`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.

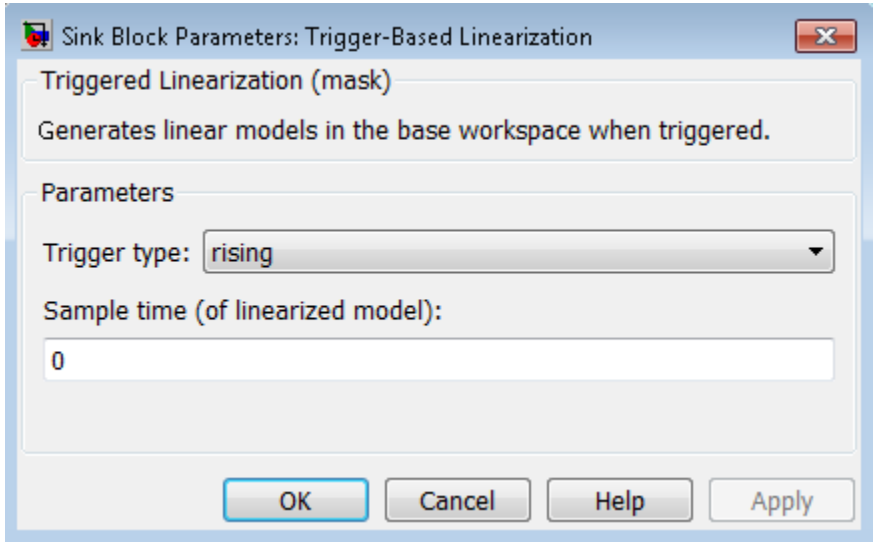
   ```
   x1 = xout(ind1,:);
   ```

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



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 2-34).

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	No

See Also

`Timed-Based Linearization`

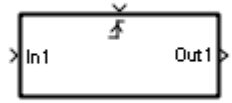
Introduced before R2006a

Triggered Subsystem

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 “Create a Triggered Subsystem”).

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

Introduced before R2006a

Trigonometric Function

Specified trigonometric function on input

Library

Math Operations

Description

Supported Functions

The Trigonometric Function block performs common trigonometric functions. You can select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
`sin`	Sine of the input	`sin(u)`	`sin`
`cos`	Cosine of the input	`cos(u)`	`cos`
`tan`	Tangent of the input	`tan(u)`	`tan`
`asin`	Inverse sine of the input	`asin(u)`	`asin`
`acos`	Inverse cosine of the input	`acos(u)`	`acos`
`atan`	Inverse tangent of the input	`atan(u)`	`atan`
`atan2`	Four-quadrant inverse tangent of the input	`atan2(u)`	`atan2`

Function	Description	Mathematical Expression	MATLAB Equivalent
`sinh`	Hyperbolic sine of the input	`sinh(`*u*`)`	`sinh`
`cosh`	Hyperbolic cosine of the input	`cosh(`*u*`)`	`cosh`
`tanh`	Hyperbolic tangent of the input	`tanh(`*u*`)`	`tanh`
`asinh`	Inverse hyperbolic sine of the input	`asinh(`*u*`)`	`asinh`
`acosh`	Inverse hyperbolic cosine of the input	`acosh(`*u*`)`	`acosh`
`atanh`	Inverse hyperbolic tangent of the input	`atanh(`*u*`)`	`atanh`
`sincos`	Sine of the input; cosine of the input	—	—
`cos + jsin`	Complex exponential of the input	—	—

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 one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

Data Type Support

The block accepts input signals of the following data types:

Functions	Input Data Types
• `sin` • `cos` • `sincos` • `cos + jsin` • `atan2`	• Floating point • Fixed point (only when **Approximation method** is `CORDIC`)
• `tan` • `asin` • `acos`	• Floating point

Functions	Input Data Types
• `atan` • `sinh` • `cosh` • `tanh` • `asinh` • `acosh` • `atanh`	

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 1-2129), 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.

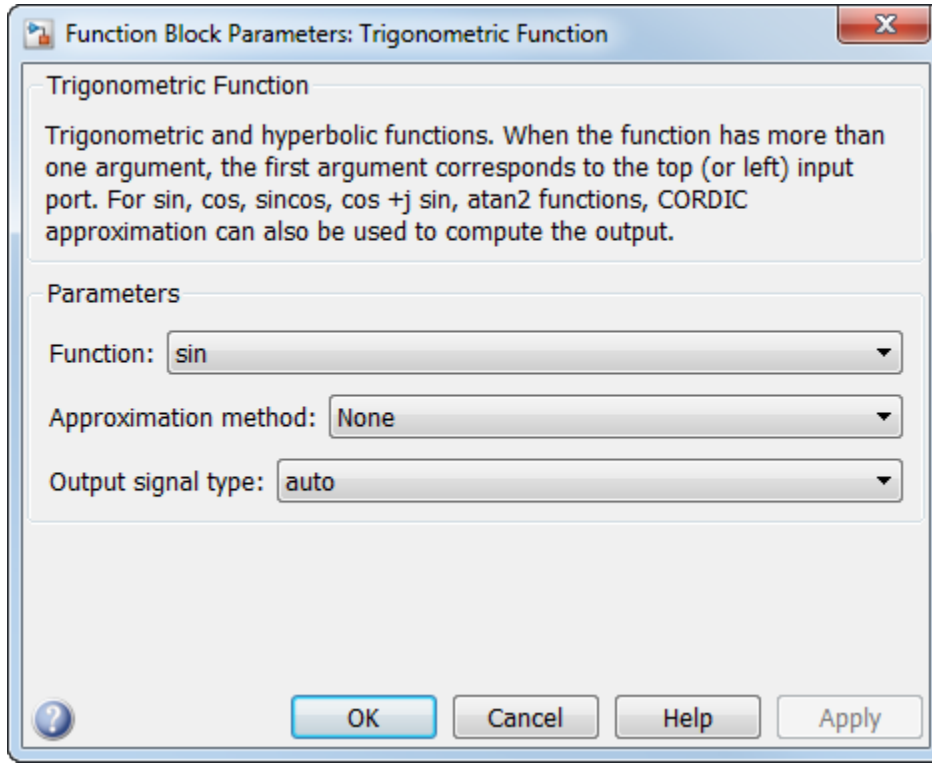
For the `atan2` function:

- 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	

Parameters and Dialog Box



Function

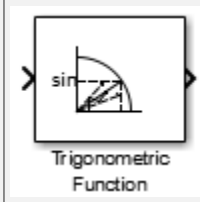
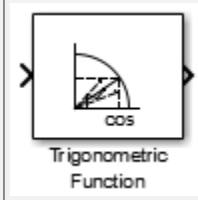
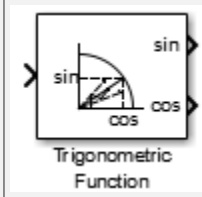
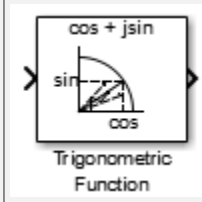
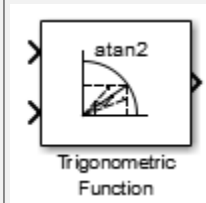
Specify the trigonometric function. The name of the function on the block icon changes to match your selection.

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`	
`cos`	
`sincos`	
`cos + jsin`	
`atan2`	

Number of iterations

Specify the number of iterations to perform the CORDIC algorithm. The default value is 11.

- When the block input uses a floating-point data type, the number of iterations can be a positive integer.
- When the block input is a fixed-point data type, the number of iterations cannot exceed the word length.

  For example, if the block input is `fixdt(1,16,15)`, the word length is 16. In this case, the number of iterations cannot exceed 16.

This parameter appears when both of the following conditions hold:

- You set **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`.
- You set **Approximation method** to `CORDIC`.

Output signal type

Specify the output signal type of the Trigonometric Function block as `auto`, `real`, or `complex`.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
Any selection for the **Function** parameter	real	real	real	complex
	complex	complex	error	complex

Note: When **Function** is `atan2`, complex input signals are not supported for simulation or code generation.

Setting **Approximation method** to `CORDIC` disables this parameter.

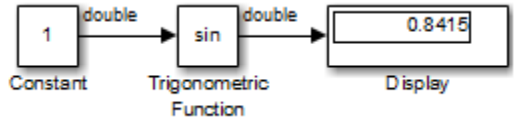
Sample time

Note: This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see "Blocks for Which Sample Time Is Not Recommended".

Examples

sin Function with Floating-Point Input

Suppose that you have the following model:



The key block parameters for the Constant block are:

Parameter	Setting
Constant value	`1`
Output data type	`Inherit: Inherit from 'Constant value'`

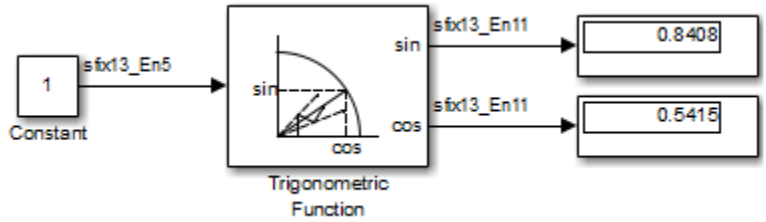
The block parameters for the Trigonometric Function block are:

Parameter	Setting
Function	`sin`
Approximation method	`None`
Output signal type	`auto`

The output type of the Trigonometric Function block is the same as the input because the input type is floating point and **Approximation method** is `None`.

sincos Function with Fixed-Point Input

Suppose that you have the following model:



The key block parameters for the Constant block are:

Parameter	Setting
Constant value	1 This value must fall within the range [–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)`

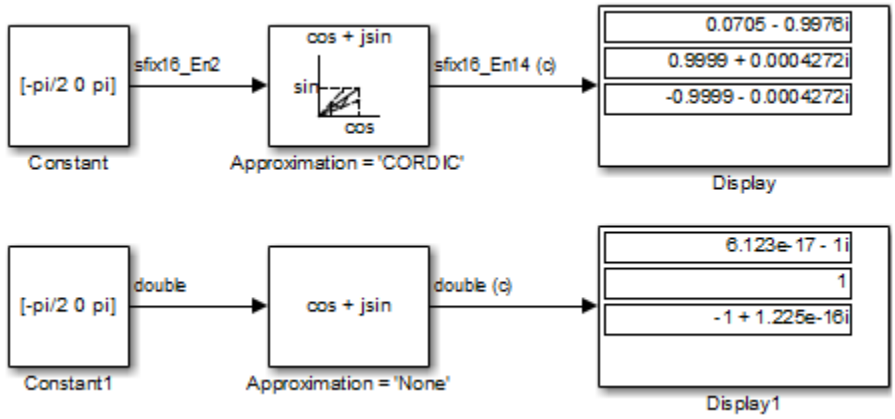
The block parameters for the Trigonometric Function block are:

Parameter	Setting
Function	`sincos`
Approximation method	`CORDIC`
Number of iterations	`11`

The output type of the Trigonometric Function block is `fixdt(1,13,11)` because the input type is fixed point and **Approximation method** is `CORDIC`. The output fraction length equals the input word length – 2.

Block Behavior for Complex Exponential Output

The following model compares the complex exponential output for the two different approximation methods:



The key block parameters for the Constant blocks are:

Block	Parameter	Setting
`Constant`	**Constant value**	`[-pi/2 0 pi]`
	Output data type	`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`
`Approximation = 'None'`	**Function**	`cos + jsin`
	Approximation method	`None`

When the **Approximation method** is `CORDIC`, the input data type can be fixed point, in this case: `fixdt(1,16,2)`. The output data type is `fixdt(1,16,14)` because the output fraction length equals the input word length – 2.

When the **Approximation method** is `None`, the input data type must be floating point. The output data type is the same as the input.

Characteristics

Data Types	Double | Single
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

References

[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

See Also

`Math Function`, `Sqrt`

Introduced before R2006a

Unary Minus

Negate input

Library

Math Operations

Description

The Unary Minus block negates the input.

For signed-integer data types, the unary minus of the most negative value is not representable by the data type. In this case, the **Saturate on integer overflow** check box controls the behavior of the block:

If you...	The block...	And...
Select this check box	Saturates to the most positive value of the integer data type	• 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.

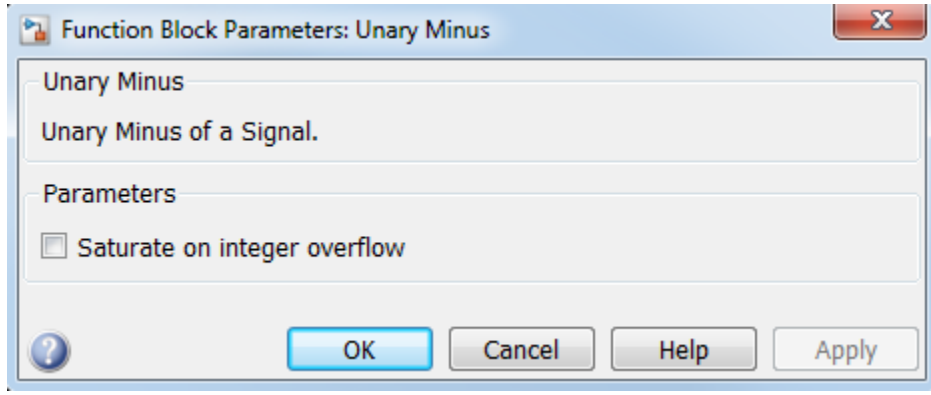
Data Type Support

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



Sample time

> **Note:** This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see "Blocks for Which Sample Time Is Not Recommended".

Saturate on integer overflow

Select to have integer overflows saturate. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can

detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

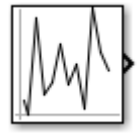
Introduced before R2006a

Uniform Random Number

Generate uniformly distributed random numbers

Library

Sources

Description 

The Uniform Random Number block generates uniformly distributed random numbers over an interval that you specify. To generate normally distributed random numbers, use the `Random Number` block.

You can generate a repeatable sequence using any Uniform Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts.

Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the `Band-Limited White Noise` block.

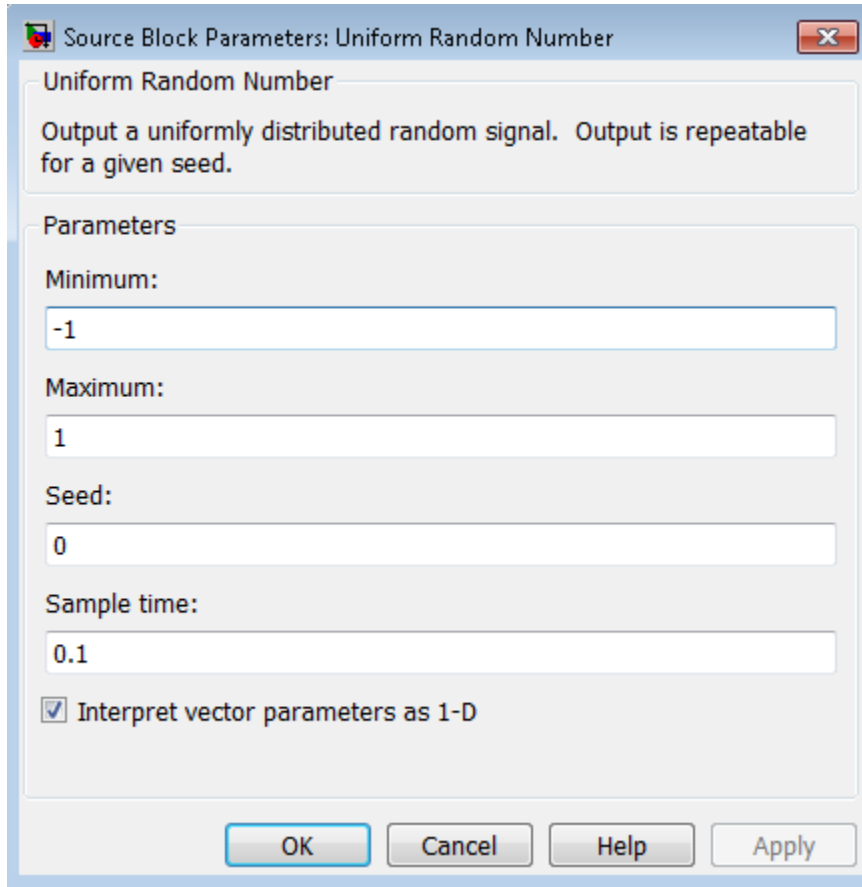
The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the **Interpret vector parameters as 1-D** check box and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensionality as the parameters.

Data Type Support

The Uniform Random Number block accepts and outputs a real signal of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters 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

Data Types	Double
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

The generator algorithm is identical to the one used in MATLAB Version 4.0 by the `rand` and `randn` functions. For details on the `mcg16807` algorithm, see "Choosing a Random Number Generator" in the MATLAB documentation.

To use other algorithms supported by MATLAB in a Simulink model, generate a set of random numbers in MATLAB, and store the output as a .mat file. Use this .mat file as the random number input for your simulation. For more information, see "Creating and Controlling a Random Number Stream". To create multiple independent streams using MATLAB, see "Multiple streams"

Note: Using multiple seeds to generate multiple parallel independent streams for a generator algorithm is not recommended for the `mcg16807` algorithm. Instead, use the method described above.

See Also

`Random Number`

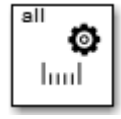
Introduced before R2006a

Unit System Configuration

Configure units

Library

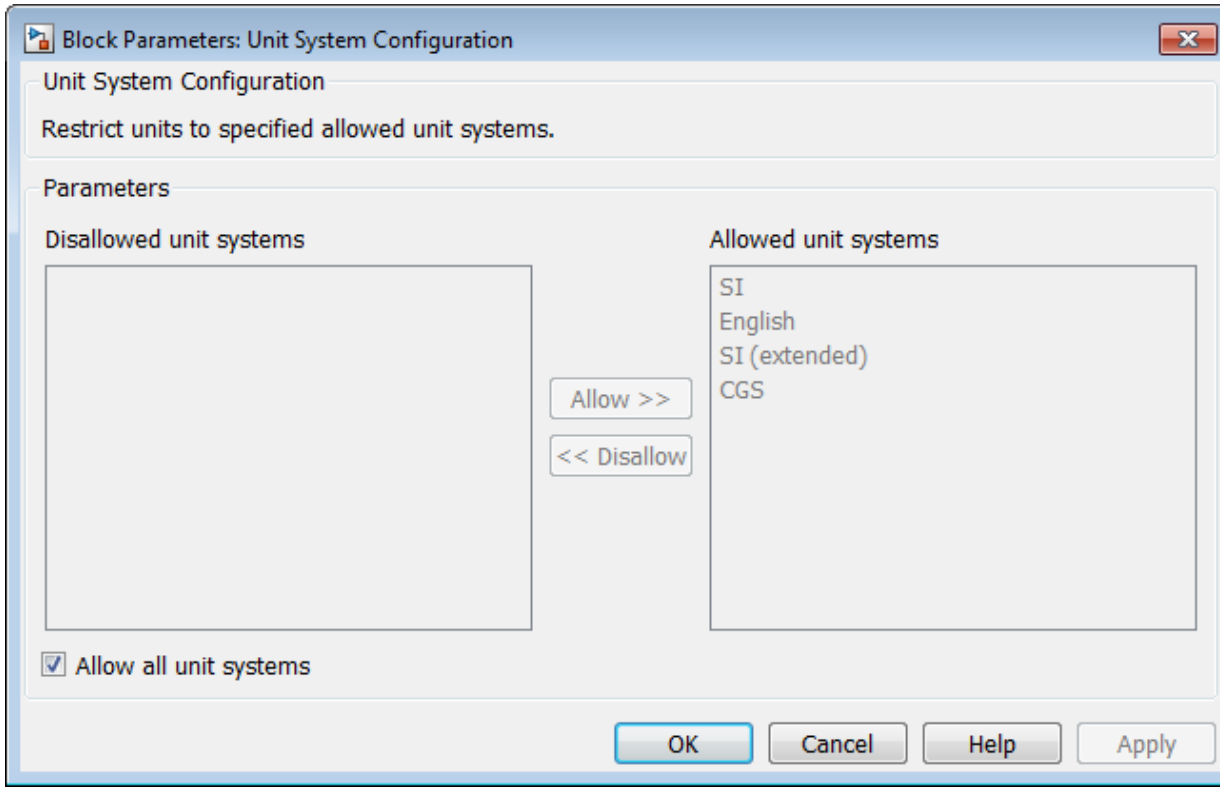
Ports & Subsystems

Description 

The `Unit System Configuration` block specifies allowed and disallowed unit systems for the component. It restricts units systems for a subsystem or top model and all its children, unless you override it with another `Unit System Configuration` block in a child.

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Parameters and Dialog Box



Disallowed unit systems

Displays disallowed unit system.

To designate a unit system as disallowed, select it in the **Allowed unit systems** column and click **<<Disallow**.

Settings

Default: None

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Allowed unit systems

Displays allowed unit system.

To designate a unit system as allowed, select it in the **Unit systems** column and click **<<Allow**.

Settings

Default: `SI`, `English`, `SI (extended)`, `CGS`

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

Allow all unit systems

Allow or restrict unit systems.

Settings

Default: On

 On

Allow all unit systems.

 Off

Restrict unit systems to those in **Allowed unit systems**.

Dependencies

Selecting the **Allowed unit systems** check box disables the **Disallowed unit systems** and **Allow all unit systems** parameters.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 for the command-line information.

See Also

`Inport` | `Outport` | `Unit Conversion`

More About

- “Units in Simulink”
- “Unit Specification in Simulink Models”
- “Restricting Unit Systems”

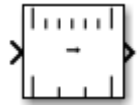
Introduced in R2016a

Unit Conversion

Convert units

Library

Signal Attributes

Description 

The `Unit Conversion` block converts the unit of the input signal to the output signal. The block can convert if the units are separated by a scaling factor or offset, or are inverse units, for example:

- $y=a*U$
- $y=a*U+b$, where a is the scale and b is the offset
- $y=a/U$

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

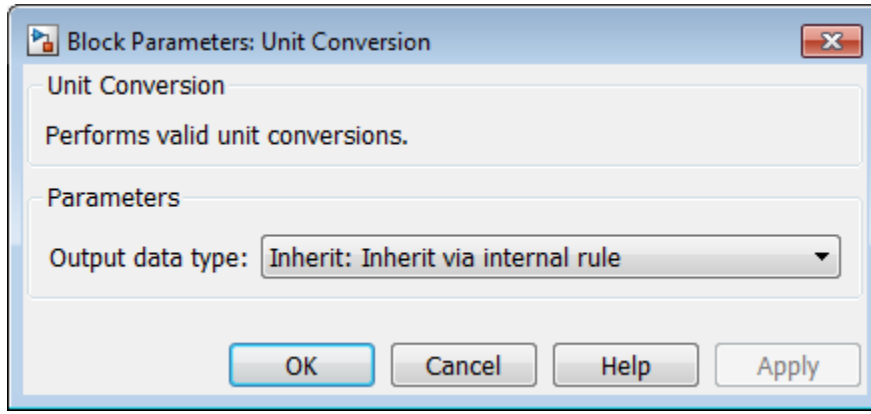
Data Type Support

The Unit Conversion block accepts and outputs real or complex values of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink”.

Parameters and Dialog Box



Output data type

Specify the output data type.

Settings

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`

Simulink chooses intermediate and output data types to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change.

`Inherit: Inherit via back propagation`

Output data type is inherited via back propagation. Internal rules determine the intermediate data types and Simulink casts the final results to the output data type.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

See Also

See “Converting Units” for more information.

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

"Converting Units"

More About

- "Units in Simulink"

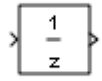
Introduced in R2016a

Unit Delay

Delay signal one sample period

Library

Discrete



Description

The Unit Delay block holds and delays its input by the sample period you specify. This block is equivalent to the z^{-1} discrete-time operator. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds and delays all elements of the vector by the same sample period.

You specify the block output for the first sampling period with the **Initial conditions** parameter. Careful selection of this parameter can minimize unwanted output behavior. You specify the time between samples with the **Sample time** parameter. A setting of `-1` means the block inherits the **Sample time**.

Note: The Unit Delay block errors out if you use it to create a transition between blocks operating at different sample rates. Use the `Rate Transition` block instead.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
`Unit Delay`	Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.	• `sldemo_enginewc` (Compression subsystem)
Memory	Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.	• `sldemo_bounce` • `sldemo_clutch` (Friction Mode Logic/Lockup FSM subsystem)
`Zero-Order Hold`	Convert an input signal with a continuous sample time to an output signal with a discrete sample time.	• `sldemo_radar_eml` • `aero_dap3dof`

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

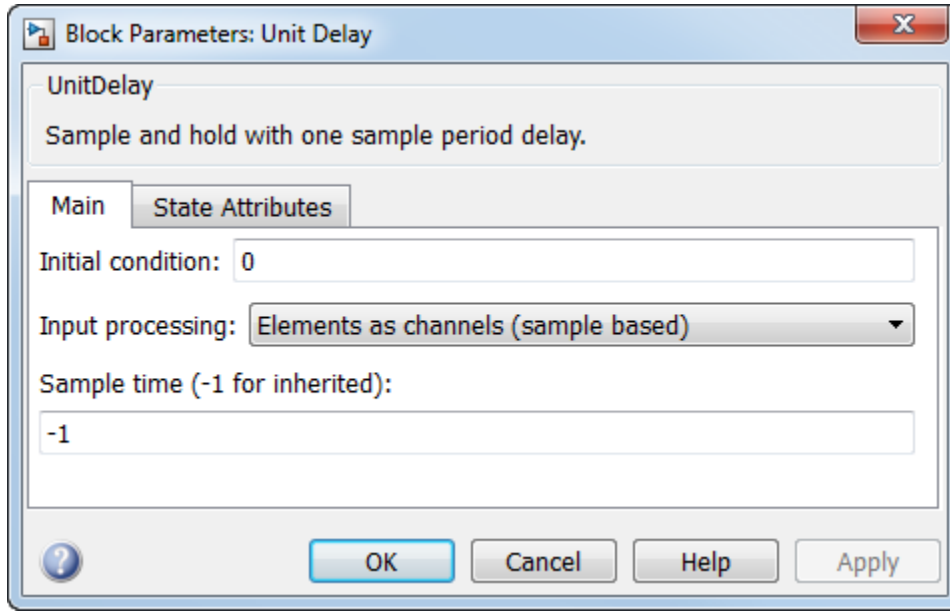
Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Support for state logging	Yes	No	No

Data Type Support

The Unit Delay block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. If the data type of the input signal is user-defined, the initial condition must be zero.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

For more information, see “Discrete Block State Naming in Generated Code” in the Simulink Coder documentation.

Initial conditions

Specify the output of the simulation for the first sampling period, during which the output of the Unit Delay block is otherwise undefined.

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 6-101 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 6-101 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify `-1` to inherit the sample time.

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 6-101 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 "Discrete Block State Naming in Generated Code" in the Simulink Coder documentation.

Command-Line Information

Parameter: `StateIdentifier`
Type: string
Value: `' '`
Default: `' '`

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

Parameter: `StateMustResolveToSignalObject`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes do not affect the generated code.

Settings

`Simulink.Signal`

Use custom storage classes from the built-in package `Simulink`.

`Package.Class`

Use custom storage classes from the package that defines the class that you select.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see "Apply Custom Storage Classes Directly to Signal Lines and Block States".

Code generation storage class

Select state storage class for code generation.

Settings

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

`StorageClass`

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see "Control Signals and States in Code by Applying Storage Classes". For information about custom storage classes, see "Control Data Representation by Applying Custom Storage Classes".

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

Dependencies

State name enables this parameter.

TypeQualifier

Note: **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Settings

- **Default:** `' '` (empty string)
- `const`
- `volatile`

Dependency

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Command-Line Information

Parameter Name: `RTWStateStorageTypeQualifier`
Value Type: string
Default: `' '` (empty string)

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 "Composite Signals" and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Unit Delay block. You can specify the **Initial conditions** parameter with:

- The value `0`. In this case, all of the individual signals in the array of buses use the initial value `0`.
- An array of structures that specifies an initial condition for each of the individual signals in the array of buses.
- A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

For details about defining and using an array of buses, see "Combine Buses into an Array of Buses".

Examples

For an example of how to use the Unit Delay block, see the `sldemo_enginewc` model. The Unit Delay block appears in the `Compression` subsystem.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

"Scalar Expansion of Inputs and Parameters" | `Delay` | `Memory` | `Resettable Delay` | `Tapped Delay` | `Variable Integer Delay` | `Zero-Order Hold`

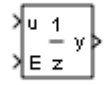
Introduced before R2006a

Unit Delay Enabled

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

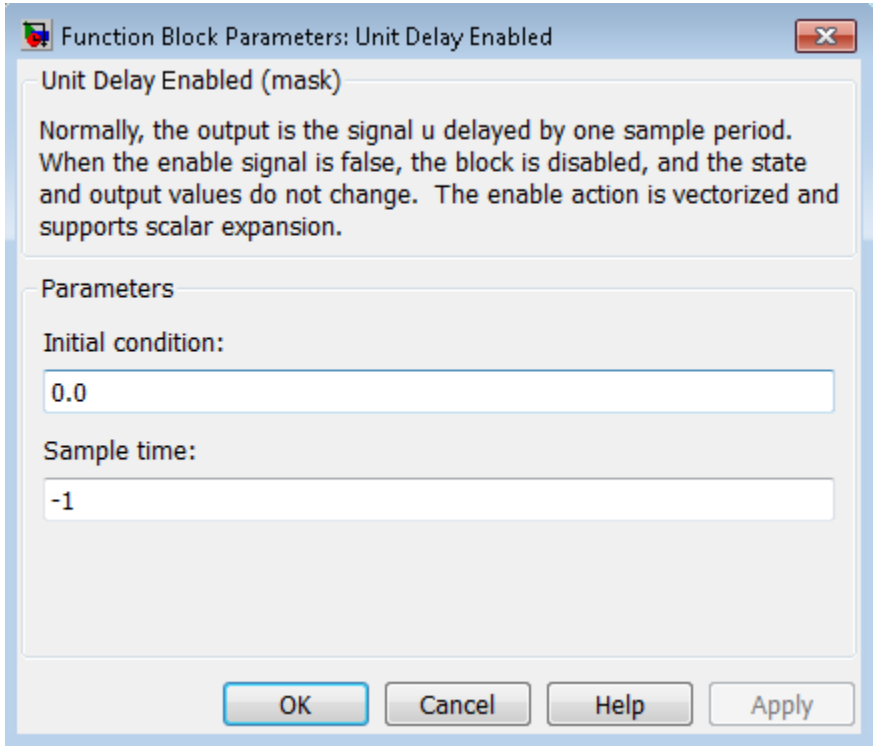
The Unit Delay Enabled block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`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`

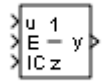
Introduced before R2006a

Unit Delay Enabled External IC

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

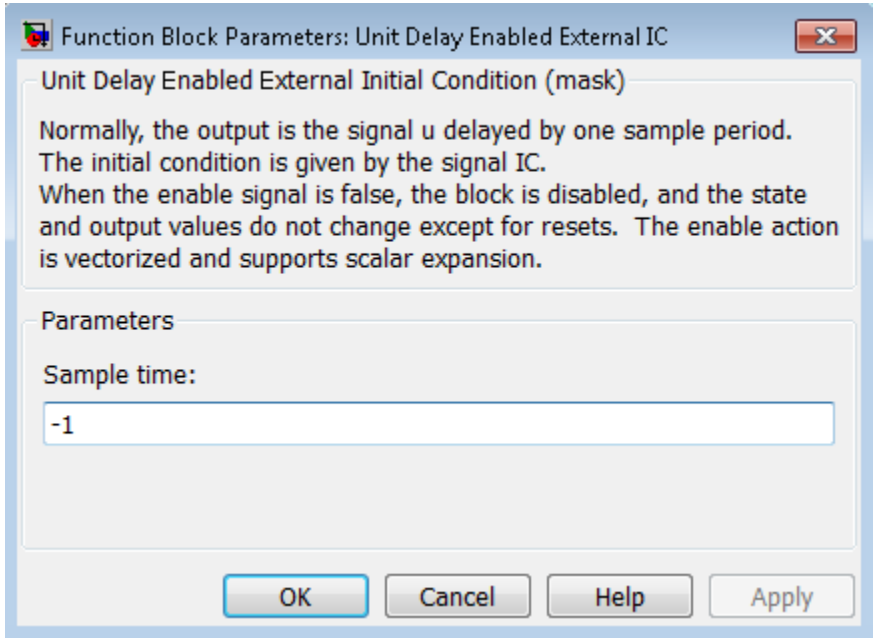
The Unit Delay Enabled External IC block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter

Direct Feedthrough	Yes, of the reset input port No, of the enable input port Yes, of the external `IC` port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

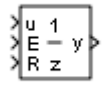
Introduced before R2006a

Unit Delay Enabled Resettable

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 and its output to the **Initial condition**.

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:

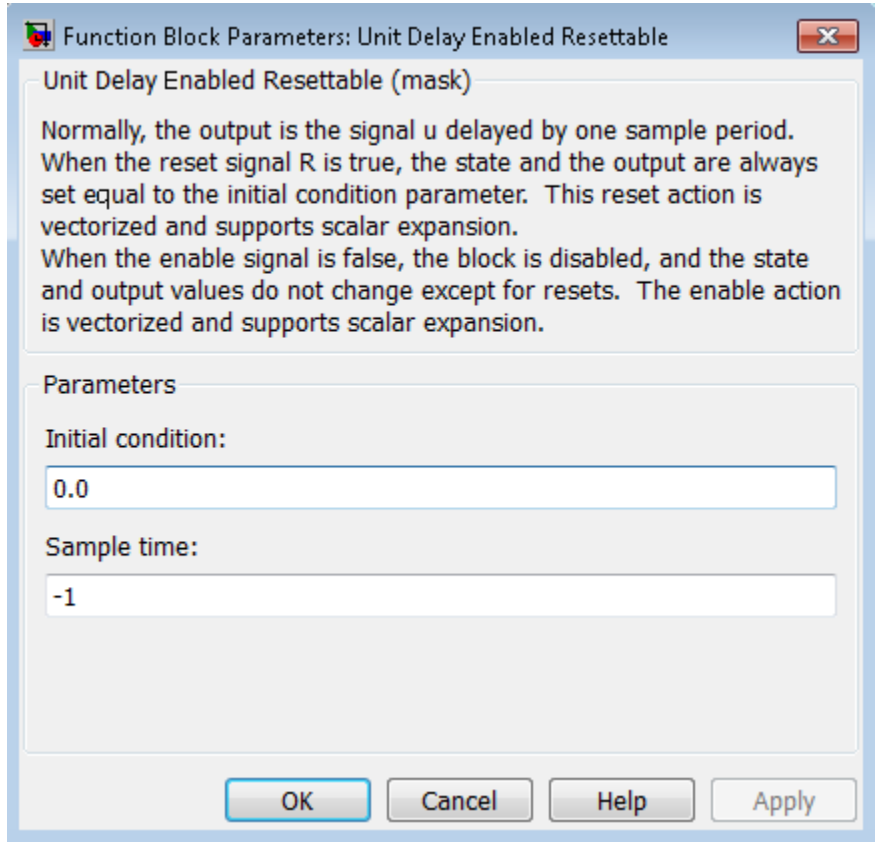
- Floating point
- Built-in integer

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	No, of the input port No, of the enable port Yes, of the reset port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

Introduced before R2006a

Unit Delay Enabled Resettable External IC

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 and its output to the **Initial condition**.

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:

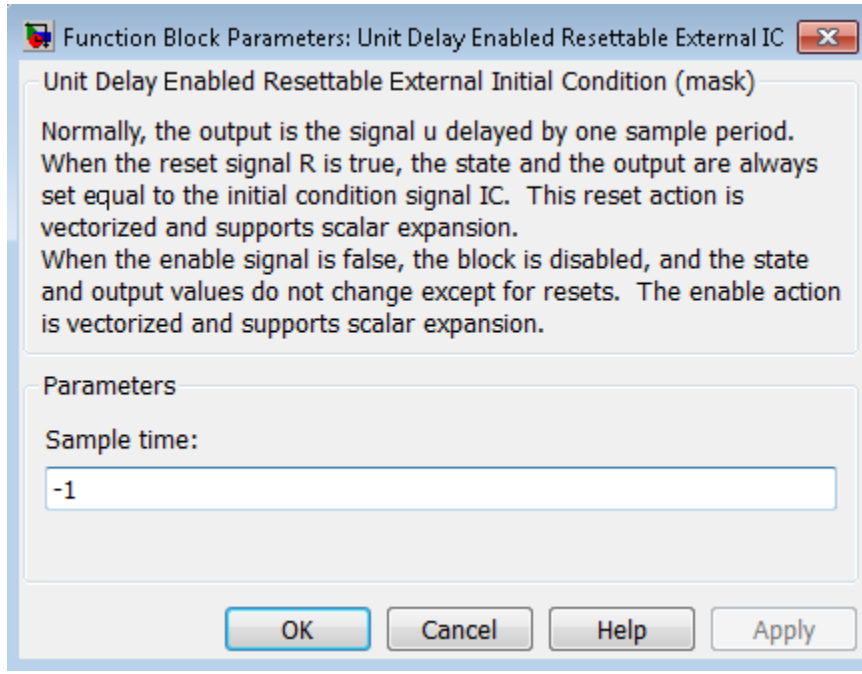
- Floating point

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	No, of the input port No, of the enable port Yes, of the enable port Yes, of the external `IC` port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

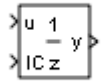
Introduced before R2006a

Unit Delay External IC

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

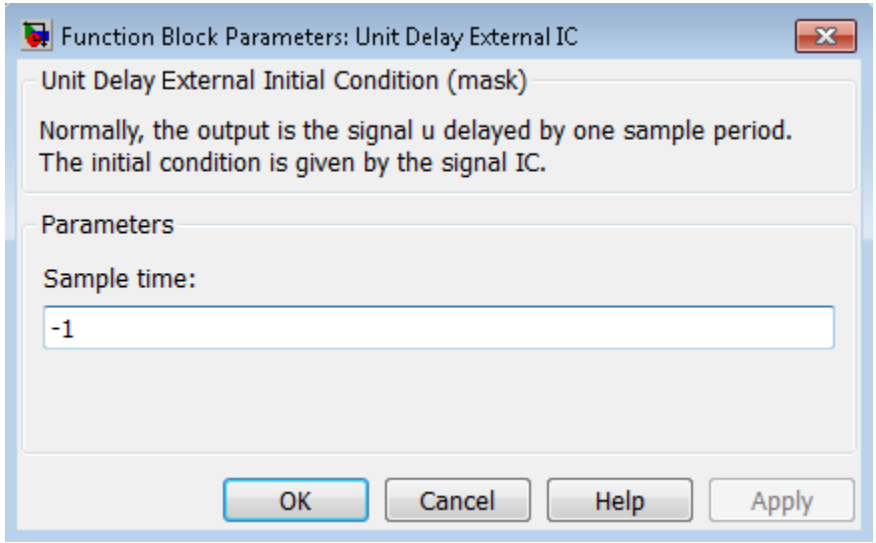
The Unit Delay External IC block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	No, of the input port Yes, of the external `IC` port

Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

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`

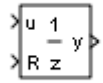
Introduced before R2006a

Unit Delay Resettable

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:

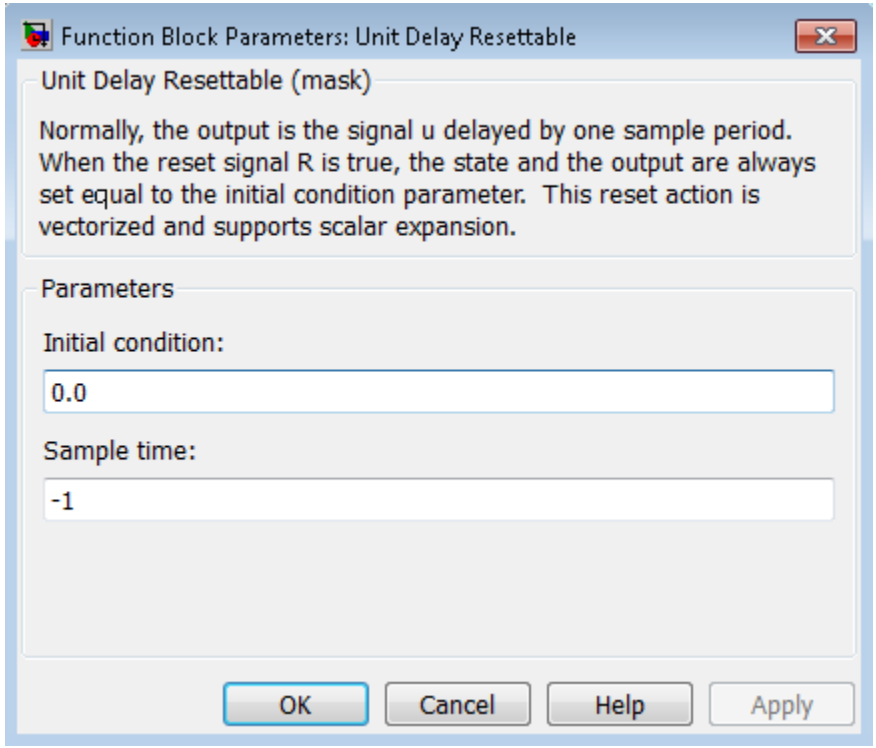
- Floating point
- Built-in integer
- Fixed point

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	No, of the input port Yes, of the reset port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

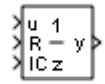
Introduced before R2006a

Unit Delay Resettable External IC

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 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

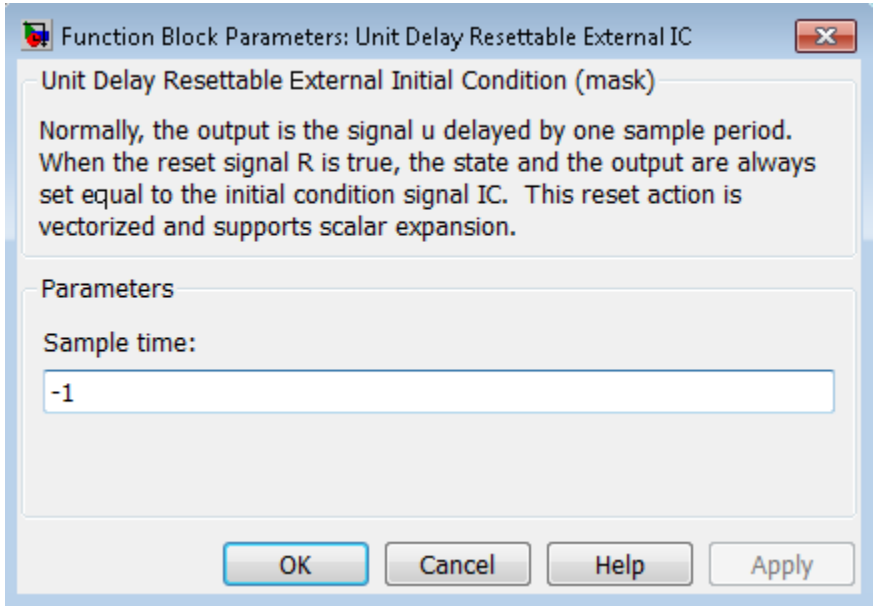
The Unit Delay Resettable External IC block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter

Direct Feedthrough	No, of the input port Yes, of the reset port Yes, of the external `IC` port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

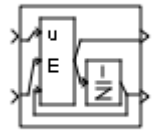
Introduced before R2006a

Unit Delay With Preview Enabled

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

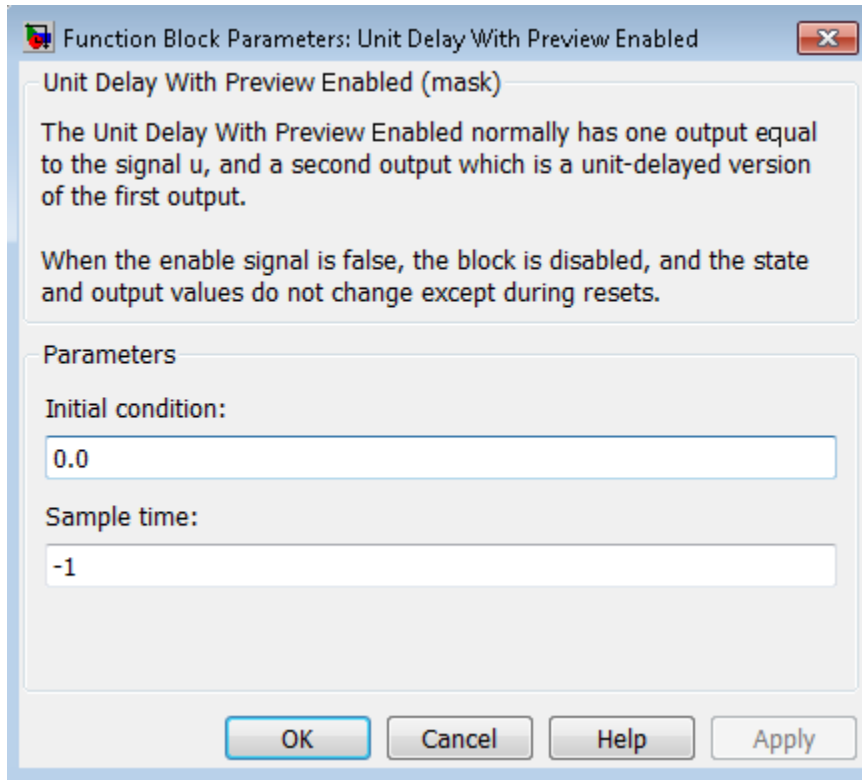
The Unit Delay With Preview Enabled block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, to first output port No, to second output port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

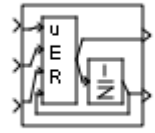
Introduced before R2006a

Unit Delay With Preview Enabled Resettable

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

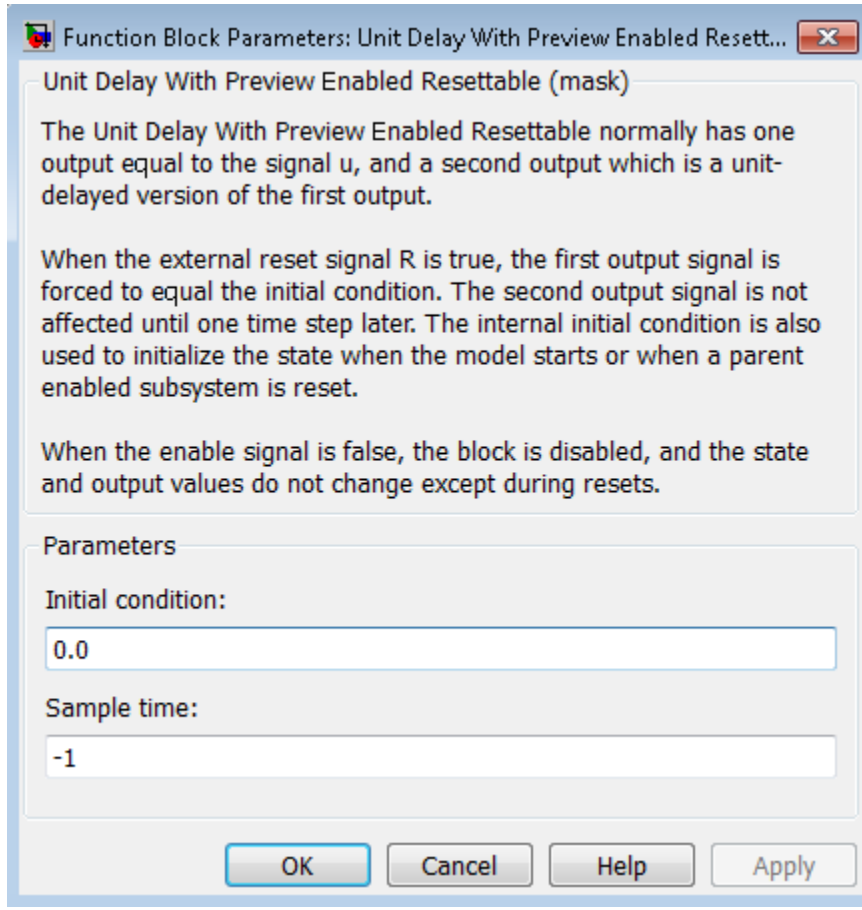
The Unit Delay With Preview Enabled Resettable block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, to first output port No, to second output port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

Introduced before R2006a

Unit Delay With Preview Enabled Resettable External RV

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

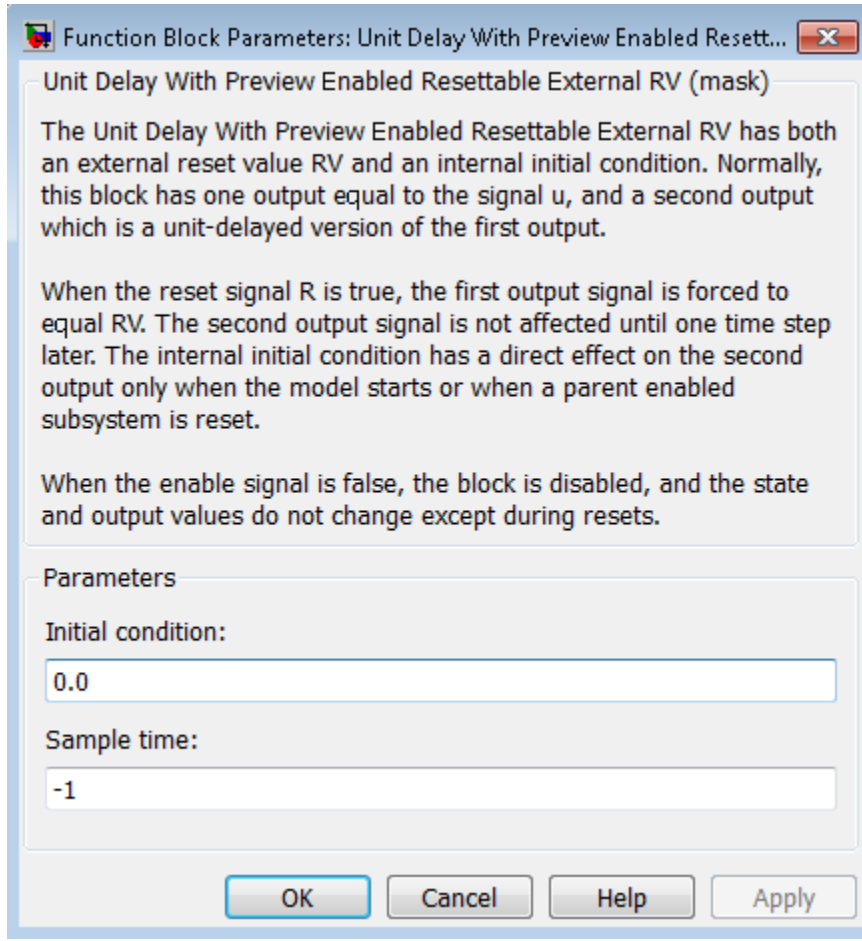
The Unit Delay With Preview Enabled Resettable External RV block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, to first output port No, to second output port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

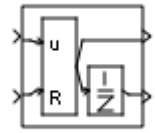
Introduced before R2006a

Unit Delay With Preview Resettable

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

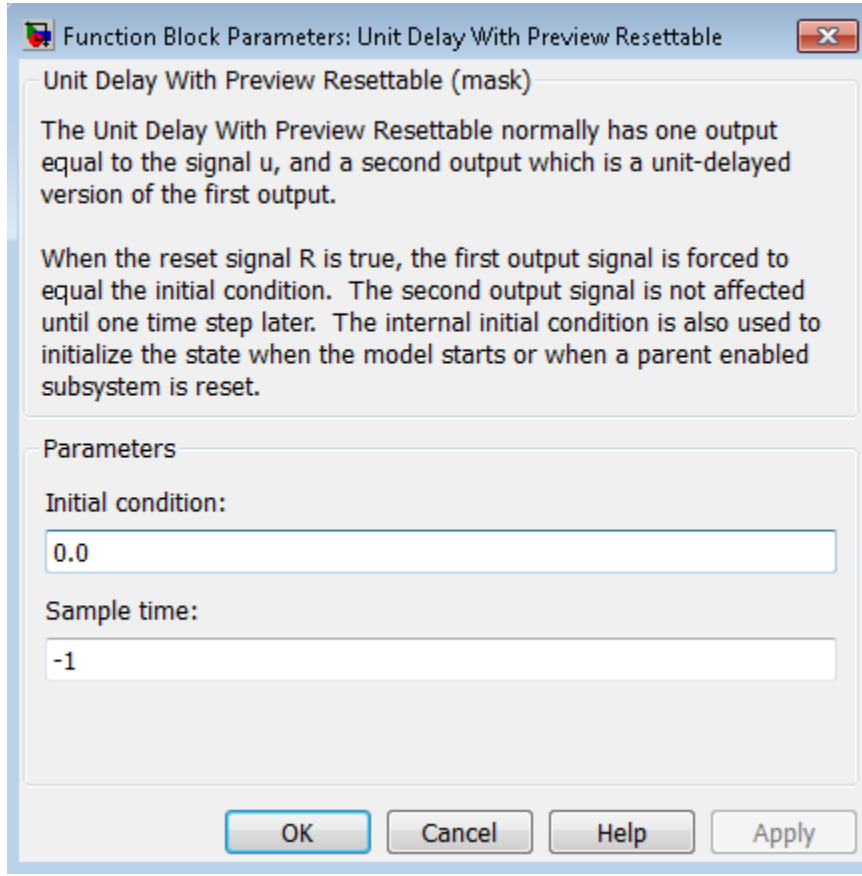
The Unit Delay With Preview Resettable block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, to first output port No, to second output port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

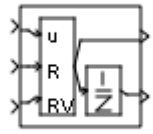
Introduced before R2006a

Unit Delay With Preview Resettable External RV

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

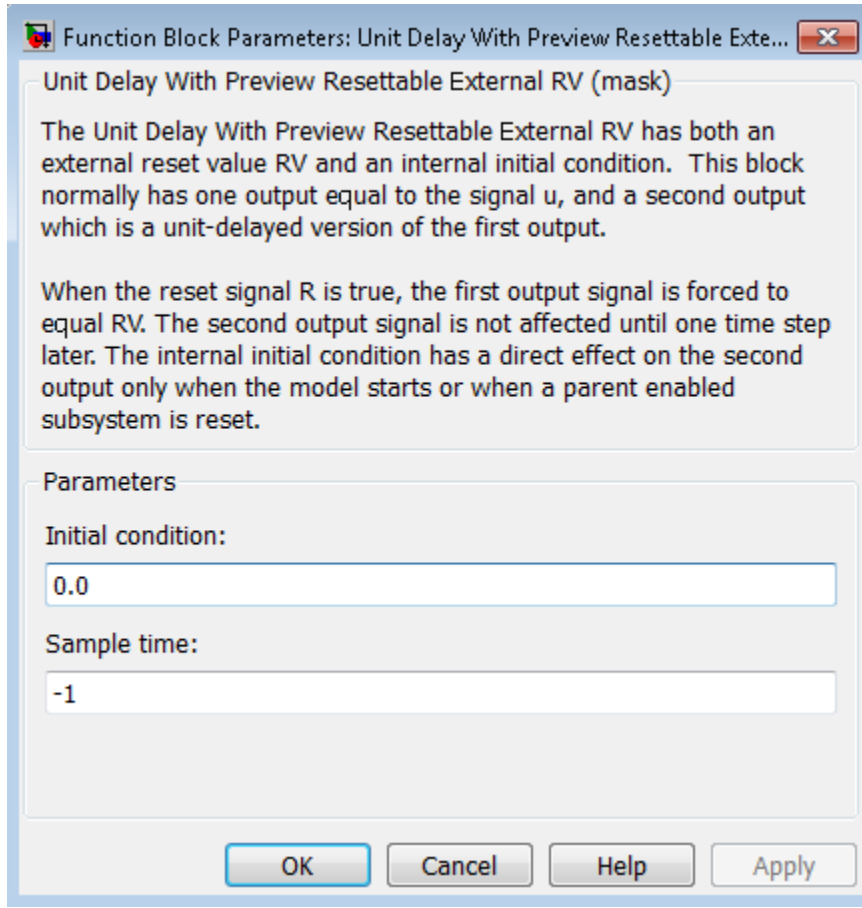
The Unit Delay With Preview Resettable External RV block accepts signals of the following data types:

- 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.

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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes, to first output port No, to second output port
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

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`

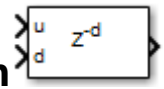
Introduced before R2006a

Variable Integer Delay

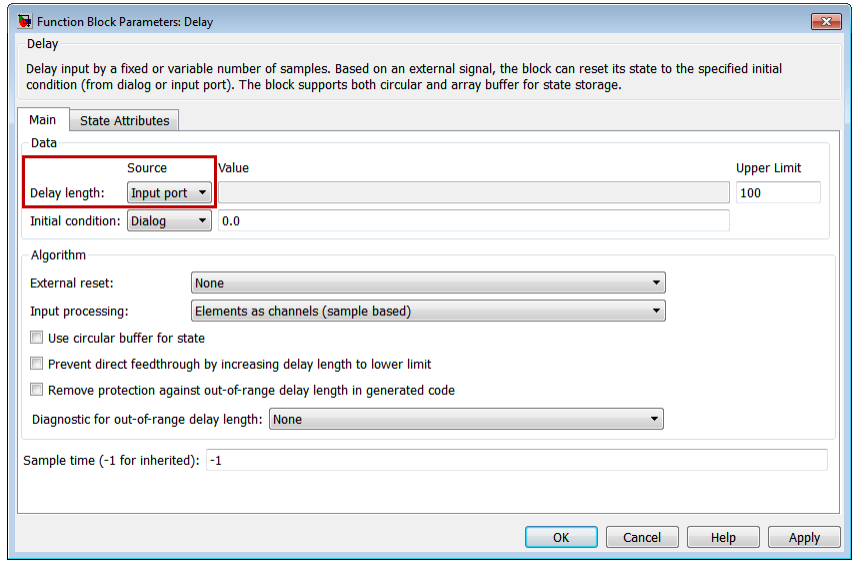
Delay input signal by variable sample period

Library

Discrete

Description 

The `Variable Integer Delay` block is a variant of the `Delay` block that has the source of the delay length set to `Input port`, by default.



Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

See Also

`Delay` | `Resettable Delay` | `Tapped Delay` | `Unit Delay`

Introduced in R2012b

Variable Time Delay, Variable Transport Delay

Delay input by variable amount of time

Library

Continuous

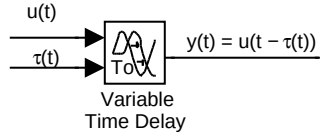
Description

The Variable Transport Delay and Variable Time Delay appear as two blocks in the Simulink block library. However, they are the same Simulink block with different settings of a **Select delay type** parameter. Use this parameter to specify the mode in which the block operates.

Variable Time Delay

In this mode, the block has a data input, a time delay input, and a data output. (See "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 step. This time step is the current simulation time minus a delay time specified by the time delay input.

$$y(t) = u(t - t_0) = u(t - \tau(t))$$



During the simulation, the block stores time and input value pairs in an internal buffer. At the start of simulation, the block outputs the value of the **Initial output** parameter until the simulation time exceeds the time delay input. Then, at each simulation step, the block outputs the signal at the time that corresponds to the current simulation time minus the delay time.

If you want the output at a time between input storing times and the solver is a continuous solver, the block interpolates linearly between points. If the time delay is smaller than the step size, the block extrapolates an output point from a previous point. For example, consider a fixed-step simulation with a step size of 1 and the current time at t = 5. If the delay is 0.5, the block needs to generate a point at t = 4.5, but the most recent stored time value is at t = 4. Thus, the block extrapolates the input at 4.5 from the input at 4 and uses the extrapolated value as its output at t = 5.

Extrapolating forward from the previous time step can produce a less accurate result than extrapolating back from the current time step. However, the block cannot use the current input to calculate its output value because the input port does not have direct feedthrough.

If the model specifies a discrete solver, the block does not interpolate between time steps. Instead, it returns the nearest stored value that precedes the required value.

Variable Transport Delay

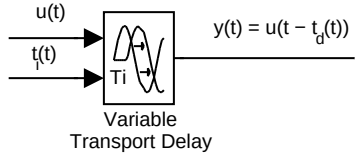
In this mode, the block output at the current time step is equal to the value of its data (top, or left) input at an earlier time step equal to the current time minus a transportation delay.

$$y(t) = u(t - t_d(t))$$

Simulink software finds the transportation delay, $t_d(t)$, by solving the following equation:

$$\int_{t-t_d(t)}^{t} \frac{1}{t_i(\tau)} d\tau = 1$$

This equation involves an instantaneous time delay, $t_i(t)$, given by the time delay (bottom, or right) input.



Suppose you want to use this block to model the fluid flow through a pipe where the fluid speed varies with time. In this case, the time delay input to the block is

$$t_i(t) = \frac{L}{v_i(t)}$$

where L is the length of the pipe and $v_i(t)$ is the speed of the fluid.

Data Type Support

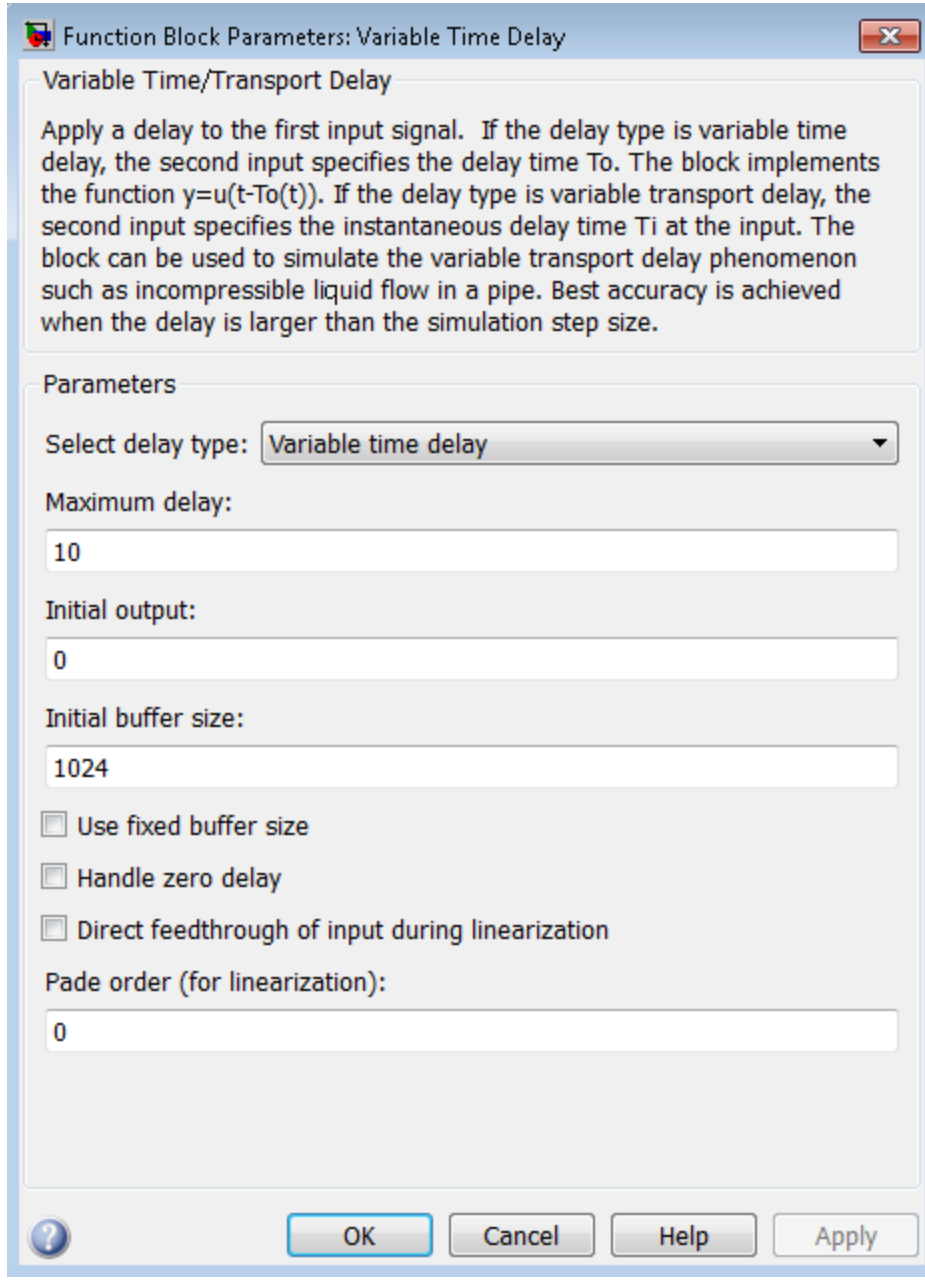
The Variable Time Delay and Variable Transport Delay blocks accept and output real signals of type `double`.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

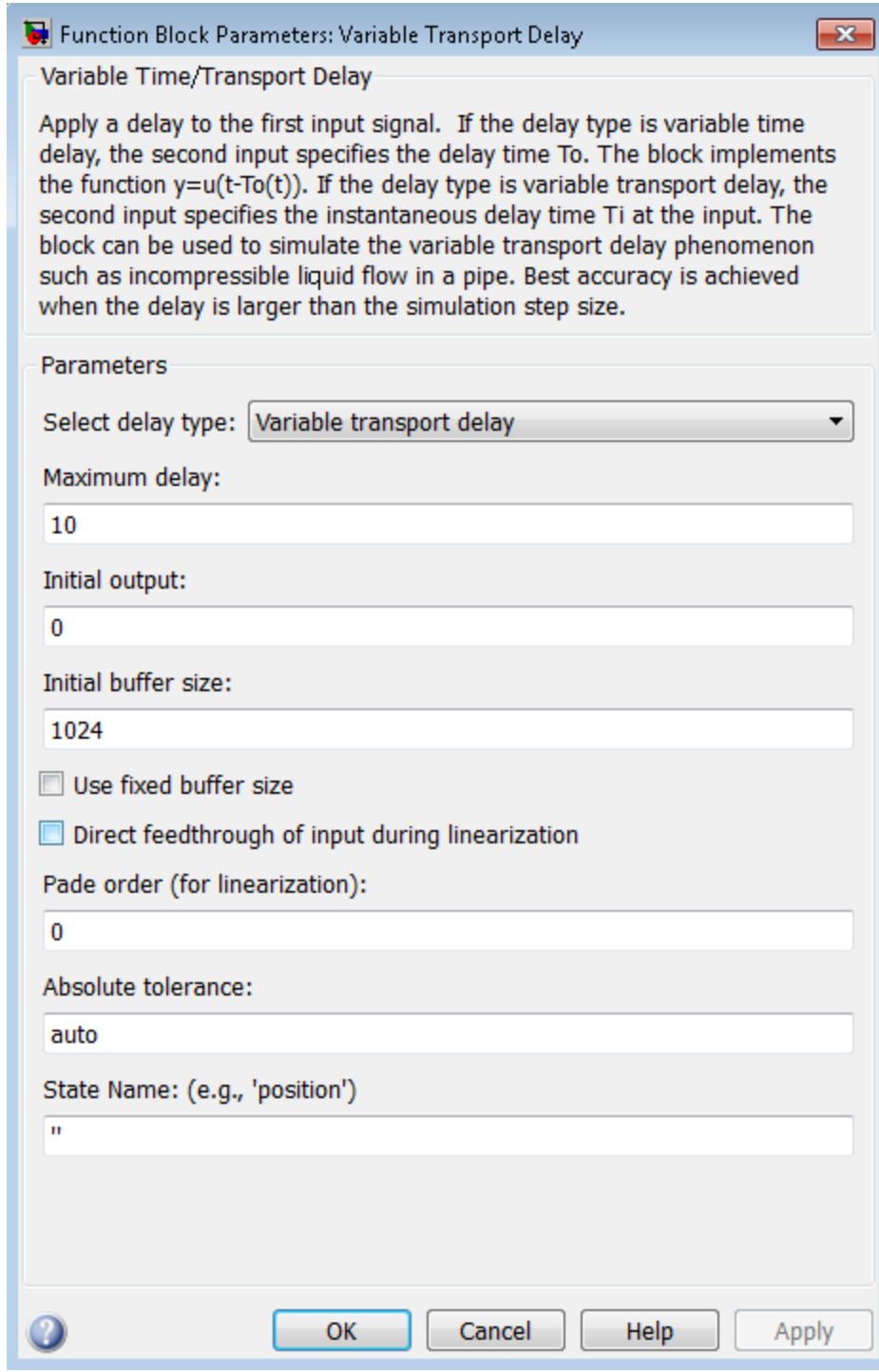
Parameters and Dialog Box

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.


Function Block Parameters: Variable Time Delay
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
OK
Cancel
Help
Apply

The dialog box for the Variable Transport Delay block appears as follows.


Function Block Parameters: Variable Transport Delay
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')
''
OK
Cancel
Help
Apply

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

Parameter: `VariableDelayType`
Type: string
Value: `'Variable transport delay' | 'Variable time delay'`
Default: `'Variable time delay'`

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

Parameter: `MaximumDelay`
Type: scalar or vector
Value: `'10'`
Default: `'10'`

Initial output

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Settings

Default:`Run-to-run tunable parameter`

A `Run-to-run tunable parameter` cannot be changed during simulation run time. However, changing it before a simulation begins will not cause Accelerator or Rapid Accelerator to regenerate code. Also, the initial output of this block cannot be `inf` or `NaN`.

Command-Line Information

Parameter: `InitialOutput`
Type: scalar or vector
Value: `'0'`
Default: `'0'`

Initial buffer size

Define the initial memory allocation for the number of input points to store. The input points define the history of the input signal up to the current simulation time.

Settings

Default: 1024

- If the number of input points exceeds the initial buffer size, the block allocates additional memory.
- After simulation ends, a message displays if the buffer is not sufficient and more memory needs to be allocated.

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

Parameter: `MaximumPoints`
Type: scalar
Value: `'1024'`
Default: `'1024'`

Use fixed buffer size

Specify use of a fixed-size buffer to save input data from previous time steps.

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 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

Parameter: `FixedBuffer`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `ZeroDelay`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Direct feedthrough of input during linearization

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

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

Parameter: `TransDelayFeedthrough`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `PadeOrder`
Type: string
Value: `'0'`
Default: `'0'`

Absolute tolerance

Specify the absolute tolerance for computing the block state.

Default: `auto`

- You can enter `auto`, `-1`, or a positive real scalar or vector.
- If you enter `auto`, or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute the block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Dependency

Setting **Select delay type** to `Variable transport delay` enables this parameter.

Command-Line Information

Parameter: `AbsoluteTolerance`
Type: string, scalar, or vector
Value: `'auto'` | `'-1'` | any positive real scalar or vector
Default: `'auto'`

State Name (e.g., 'position')

Assign a unique name to each state.

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.

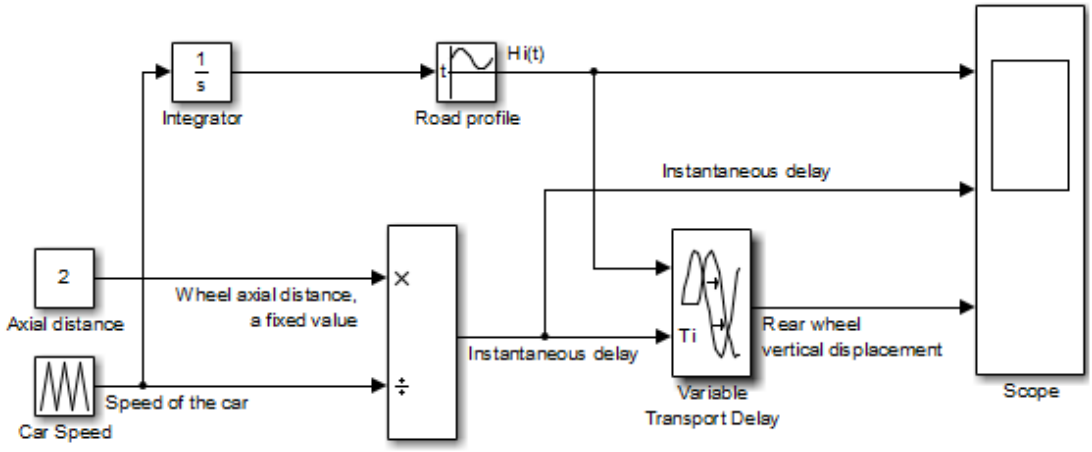
Command-Line Information

Parameter: `ContinuousStateAttributes`
Type: string
Value: `' '` | user-defined
Default: `' '`

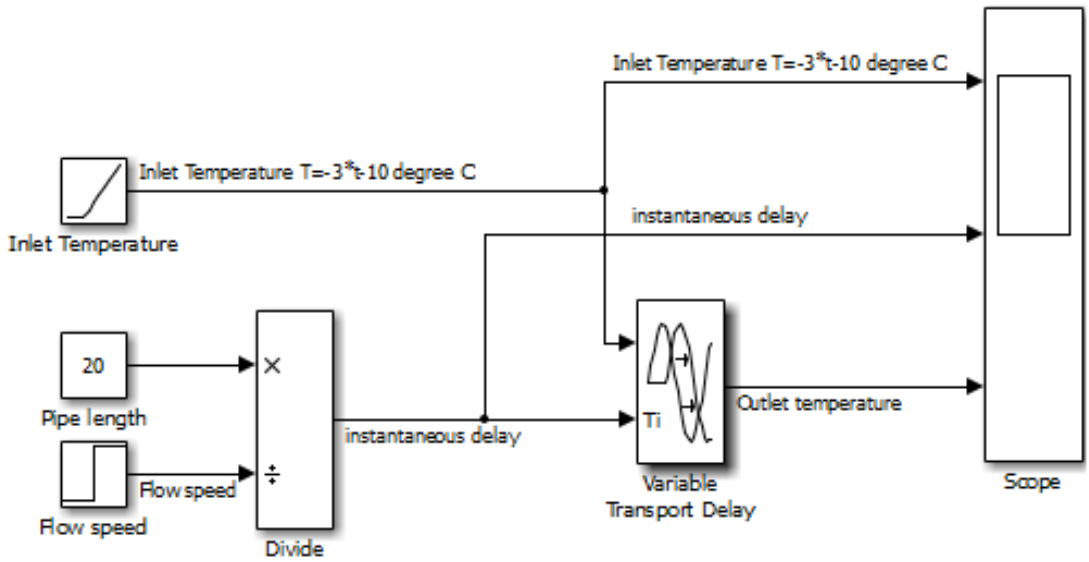
Examples

The `sldemo_VariableTransportDelay` and `sldemo_VariableTransportDelay_pipe` models show how you can use the Variable Transport Delay block.

The `sldemo_VariableTransportDelay` model shows how to model vertical wheel displacement on a one-dimensional car. The Variable Transport Delay block models the delay in vertical displacement of the rear wheel when the road profile changes:



The `sldemo_VariableTransportDelay_pipe` model shows how to model incompressible flow through a fixed-length pipe. The Variable Transport Delay block models the delay in temperature change at the outlet when fluid flow occurs:



Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Yes, of the time delay (second) input
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Transport Delay`

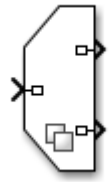
Introduced in R2007a

Variant Sink

Define variant choices as sinks without requiring subsystems

Library

Signal Routing



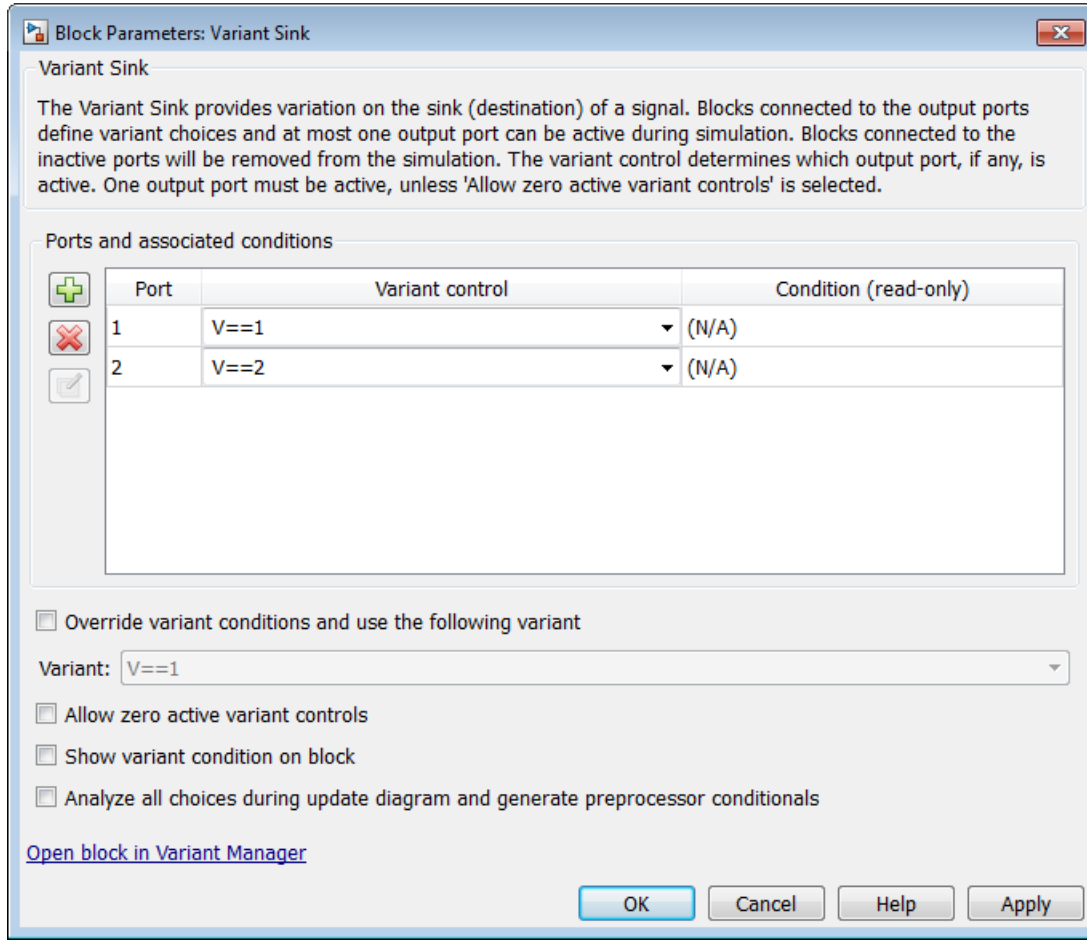
Description

The `Variant Sink` block has one input port and one or more output ports. You can define variant choices as blocks that are connected to the output port so that, at most, one choice is active.

Each output port is associated with a variant control. The variant control that evaluates to `true`, determines which output port is active.

During simulation, Simulink connects the active choice directly to the input port of the `Variant Sink` block and ignores the inactive choices.

Dialog Box and Parameters



Ports and associated conditions

A table of port choices, variant controls, and conditions. The **Variant control** that evaluates to `true` determines which input port must be active.

Settings

Default: The table has a row for each variant choice exiting in the Variant Sink block. See each column parameter for its default value:

- "Port" on page 1-2234
- "Variant Control" on page 1-2235
- "Condition (read-only)" on page 1-2236

See Also

- "Define, Configure, and Activate Variants"
- "Switch Between Variant Choices"

Port

Number of the input port that is connected to one variant choice upstream of the Variant Sink block

Settings

A read-only field, based on the input port that is connected to one variant choice upstream of the Variant Sink block.

Click  to add a port or  to delete an existing one.

Variant Control

Displays the variant controls available in the base workspace. The variant control can be a Boolean condition expression or a `Simulink.Variant` object representing a Boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.

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

Parameter: `VariantControls`
Type: cell array of strings
Value: Variant control that is associated with the variant choice.
Default: `''`

Condition (read-only)

Displays the **Condition** for the variant controls that are `Simulink.Variant` objects.

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

- "Create, Export, and Reuse Variant Controls"
- `Simulink.Variant`

Override variant conditions and use the following variant

Ignore the current variant conditions while determining the active variant choice. Instead, use the value of the **Variant** parameter as the variant control to determine the active variant choice.

If you do not select this option, Simulink determines the active variant choice based on the variant control that evaluates to `true`.

Settings

Default: Off

 On

Use the specified variant instead of selecting the active variant from the variant control.

 Off

Determine the active variant choice based on the variant control that evaluates to `true`.

Dependencies

This parameter enables **Variant**.

Command-Line Information

Parameter: `OverrideUsingVariant`
Type: string
Value: `''` if no overriding variant is specified.
Default: `''`

See Also

- "Activation States of Variant Choices"

Variant

Select the variant control that Simulink uses if you select **Override variant conditions and use the following variant**.

Settings

Default: `''`

The **Variant** drop-down list displays all variant controls that are currently defined in the base workspace or a data dictionary. Use valid MATLAB identifiers to specify variant controls.

Dependencies

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

- `Simulink.Variant`

Allow zero active variant controls

When you select this option, Simulink disables the `Variant Source` block when the variant condition activates no variant choice. This option is useful when you are prototyping variant choices.

If you do not select this option, Simulink generates an error when no variant choice is active.

Settings

Default: Disabled

Command-Line Information

Parameter: `AllowZeroVariantControls`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Show variant condition on block

When you select this option, Simulink annotates each variant choices with the condition that activates that choice.

Settings

Default: Disabled

Command-Line Information

Parameter: `ShowConditionOnBlock`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Analyze all choices during update diagram and generate preprocessor conditionals

When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`).

When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

Settings

Default: Disabled

Dependencies

- The check box is available for generating ERT targets only.
- **Override variant conditions and use following variant** is cleared (`'off'`)

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

See Also

- "Represent Variant Source and Sink Blocks in Generated Code"

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

- "Define, Configure, and Activate Variants"
- "Variant Systems"

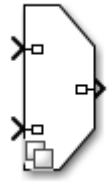
Introduced in R2016a

Variant Source

Define variant choices as sources without requiring subsystems

Library

Signal Routing



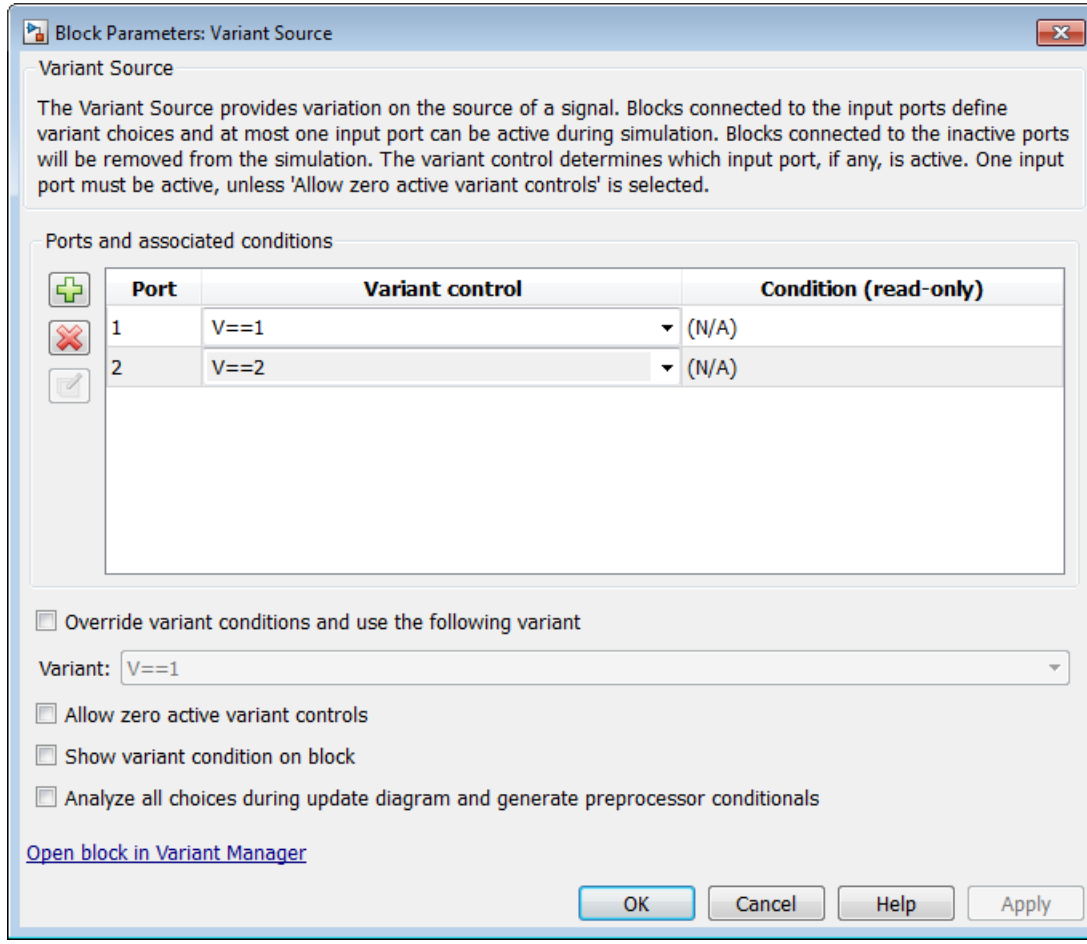
Description

The `Variant Source` block has one or more input ports and one output port. You can define variant choices as blocks that are connected to the input port so that, at most, one choice is active.

Each input port is associated with a variant control. The variant control that evaluates to `true`, determines which input port is active.

During simulation, Simulink connects the active choice directly to the output port of the `Variant Source` block and ignores the inactive choices.

Dialog Box and Parameters



Ports and associated conditions

A table of port choices, variant controls, and conditions. The **Variant control** that evaluates to `true` determines which input port must be active.

Settings

Default: The table has a row for each variant choice feeding into the Variant Source block. See each column parameter for its default value:

- "Port" on page 1-2242
- "Variant Control" on page 1-2243
- "Condition (read-only)" on page 1-2244

See Also

- "Define, Configure, and Activate Variants"
- "Switch Between Variant Choices"

Port

Number of the input port that is connected to one variant choice upstream of the Variant Source block

Settings

A read-only field, based on the input port that is connected to one variant choice upstream of the Variant Source block.

Click  to add a port or  to delete an existing one.

Variant Control

Displays the variant controls available in the base workspace. The variant control can be a Boolean condition expression or a `Simulink.Variant` object representing a Boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.

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

Parameter: `VariantControls`
Type: cell array of strings
Value: Variant control that is associated with the variant choice.
Default: `''`

Condition (read-only)

Displays the **Condition** for the variant controls that are `Simulink.Variant` objects.

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

- "Create, Export, and Reuse Variant Controls"
- `Simulink.Variant`

Override variant conditions and use the following variant

Ignore the current variant conditions while determining the active variant choice. Instead, use the value of the **Variant** parameter as the variant control to determine the active variant choice.

If you do not select this option, Simulink determines the active variant choice based on the variant control that evaluates to `true`.

Settings

Default: Off

 On

Use the specified variant instead of selecting the active variant from the variant control.

 Off

Determine the active variant choice based on the variant control that evaluates to `true`.

Dependencies

This parameter enables **Variant**.

Command-Line Information

Parameter: `OverrideUsingVariant`
Type: string
Value: `''` if no overriding variant is specified.
Default: `''`

See Also

- "Activation States of Variant Choices"

Variant

Select the variant control that Simulink uses if you select **Override variant conditions and use the following variant**.

Settings

Default: `''`

The **Variant** drop-down list displays all variant controls that are currently defined in the base workspace or a data dictionary. Use valid MATLAB identifiers to specify variant controls.

Dependencies

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

- `Simulink.Variant`

Allow zero active variant controls

When you select this option, Simulink disables the `Variant Source` block when the variant condition activates no variant choice. This option is useful when you are prototyping variant choices.

If you do not select this option, Simulink generates an error when no variant choice is active.

Settings

Default: Disabled

Command-Line Information

Parameter: `AllowZeroVariantControls`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Show variant condition on block

When you select this option, Simulink annotates each variant choices with the condition that activates that choice.

Settings

Default: Disabled

Command-Line Information

Parameter: `ShowConditionOnBlock`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Analyze all choices during update diagram and generate preprocessor conditionals

When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`).

When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

Settings

Default: Disabled

Dependencies

- The check box is available for generating ERT targets only.
- **Override variant conditions and use following variant** is cleared (`'off'`)

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

See Also

- "Represent Variant Source and Sink Blocks in Generated Code"

Characteristics

Data Types	Double | Single | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

- "Define, Configure, and Activate Variants"
- "Variant Systems"

Introduced in R2016a

Variant Subsystem

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.

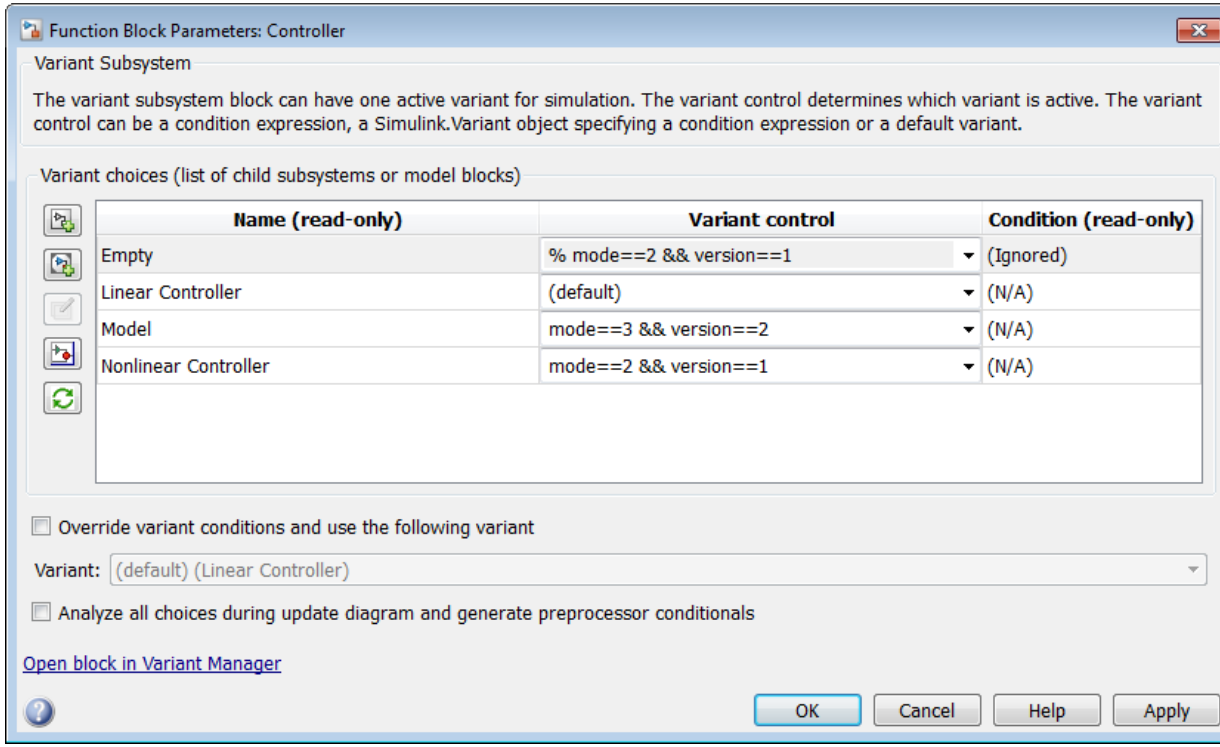
Mapping Inports and Outports

Each subsystem block within a Variant Subsystem represents one variant configuration. These subsystem blocks can have different numbers of inports and outports than their parent variant subsystem, provided the following conditions are satisfied.

- The inport names on a variant are a subset of the inport names used by the parent variant subsystem.
- The outport names on a variant are a subset of the outport names used by the parent variant subsystem.

During simulation, Simulink disables the inactive ports in a variant subsystem block.

Parameters and Dialog Box



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 1-2253
- "Variant Control" on page 1-2254
- "Condition (read-only)" on page 1-2255

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 and add a new model variant choice: Places a new model choice in the table and creates a model 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**.	
Open selected variant choice block: 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

- “Define, Configure, and Activate Variants”
- “Switch Between Variant Choices”

Name (read-only)

Name of the subsystem or model variant choice (contained in the Variant Subsystem block)

Settings

A read-only field, based on the subsystem or model variant choice contained in the Variant Subsystem block.

Click  to add a subsystem variant choice or  to add a model variant choice to the Variant Subsystem block.

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

To programmatically change this parameter, access the `VariantControl` parameter of the child subsystem or model inside the parent variant subsystem. For example, consider a `Variant Subsystem` block `LeftController` that contains two variant choices: `Linear` and `Nonlinear`. To edit the variant control for `Linear`, run the following command:

```
set_param('rtwdemo_preprocessor_subsys/LeftController/Linear',...
'VariantControl','NONLINEAR');
```

Structure field: Represented by the read-only `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

- "Create, Export, and Reuse Variant Controls"
- `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

- “Activation States of Variant Choices”

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

- `Simulink.Variant`

Analyze all choices during update diagram and generate preprocessor conditionals

When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`).

When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

Settings

Default: Disabled

Dependencies

- The check box is available for generating only ERT targets.
- **Override variant conditions and use following variant** is cleared (`'off'`)

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

See Also

- “Define, Configure, and Activate Variants”
- “Variant Systems”

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

Introduced in R2010b

Vector Concatenate, Matrix Concatenate

Concatenate input signals of same data type to create contiguous output signal

Library

Math Operations, Signal Routing

Description

The Concatenate block concatenates the signals at its inputs to create an output signal whose elements reside in contiguous locations in memory.

Tip The Concatenate block is useful for creating an output signal that is nonvirtual. However, to create a vector of function calls, use a `Mux` block instead.

You use a Concatenate block to define an array of buses. For details about defining an array of buses, see "Combine Buses into an Array of Buses".

The Concatenate block operates in either vector or multidimensional array concatenation mode, depending on the setting of its **Mode** parameter. In either case, the block concatenates the inputs from the top to bottom, or left to right, input ports.

Vector Mode

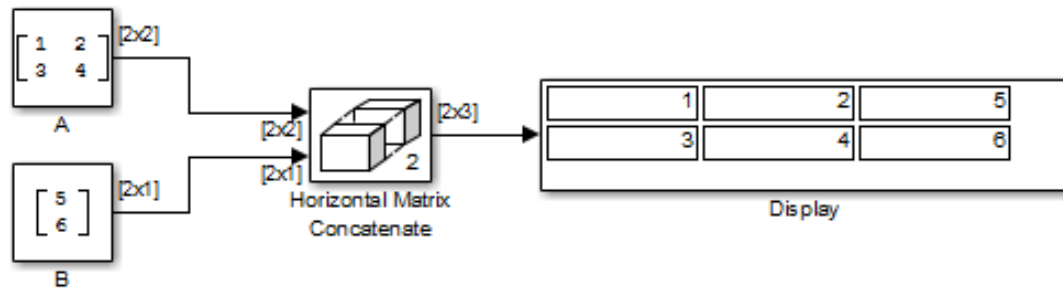
In vector mode, all input signals must be either vectors or row vectors [1xM matrices] or column vectors [Mx1 matrices] or a combination of vectors and either row or column vectors. The output is a vector if all inputs are vectors.

The output is a row or column vector if any of the inputs are row or column vectors, respectively.

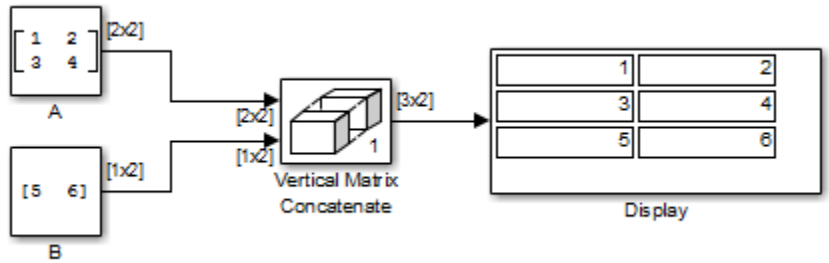
Multidimensional Array Mode

Multidimensional array mode accepts vectors and arrays of any size. It assumes that the trailing dimensions are all ones for input signals with lower dimensionality. For example, if the output is 4-D and the input is `[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:

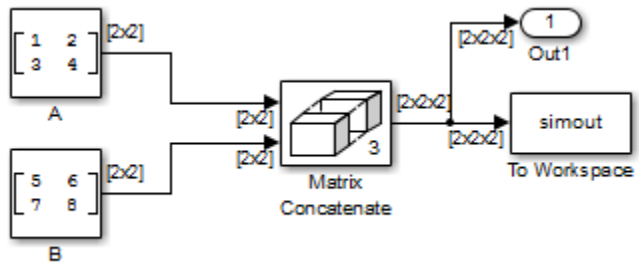


If you set the **Concatenate dimension** parameter to `1` and inputs are 2-D matrices, the block performs vertical matrix concatenation and stacks the input matrices on top of each other to create the output matrix, for example:



For horizontal concatenation, the input matrices must have the same column dimension. For vertical concatenation, the input matrices must have the same row dimension. All input signals must have the same dimension for all dimensions other than the concatenation dimensions.

If you set the **Mode** parameter to `Multidimensional array`, the **Concatenate dimension** parameter to `3`, and the inputs are 2-D matrices, the block performs multidimensional matrix concatenation, for example:



Data Type Support

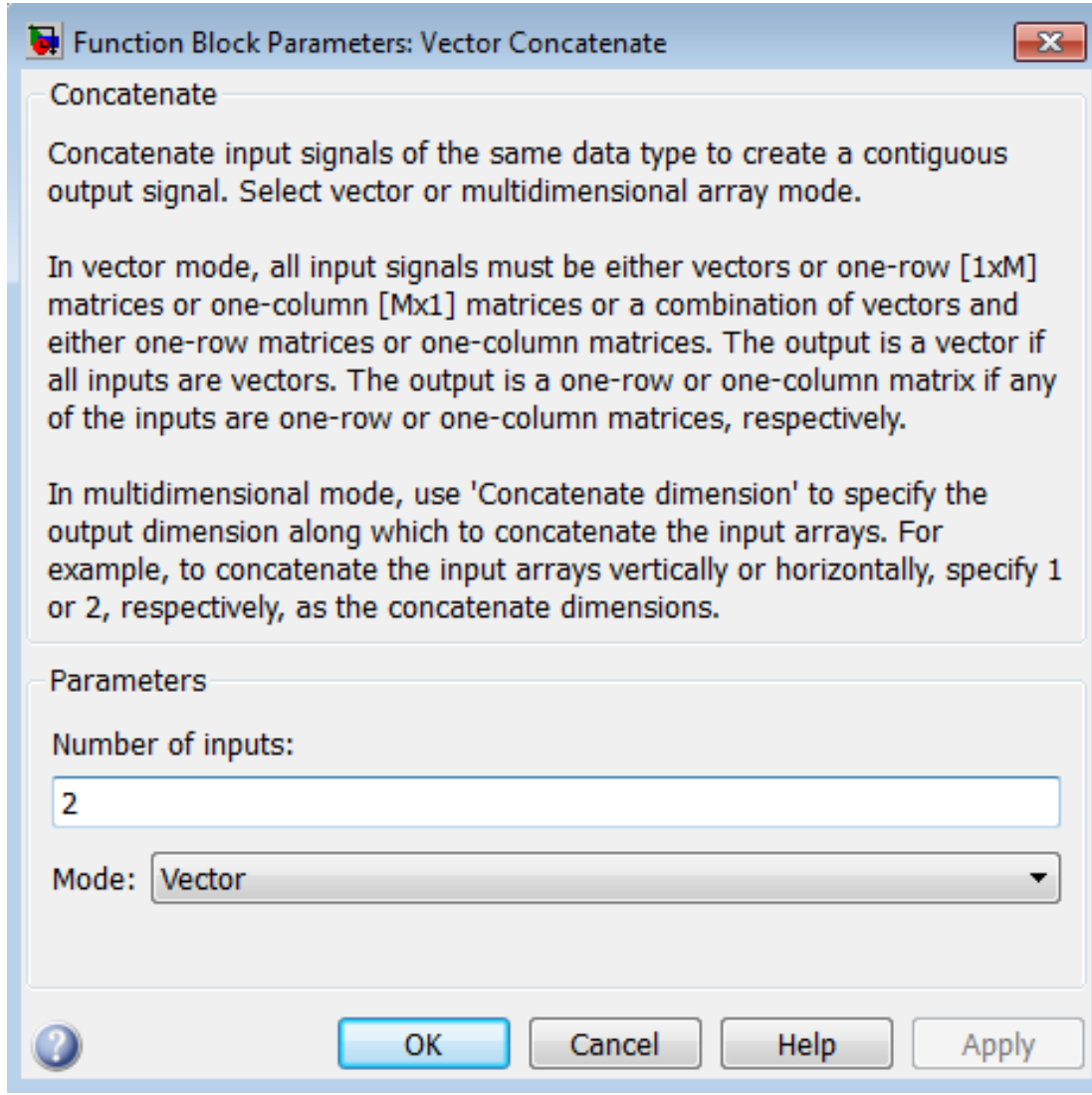
Accepts signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types. All inputs must be of the same data type. Outputs have the same data type as the input.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

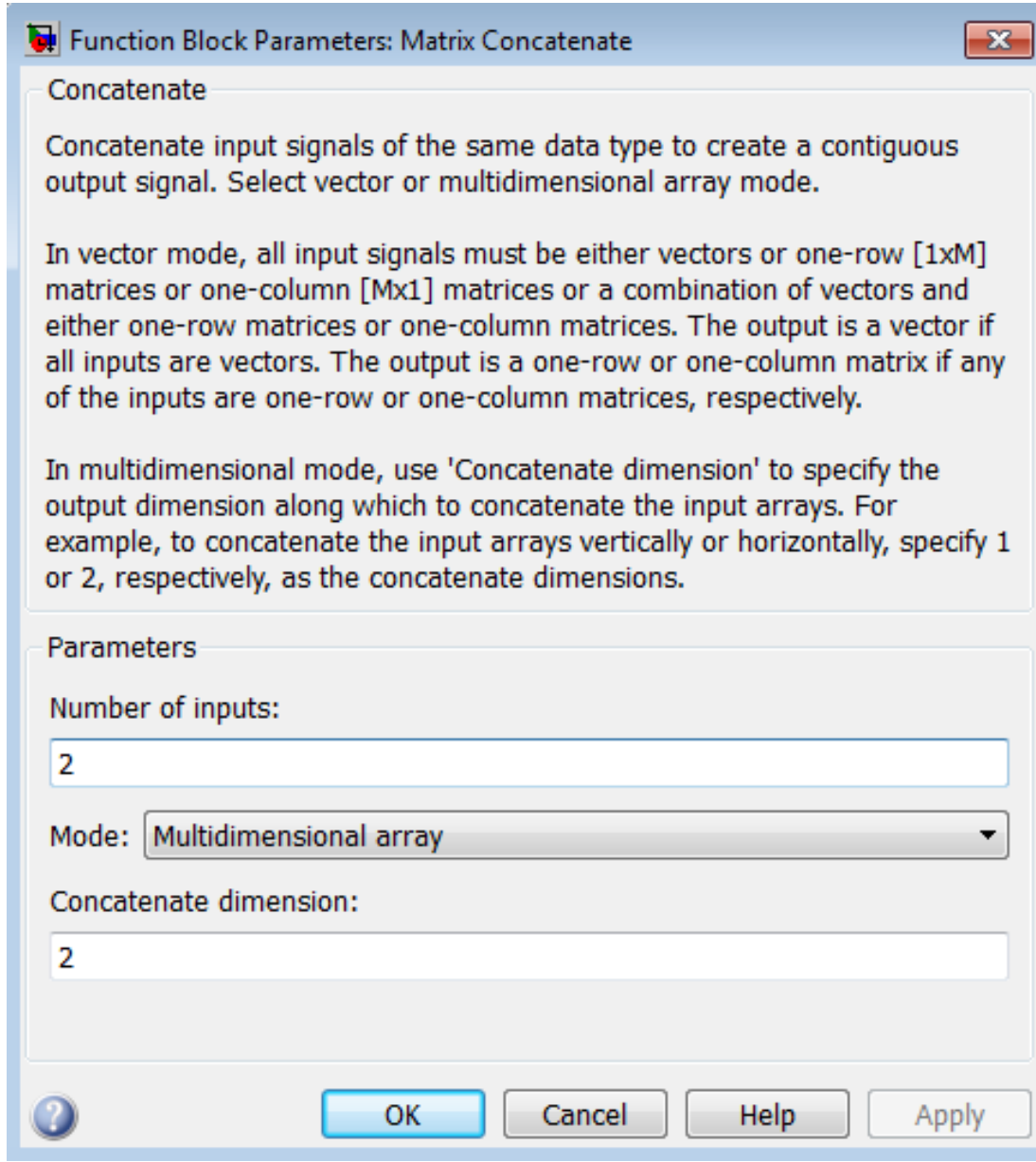
Parameters and Dialog Box

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.



The dialog box for the Matrix Concatenate block appears as follows.


Function Block Parameters: Matrix 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:
Multidimensional array
Concatenate dimension:
2
OK
Cancel
Help
Apply

Number of inputs

Specifies the number of inputs for the block.

Settings

Default: 2

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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 1-2260 for details).

`Multidimensional array`

Perform matrix concatenation (see "Multidimensional Array Mode" on page 1-2261 for details).

Dependency

This parameter enables **Concatenate dimension**.

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 6-101 for the command-line information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`cat` in the MATLAB reference documentation

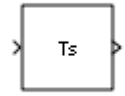
Introduced in R2009b

Weighted Sample Time

Support calculations involving sample time

Library

Signal Attributes



Description

The Weighted Sample Time block is an implementation of the Weighted Sample Time Math block. See Weighted Sample Time Math for more information.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Waveform Generator

Output waveforms using signal notations

Library

Sources

Description 

The Waveform Generator block outputs waveforms based on signal notations that you enter in the **Waveform Definition** table.

This block supports these syntaxes for the signal notations:

- Function syntax — Specify all arguments in the specific order for the signal syntax (see "Supported Waveforms" on page 1-2270).
- Name-value syntax — Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. For more information, see "Supported Waveforms" on page 1-2270.

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Supported Operators

Operation	Operator
Absolute value	`abs()`
Addition	`+`
Division	`/`

Operation	Operator
Multiplication	*
Parentheses	( )
Subtraction	-
Unary minus	-

The Waveform block observes the following rules of operator precedence:

1 ( )

2 + - (unary)

3 * /

4 + -

Supported Waveforms

Enter signal notations in the **Waveform Definition** table, one waveform definition per line. To add a waveform definition, click **Add**. The new waveform appears as an empty string. The block interprets empty strings or white space strings as ground.

To remove a waveform definition, click **Remove**. You can select multiple waveforms using **Ctrl+**click or **Shift+**click.

- "Constant" on page 1-2270
- "Gaussian Noise" on page 1-2271
- "Pulse" on page 1-2272
- "Sawtooth" on page 1-2273
- "Sine Wave" on page 1-2274
- "Square" on page 1-2276
- "Step" on page 1-2277

Constant

Constant values can be:

- Numbers
- Workspace variables
  - Scalar, real variables only
- Built-in MATLAB constant, `pi`

Examples

- `1`
- `1.1`
- `x`
- `pi`

Gaussian Noise

Syntax

```
gaussian(mean,variance,seed)
```

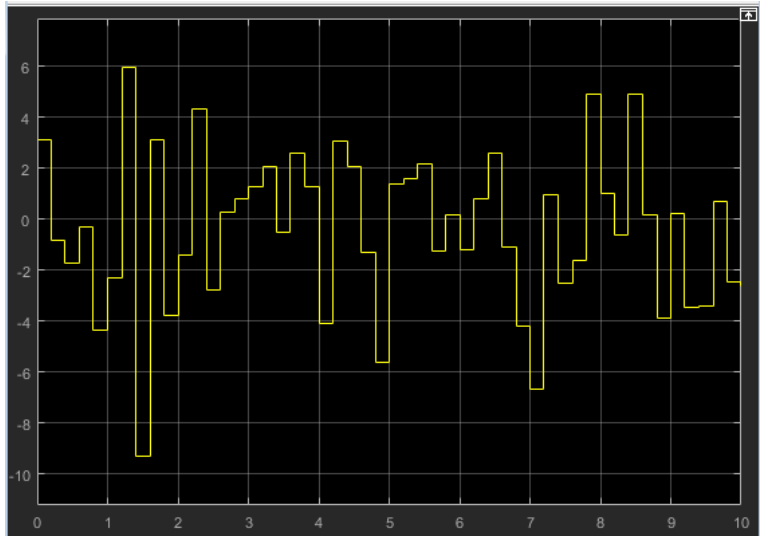
```
gaussian('Mean',mean,'Variance',variance,'Seed',seed)
```

Input Arguments

- `mean` — Mean value of the random variable output.
  - Default: 0
- `variance` — Standard deviation squared of the random variable output.
  - Default: 1
  - Value: Positive constant or positive real scalar variable
- `seed` — Initial seed value for the random number generator.
  - Default: 0
  - Value: Constant or real scalar variable

Example

```
gaussian('Mean',0,'Variance',10,'Seed',1)
```



Pulse

Syntax

```
pulse(amplitude,trigger_time,duration)
```

```
pulse('Amplitude',amplitude,'TriggerTime',trigger_time,'Duration',duration)
```

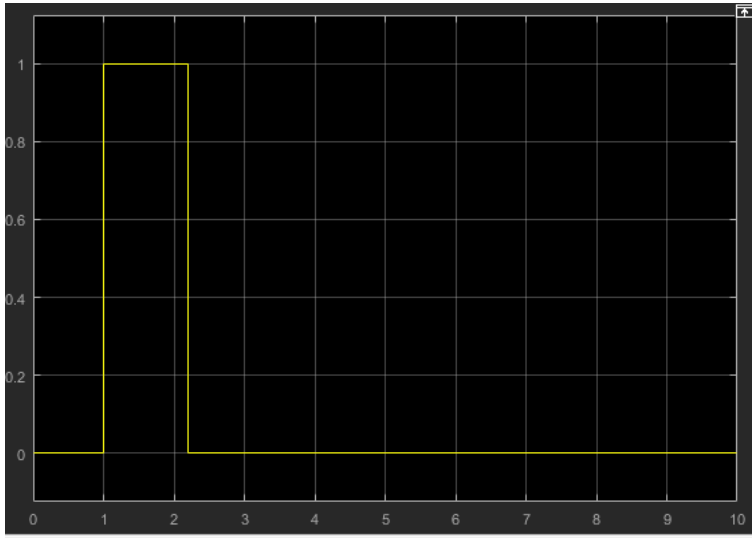
Input Arguments

- `amplitude` — Value of signal when pulse is high.
  - Default: 1
- `trigger_time` — Elapsed simulation time when signal changes to amplitude, in seconds.
  - Default: 1
  - Value: Constant or real scalar variable
- `duration` — How long the signal remains at the given amplitude before returning to ground, in seconds.
  - Default: 1

- Value: Positive constant or positive real scalar variable

Example

```
pulse('Amplitude',1,'TriggerTime',1,'Duration',1)
```



Sawtooth

Syntax

```
sawtooth(amplitude,frequency,phase_offset)
```

```
sawtooth('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_offset)
```

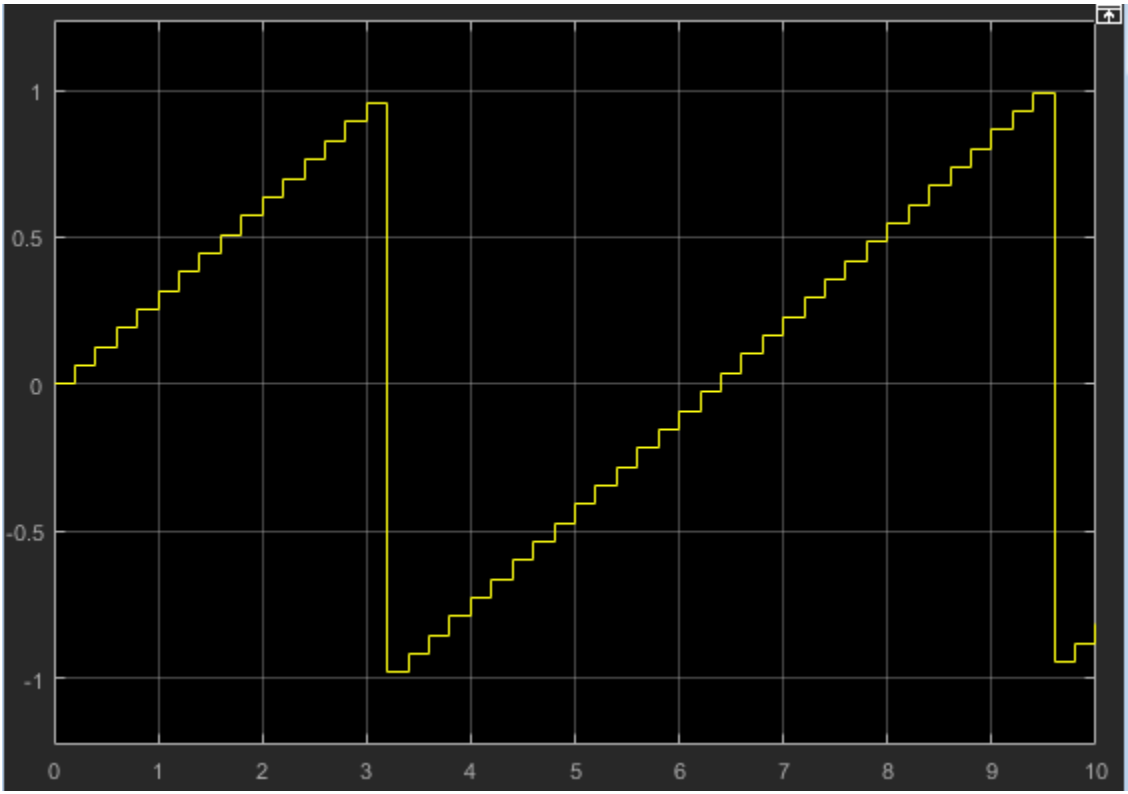
Input Arguments

- `amplitude` — Sawtooth peak value.
  - Default: 1
- `frequency` — Waveform frequency, in rad/s.
  - Default: 1

- `phase_offset` — Horizontal signal shift, based on elapsed simulation time, in seconds.
  - Default: 0

Example

```
sawtooth('Amplitude',1,'Frequency',1,'Phase',0)
```



Sine Wave

Syntax

```
sin(amplitude,frequency,phase_offset)
```

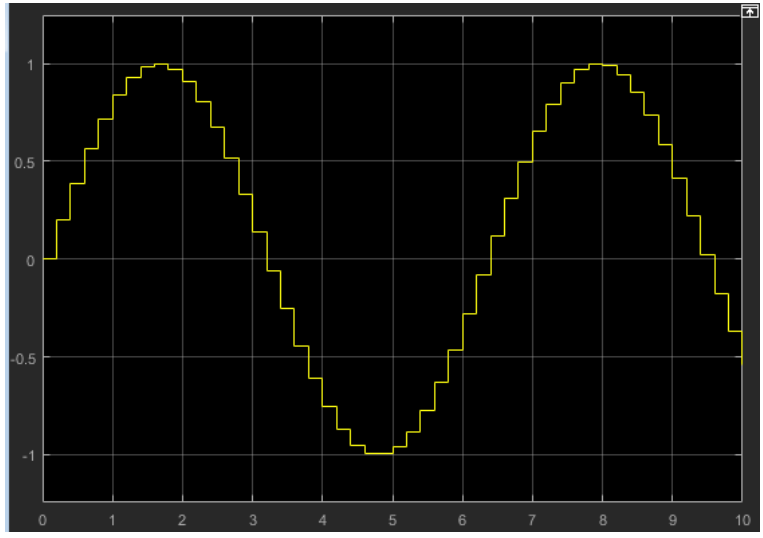
```
sin('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_offset)
```

Input Arguments

- `amplitude` — Amplitude of sine wave.
  - Default: 1
- `frequency` — Waveform frequency, in rad/s.
  - Default: 1
- `phase_offset` — Phase offset, in rads.
  - Default: 0

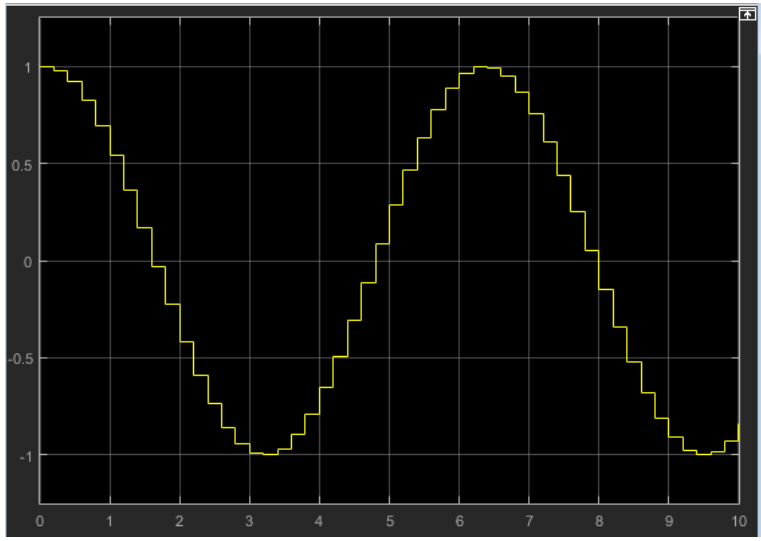
Examples

```
sin('Amplitude',1,'Frequency',1,'Phase',0)
```



To get the cosine waveform:

```
sin('Amplitude',1,'Frequency',1,'Phase',pi/2)
```



Square

Syntax

```
square(amplitude,frequency,phase_delay,duty_cycle)

square('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_delay,...
'DutyCycle',duty_cycle)
```

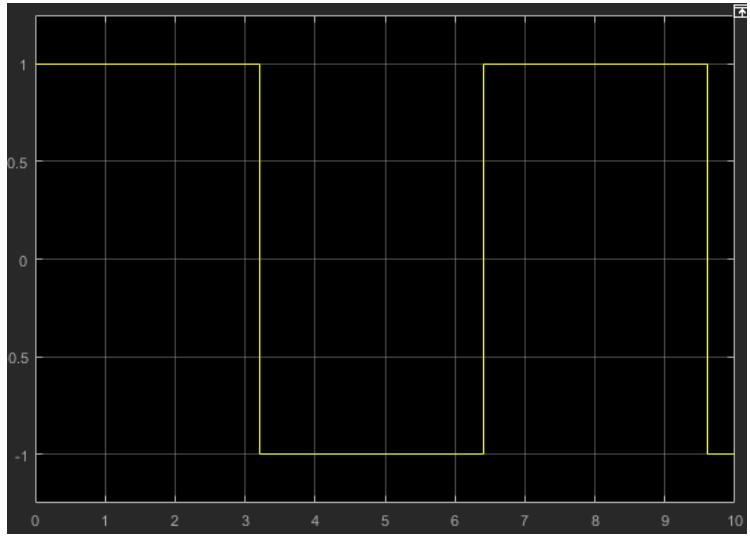
Input Arguments

- `amplitude` — Amplitude of signal.
  - Default: 1
- `frequency` — Waveform frequency in rad/s.
  - Default: 1
- `phase_delay` — Horizontal signal shift based on elapsed simulation time, in seconds.
  - Default: 0
- `duty_cycle` — Percentage of high signal per period (0–100%). The block clips the minimum signal to 0% and the maximum signal to 100%.

- Default: 50

Example

```
square('Amplitude',1,'Frequency',1,'Phase',0,'DutyCycle',50)
```



Step

Syntax

```
step(step_time,initial_value,final_value)

step('StepTime',step_time,'InitialValue',initial_value,'FinalValue',final_value)
```

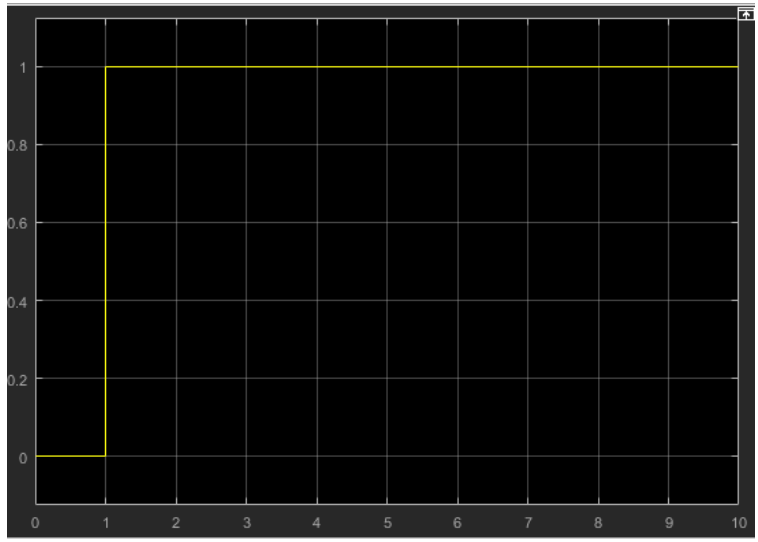
Input Arguments

- `step_time` — Elapsed simulation time when signal changes from `initial value` to `final value`, in seconds.
  - Default: 1
  - Value: Constant or positive real scalar variable.

- `initial_value` — Value of signal when elapsed simulation time is less than `step_time`, in seconds.
  - Default: 0
- `final_value` — Value of signal when elapsed simulation time is greater than or equal to step time, in seconds.
  - Default: 1

Example

```
step('StepTime',1,'InitialValue',0,'FinalValue',1)
```

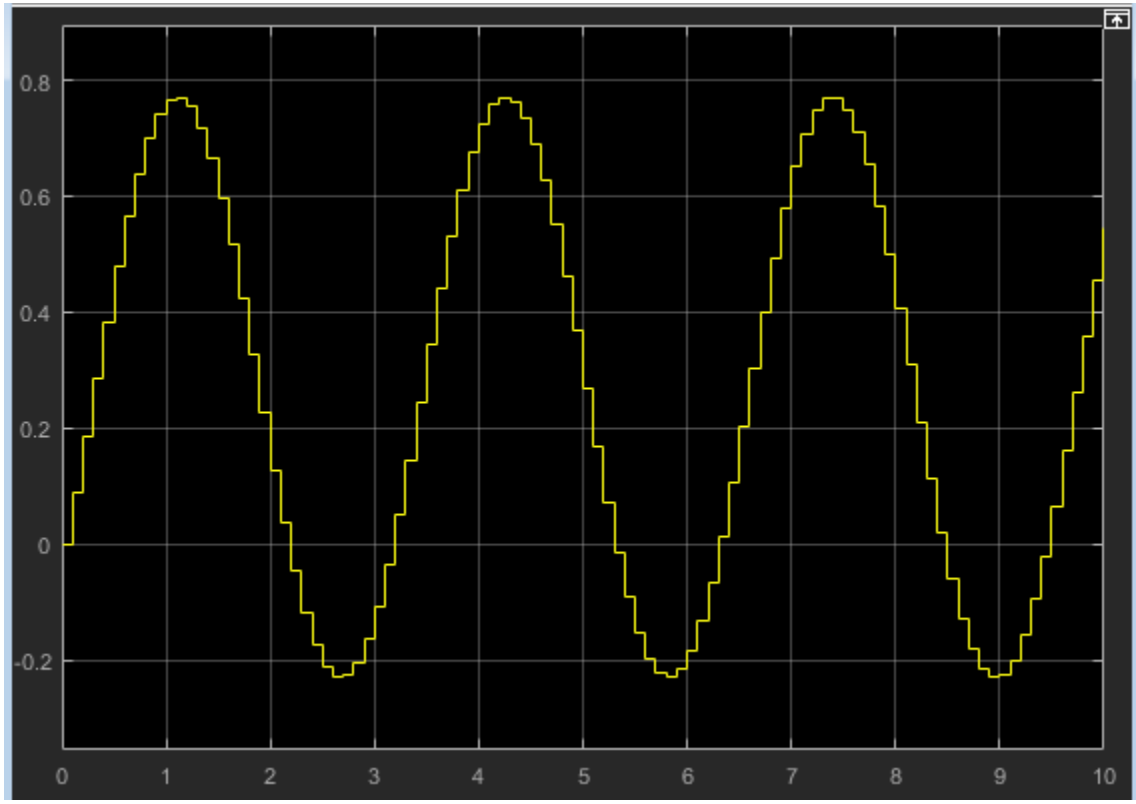


Supported Operations

The Waveform Generator block outputs one signal at a time. You can change this output signal. Express frequency and phase offset parameters in radians. You can also:

- Nest signal notations, for example:

  ```
  sin('Amplitude',sin('Amplitude',1,'Frequency',1,'Phase',0),'Frequency',1,'Phase',1)
  ```



- Reference real scalar variables in the base or model workspace, for example:

  ```
  sin('Amplitude',x,'Frequency',y,'Phase',z)
  ```

  x, *y*, and *z* exist in the base workspace.

For more information on waveforms, see “Supported Waveforms” on page 1-2270.

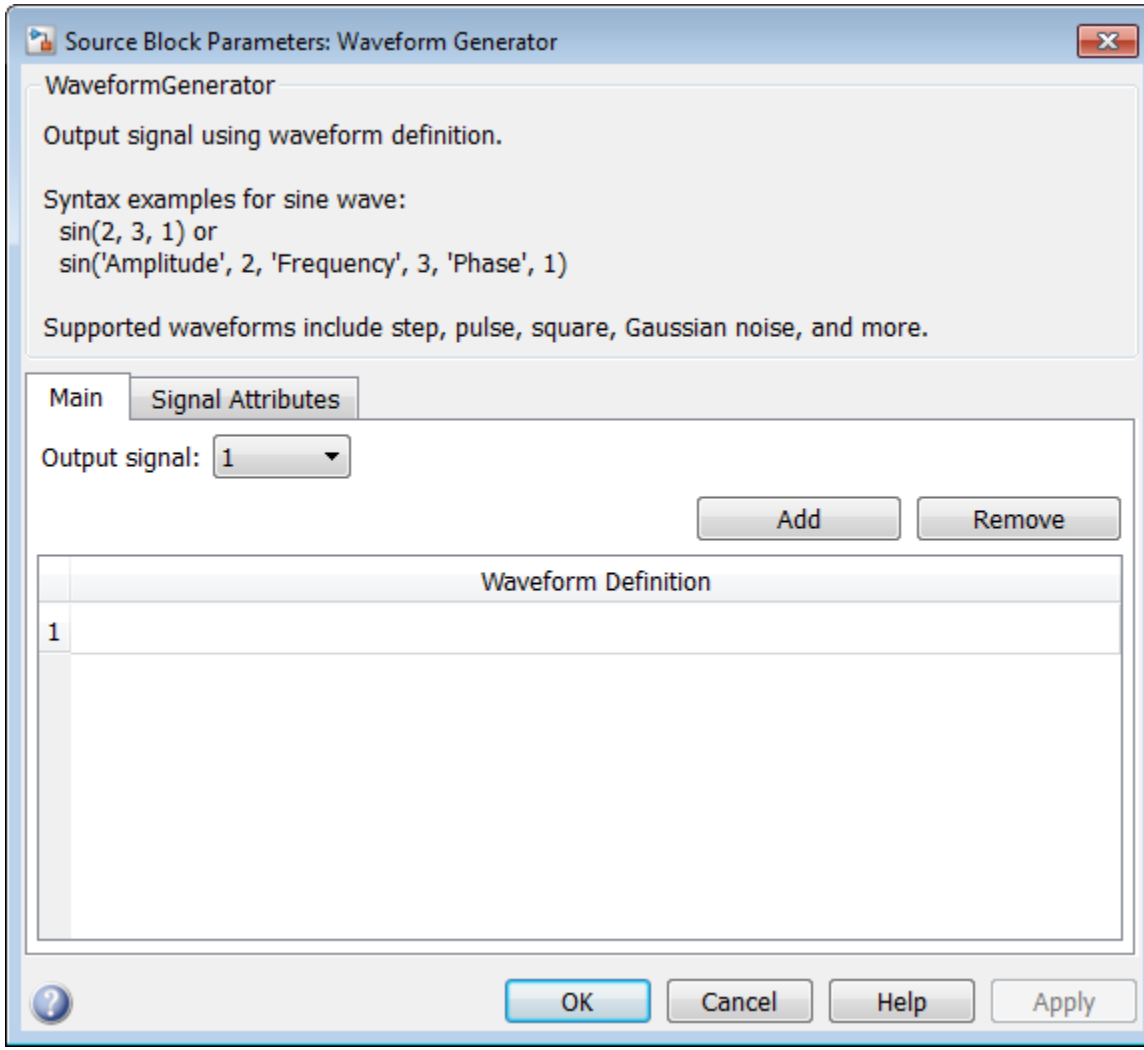
To quickly determine the response of a system to different types of inputs, you can vary the output signal of the Waveform Generator block while a simulation is in progress.

Data Type Support

The Waveform Generator block outputs scalar values. The default data type is double. However, the block also supports a large subset of Simulink data types, including fixed point (see “Output data type” on page 1-833).

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Output Signal

Select the signal. The number corresponds to the line item in the **Waveform Definition** table. You can change this parameter while a simulation is running.

Settings

Default: 1

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

Waveform Definition

Enter signal notations in the **Waveform Definition** table, one waveform definition per line. For more information on waveform definitions and how to enter them, see “Supported Waveforms” on page 1-2270.

Settings

Default: none

Output minimum

Lower value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMin`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output maximum

Upper value of the output range that Simulink checks.

Settings

Default: `[ ]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Control Block Parameter Value Ranges”) 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

Parameter: `OutMax`
Type: string
Value: `'[ ]'`
Default: `'[ ]'`

Output data type

Specify the output data type.

Settings

Default: `Inherit: Inherit via back propagation`

`Inherit: Inherit via back propagation`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type `int8` by a gain of `int16` and `ASIC/FPGA` is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified (assume 32-bit Generic)`, i.e., a generic 32-bit microprocessor, is specified as the target hardware, the output data type is `int32`. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a `Data Type Propagation` block. Examples of how to use this block are available in the Signal Attributes library `Data Type Propagation Examples` block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`double`

Output data type is `double`.

`single`

Output data type is `single`.

`int8`
: Output data type is `int8`.

`uint8`
: Output data type is `uint8`.

`int16`
: Output data type is `int16`.

`uint16`
: Output data type is `uint16`.

`int32`
: Output data type is `int32`.

`uint32`
: Output data type is `uint32`.

`boolean`
: Output data type is boolean.

`fixdt(1,16)`
: Output data type is fixed point `fixdt(1,16)`.

`fixdt(1,16,0)`
: Output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
: Output data type is fixed point `fixdt(1,16,2^0,0)`.

Command-Line Information

See “Block-Specific Parameters” on page 6-101 for the command-line information.

See Also

See “Control Signal Data Types” for more information.

Data type override

Specify data type override mode for this signal.

Settings

Default: `double`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

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.

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.

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**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**

Command-Line Information

See “Block-Specific Parameters” on page 6-101 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.

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.

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.

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

Parameter: `LockScale`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

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

Parameter: `RndMeth`
Type: string
Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
Default: `'Floor'`

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 the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “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

Parameter: `SaturateOnIntegerOverflow`
Type: string
Value: `'off' | 'on'`
Default: `'off'`

Sample time

Enter the time interval between sample time hits or specify `-1` to inherit the sample time.

Settings

Default: `0.1`

This value cannot be 0 (continuous) or inf (constant). For more information, see "Sample Time".

Command-Line Information

See "Block-Specific Parameters" on page 6-101 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 `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 6-101 for the command-line information.

See Also

See “Specify Data Types Using Data Type Assistant”.

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Characteristics

Data Types	See “Data Type Support” on page 1-2280
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced in R2015b

Weighted Sample Time Math

Support calculations involving sample time

Library

Math Operations

Description

The Weighted Sample Time Math block adds, subtracts, multiplies, or divides its input signal, `u`, by a weighted sample time, `Ts`. If the input signal is continuous, `Ts` is the sample time of the Simulink model. Otherwise, `Ts` is the sample time of the discrete input signal. If the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

You specify the math operation with the **Operation** parameter. 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

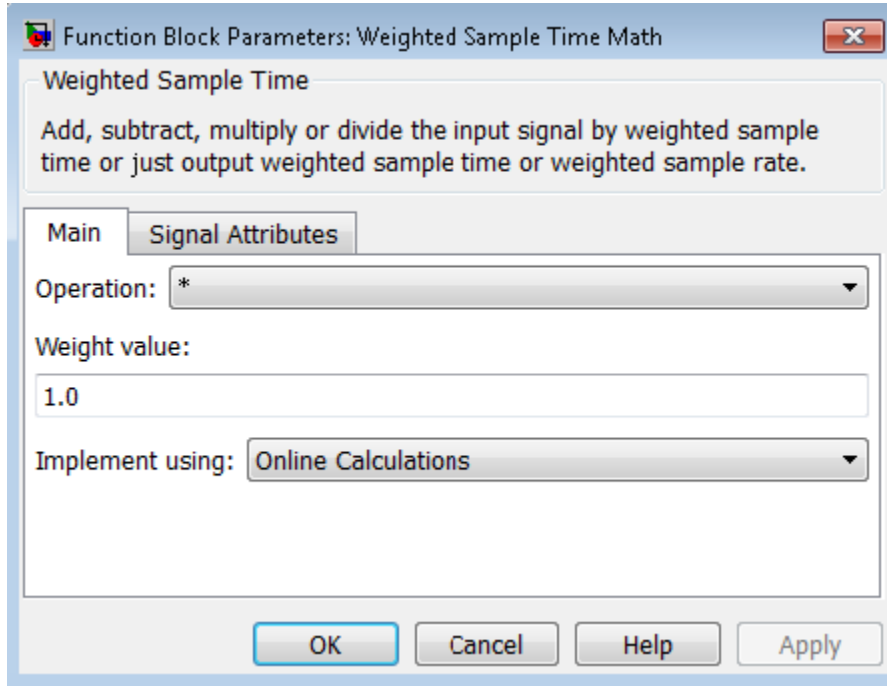
The Weighted Sample Time Math block accepts signals of the following data types:

- 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:



Operation

Specify operation to use: `+`, `-`, `*`, `/`, `Ts Only`, or `1/Ts Only`.

Weight value

Enter the weight of sample time.

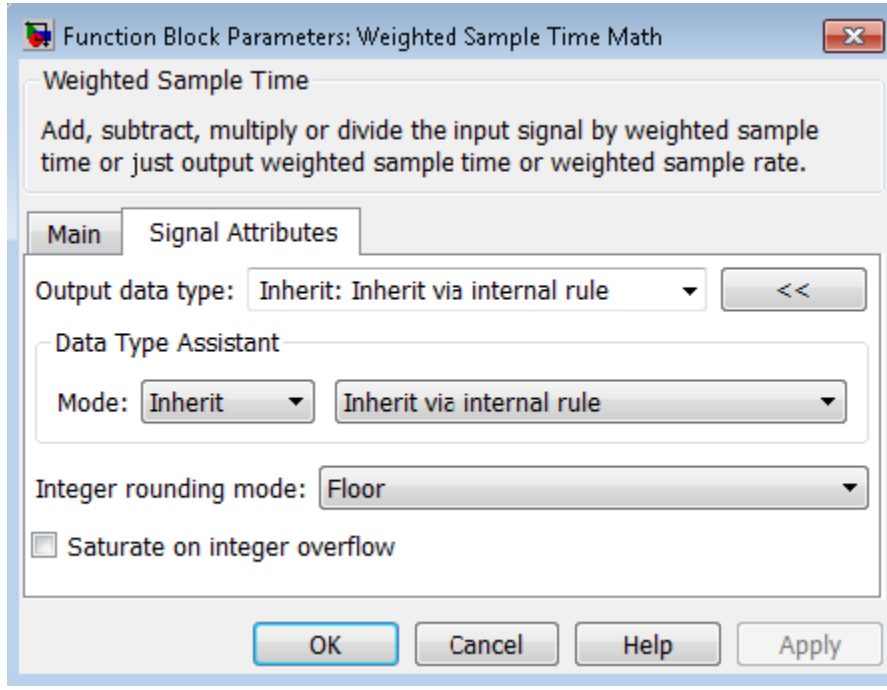
Implement using

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:



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.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	For all math operations except `Ts` and `1/Ts`
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

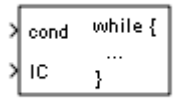
Introduced before R2006a

While Iterator

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.

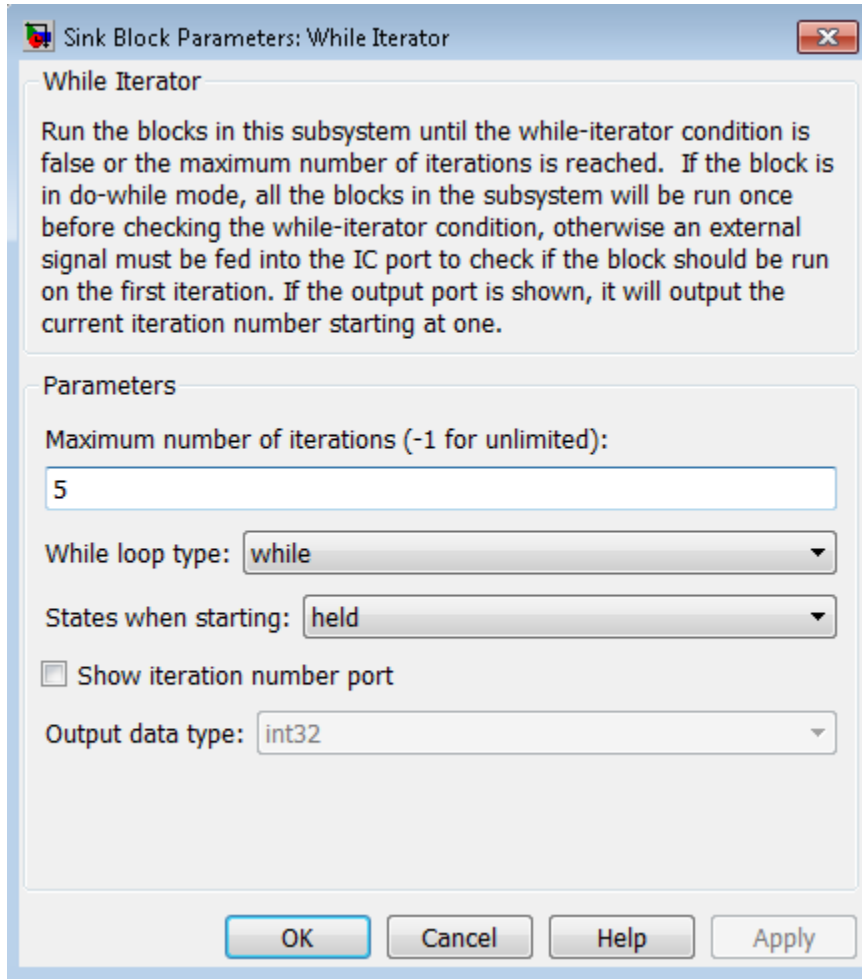
Tip Specify a maximum number of iterations to avoid infinite loops, which you can break only by terminating MATLAB.

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`.

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

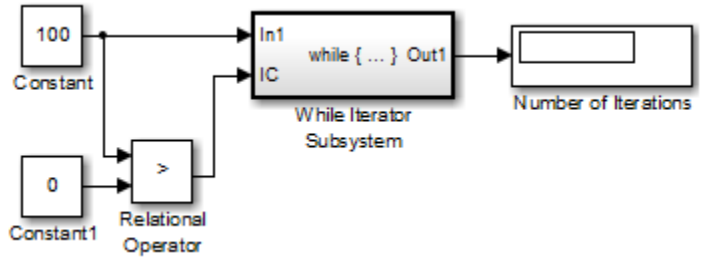
If you select this check box, the While Iterator block outputs its iteration value. This value starts at 1 and is incremented by 1 for each succeeding iteration. By default, this check box is not selected.

Output data type

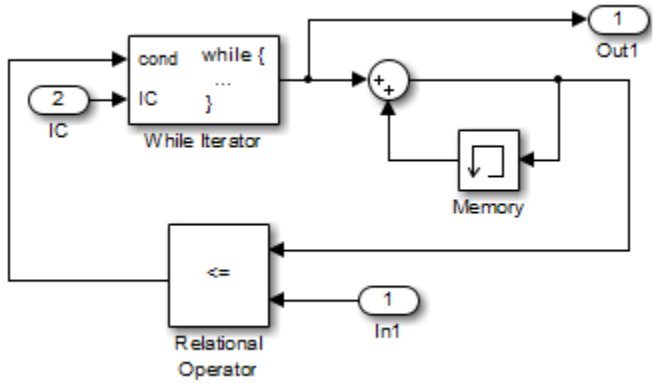
If you select the **Show iteration number port** check box (the default), this field is enabled. Use it to set the data type of the iteration number output to `int32`, `int16`, `int8`, or `double`.

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.



The contents of the `While Iterator` subsystem are:



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

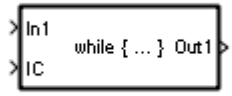
Introduced before R2006a

While Iterator Subsystem

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 “Use 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”.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

Introduced before R2006a

Width

Output width of input vector

Library

Signal Attributes

Description

The Width block generates as output the width of its input vector.

You can use an array of buses as an input signal to a Width block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Data Type Support

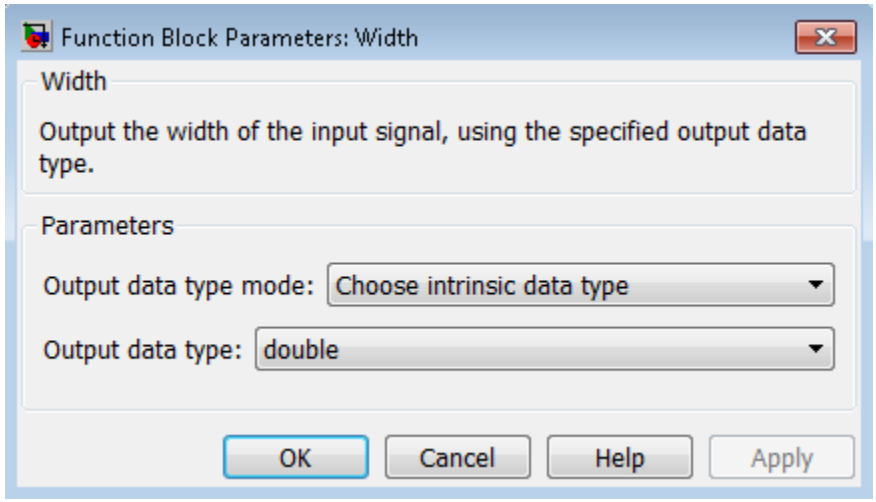
The Width block accepts real or complex signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The Width block also supports mixed-type signal vectors.

When the **Output data type mode** is not `Choose intrinsic data type`, the block supports only built-in numeric types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box



Note The Width block ignores the **Data type override** setting of the Fixed-Point Tool.

Output data type mode

Specify the output data type to be the same as the input, or inherit the data type by back propagation. You can also choose to specify a built-in data type from the drop-down list in the **Output data type** parameter.

Output data type

This parameter is visible when you select `Choose intrinsic data type` for **Output data type mode**. Select a built-in data type from the drop-down list.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Constant
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Wrap To Zero

Set output to zero if input is above threshold

Library

Discontinuities

Description

The Wrap To Zero block sets the output to zero when the input is above the **Threshold** value. However, the block outputs the input when the input is less than or equal to the **Threshold**.

Data Type Support

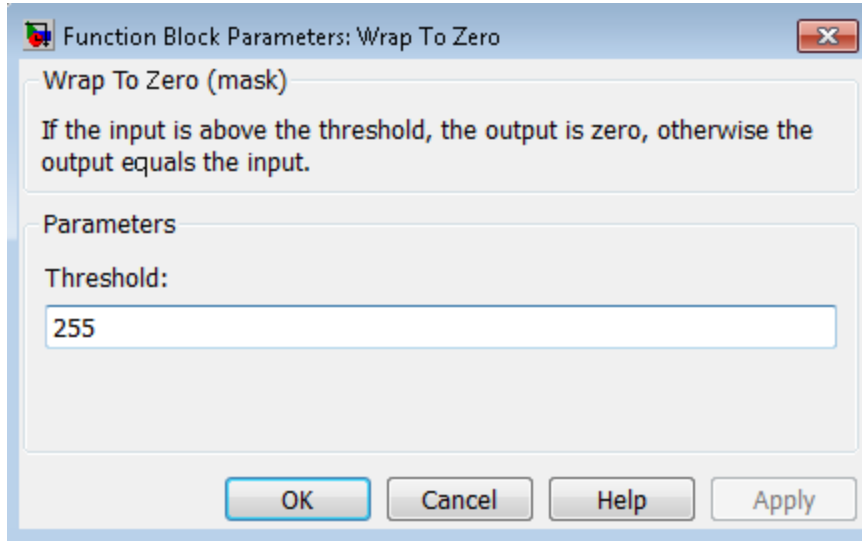
The Wrap To Zero block accepts inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The block output has the same data type as the input. For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Tip If the input data type cannot represent zero, parameter overflow occurs. To detect this overflow, go to the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box and set **Parameters > Detect overflow** to `warning` or `error`.

Parameters and Dialog Box



Threshold

When the input exceeds the threshold, the block sets the output to zero.

Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

XY Graph

Display X-Y plot of signals using MATLAB figure window

Library

Sinks

Description

The XY Graph block displays an X-Y plot of its inputs in a MATLAB figure window.

The block has two scalar inputs. The block plots data in the first input (the *x* direction) against data in the second input (the *y* direction). (See “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.

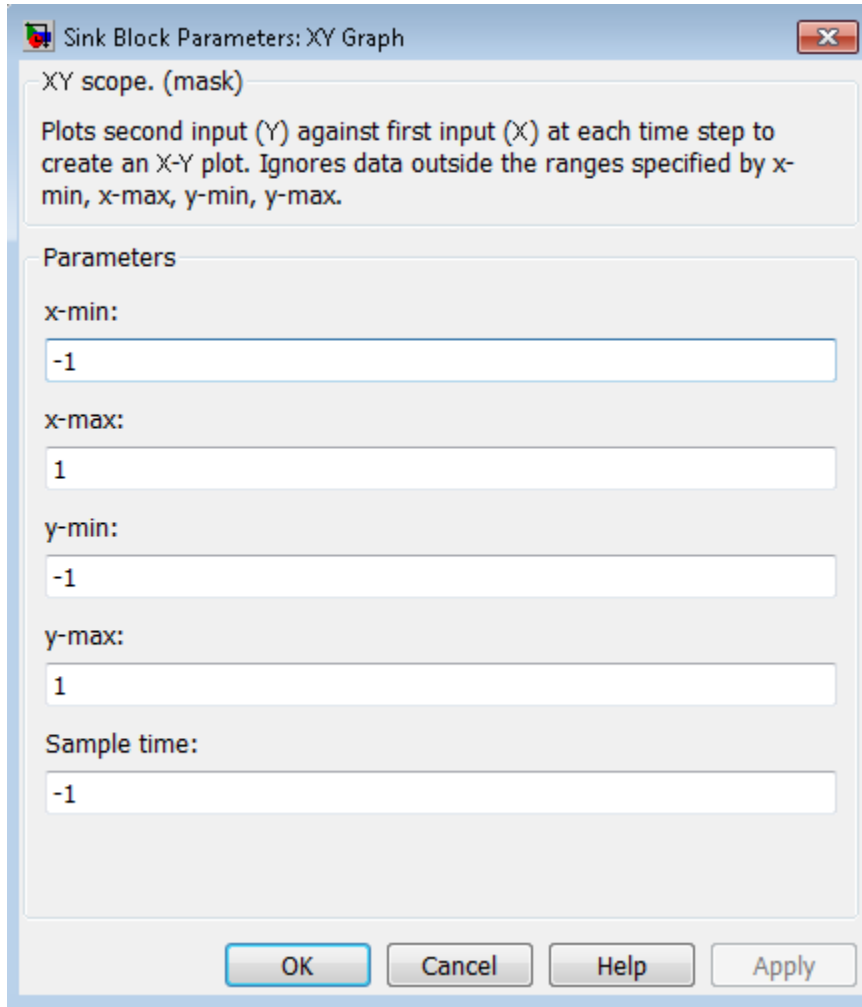
Data Type Support

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.

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

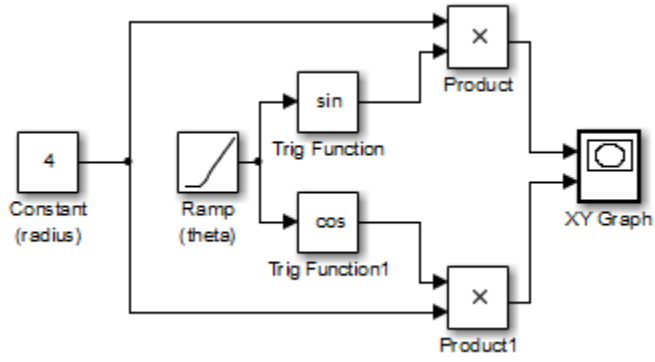
Specify the maximum *y*-axis value. The default is `1`.

Sample time

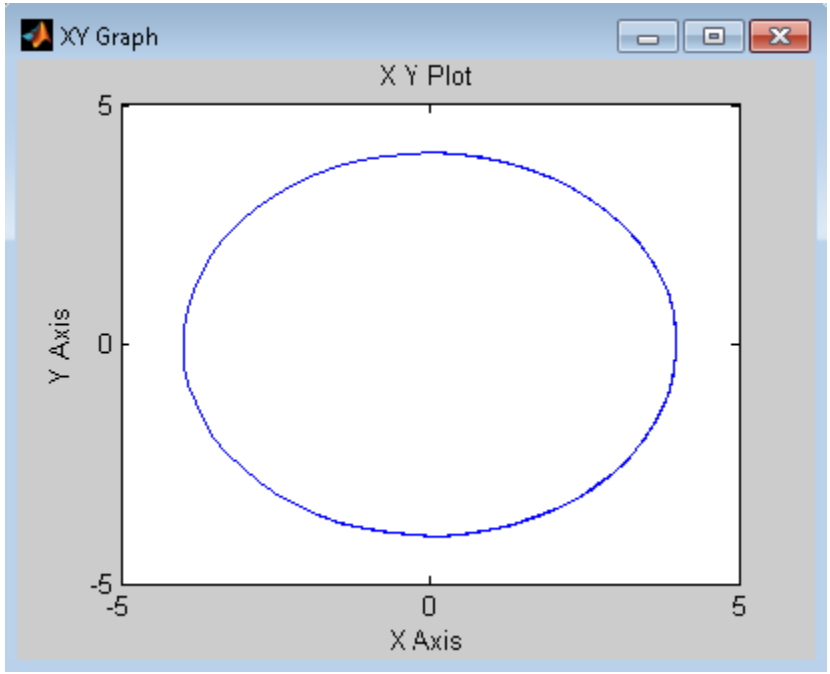
Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specify Sample Time" in the Simulink documentation for more information.

Examples

The following model computes the points that define a circle of radius 4, centered at the origin of the *x*-*y* plane.



When you simulate the model, a figure window appears.



Characteristics

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point
Sample Time	Specified in the **Sample time** parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	No

Introduced before R2006a

Zero-Order Hold

Implement zero-order hold of one sample period

Library

Discrete

Description

The Zero-Order Hold block holds its input for the sample period you specify. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds all elements of the vector for the same sample period.

You specify the time between samples with the **Sample time** parameter. A setting of `-1` means the block inherits the **Sample time**.

Tip Do not use the Zero-Order Hold block to create a fast-to-slow transition between blocks operating at different sample rates. Instead, use the `Rate Transition` block.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
`Unit Delay`	Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.	• `sldemo_enginewc` (Compression subsystem)
Memory	Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.	• `sldemo_bounce` • `sldemo_clutch` (Friction Mode Logic/Lockup FSM subsystem)
`Zero-Order Hold`	Convert an input signal with a continuous sample time to an output signal with a discrete sample time.	• `sldemo_radar_eml` • `aero_dap3dof`

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

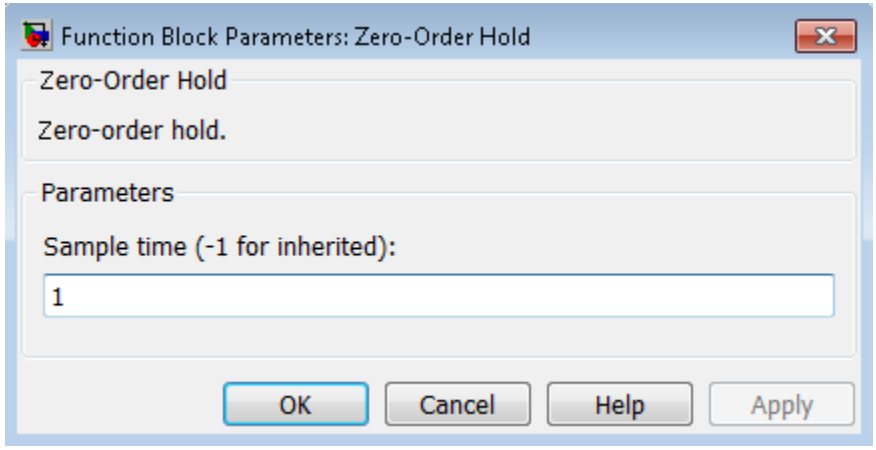
Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Support for state logging	Yes	No	No

Data Type Support

The Zero-Order Hold block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Parameters and Dialog Box



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 “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

Data Types	Double | Single | Boolean | Base Integer | Fixed-Point | Enumerated | Bus
Sample Time	Specified in the **Sample time** parameter
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Memory`, `Unit Delay`

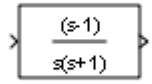
Introduced before R2006a

Zero-Pole

Model system by zero-pole-gain transfer function

Library

Continuous



Description

The Zero-Pole block models a system that you define with the zeros, poles, and gain of a Laplace-domain transfer function. This block can model single-input single output (SISO) and single-input multiple-output (SIMO) systems.

Conditions for Using This Block

The Zero-Pole block assumes the following conditions:

- The transfer function has the form

$$H(s) = K\frac{Z(s)}{P(s)} = K\frac{(s-Z(1))(s-Z(2))\ldots(s-Z(m))}{(s-P(1))(s-P(2))\ldots(s-P(n))},$$

  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**.

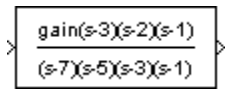
Each element of the output vector corresponds to a column in **Zeros**.

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.

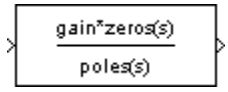
- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified zeros, poles, and gain. If you specify a variable in parentheses, the block evaluates the variable.

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.

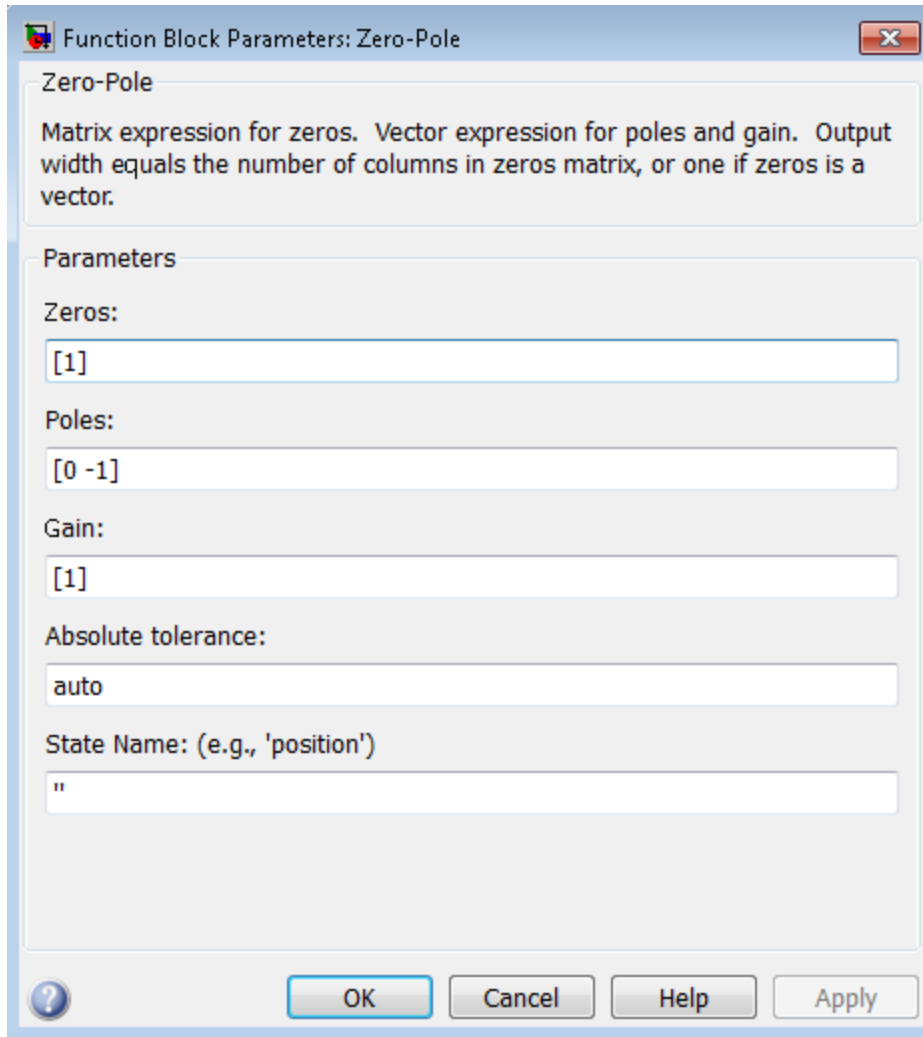
  For example, if you specify **Zeros** as `zeros`, **Poles** as `poles`, and **Gain** as `gain`, the block looks like this:



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.

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

Parameter: `Zeros`
Type: vector
Value: `'[1]'`
Default: `'[1]'`

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

Parameter: `Poles`
Type: vector
Value: `'[0 -1]'`
Default: `'[0 -1]'`

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

Parameter: `Gain`
Type: vector
Value: `'[1]'`
Default: `'[1]'`

Absolute tolerance

Specify the absolute tolerance for computing block states.

Settings

Default: `auto`

- You can enter `auto`, –1, a positive real scalar or vector.
- If you enter `auto` or –1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see "Solver Pane") to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Command-Line Information

Parameter: `AbsoluteTolerance`
Type: string, scalar, or vector
Value: `'auto' | '-1' |` any positive real scalar or vector
Default: `'auto'`

State Name (e.g., 'position')

Assign a unique name to each state.

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

Parameter: `ContinuousStateAttributes`
Type: string
Value: `' '` | user-defined
Default: `' '`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Only if the lengths of the **Poles** and **Zeros** parameters are equal

Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

`Discrete Zero-Pole`

Introduced before R2006a

Functions — Alphabetical List

add_block

Add block to model

Syntax

```
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')
add_block('built-in/Note', 'path/text', 'Position', position_array)
add_block('built-in/Area', 'path/text', 'Position', rect_array)
add_block('built-in/Image', 'path/text', 'Position', position_array,
'imagePath', path_to_image)
```

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. The new block appears in front of any blocks that it overlaps. If the `'src'` block is a Subsystem block, then `add_block` copies all the blocks in the subsystem.

Note: Load the models for the *'src'* and *'dest'* block paths before using the `add_block` command. If the *'src'* block is in a custom library, load the library first using `load_system` or `open_system`.

`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.

`add_block('built-in/Note', '`*`path`*`/`*`text`*`', 'Position', `*`position_array`*`)` creates an annotation in a Simulink model or a note in a Stateflow chart. The first part of the `path/text` argument is the path of the model or chart where you want the annotation or note. Append to the path a slash (/), followed by the text for the annotation or note. The `position_array` argument is a 1x4 array, specified as `[left, top, right, bottom]`, that gives the position of the upper-left corner of the annotation or note in pixels, relative to the upper left corner of the model or chart. If the annotation is autosized (the default), you can specify a 1x2 array `[left top]`. Positive x and y values are to the right of and down from the origin, respectively.

`add_block('built-in/Area', '`*`path`*`/`*`text`*`', 'Position', `*`rect_array`*`)` creates an area in a Simulink model. The first part of the `path/text` argument is the path of the model where you want the area. Append to the path a slash (/), followed by the text for the area. The `rect_array` argument is a 1x4 array, specified as `[left, top, right, bottom]`, that gives the coordinates of the upper-left and lower-right corners of the area in pixels. These coordinates are relative to the upper left corner of the model or chart. Positive x and y values are to the right of and down from the origin, respectively.

`add_block('built-in/Image', '`*`path`*`/`*`text`*`', 'Position', `*`position_array`*`, 'imagePath', `*`path_to_image`*`)` creates an image in a Simulink model. The first part of the `path/text` argument is the path of the model where you want to place the image. Append to the path a slash (/), followed by the text for the image. The `position_array` argument is a 1x2 array, specified as `[left, top]`, that gives the coordinates of the upper-left corner in pixels. These coordinates are relative to the upper-left corner of the model or chart. Positive x and y values are to the right of and down from the origin, respectively. Specify the path to the image you want to insert in the `path_to_image` argument.

Calling `add_block` triggers the `CopyFcn` 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 6-86). 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.

Tips

Do not use `delete_block` to delete an annotation. For details, see "Delete an Annotation Programmatically".

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')
```

Create an annotation that says `This simulates a nonlinear second order system`. Position the annotation above the copyright line.

```
open_system('vdp');
```

```
block = add_block('built-in/Note', ...
'vdp/This simulates a nonlinear second order system', ...
'Position', [200 250])
```

In the `vdp` model, create an area labeled `Sample Area`. Position the annotation above the copyright line.

```
open_system('vdp');
block = add_block('built-in/Area', ...
'vdp/Sample Area', ...
'Position', [200 250 300 270]);
```

In the `vdp` model, insert an image of an airplane. Position the image below the copyright line.

```
open_system('vdp');
block = add_block('built-in/Image', ...
'vdp/My Image', ...
'Position', [0 300], ...
'imagePath', fullfile(matlabroot, 'toolbox', 'simulink', 'simulink', 'b747.jpg'));
```

More About

- "Annotation Callback Functions"

See Also

`delete_block` | `replace_block` | `set_param` | `Simulink.Annotation`

Introduced before R2006a

add_exec_event_listener

Register listener for block method execution event

Syntax

```
h = add_exec_event_listener(blk, event, listener);
```

Description

`h = add_exec_event_listener(blk, event, listener)` registers a listener for a block method execution event where the listener is a MATLAB program that performs some task, such as logging runtime data for a block, when the event occurs (see "Listen for Method Execution Events"). Simulink software invokes the registered listener whenever the specified event occurs during simulation of the model.

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.

Introduced before R2006a

add_line

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`

Introduced before R2006a

add_param

Add parameter to Simulink system

Syntax

```
add_param('sys','parameter1',value1,'parameter2',value2,...)
```

Description

The `add_param` command adds the specified parameters to the specified system and initializes the parameters to the specified values. Case is ignored for parameter names. Value 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`

Introduced before R2006a

addFile

Add file to Simulink Project

Syntax

```
addFile(proj,file)
```

Description

`addFile(proj,file)` adds a file to the project `proj`.

Examples

Add Files to a Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = simulinkproject;
```

Remove a file.

```
removeFile(proj,'models/AnalogControl.mdl')
```

Add the file back to the project.

```
addFile(proj,'models/AnalogControl.mdl');
```

Create and save a new model.

```
new_system('mymodel');
save_system('mymodel');
```

Add the new file to the project and return a project file object.

```
newPrjFile = addFile(proj,'mymodel.slx');
```

Use the project file object to manipulate the file, for example, adding a label.

```
addLabel(newPrjFile, 'Classification', 'Design')
```

Input Arguments

`proj` — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

`file` — Path of file

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'

See Also

Functions

`addFolderIncludingChildFiles` | `removeFile` | `simulinkproject`

Introduced in R2013a

addFolderIncludingChildFiles

Add folder and child files to Simulink Project

Syntax

```
addFolderIncludingChildFiles(proj,folder)
```

Description

`addFolderIncludingChildFiles(proj,folder)` adds a folder and all child files to the project `proj`.

Examples

Add Folders to a Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a new folder in the project folder.

```
new_folder_path = fullfile(proj.RootFolder, 'new_folder')
mkdir(new_folder_path);
```

Create a new folder in the previous folder.

```
new_sub_folder_path = fullfile(new_folder_path, 'new_sub_folder')
mkdir(new_sub_folder_path);
```

Create a new file in the folder.

```
filepath = fullfile(new_sub_folder_path, 'new_model_in_subfolder.slx')
new_system('new_model_in_subfolder');
save_system('new_model_in_subfolder', filepath)
```

Add this new folder and child files to the project.

```
projectFile = addFolderIncludingChildFiles(proj, new_folder_path)
```

Input Arguments

`proj` — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

`folder` — Path of folder

string

Path of the folder to add relative to the project root folder, specified as a string. The folder must be within the root folder.

Example: 'models'

See Also

Functions

`addFile` | `removeFile` | `simulinkproject`

Introduced in R2015b

addterms

Add terminators to unconnected ports in model

Syntax

```
addterms('sys')
```

Description

`addterms('sys')` adds Terminator and Ground blocks to the unconnected ports in the Simulink block diagram `sys`.

See Also

`slupdate`

Introduced before R2006a

attachConfigSet

Associate configuration set or configuration reference with model

Syntax

```
attachConfigSet(model, configObj)
attachConfigSet(model, configObj, allowRename)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObj`

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

`allowRename`

Boolean that determines how Simulink software handles a name conflict

Description

`attachConfigSet` associates the configuration set or configuration reference (configuration object) specified by `configObj` with `model`.

You cannot attach a configuration object to a model if the configuration object is already attached to another model, or has the same name as another configuration object attached to the same model. The optional Boolean argument `allowRename` determines how Simulink software handles a name conflict between configuration objects. If `allowRename` is `false` and the configuration object specified by `configObj` has the same name as a configuration object already attached to `model`, Simulink software generates an error. If `allowRename` is `true` and a name conflict occurs, Simulink software provides a unique name for `configObj` before associating `configObj` with `model`.

Examples

The following example creates a copy of the current model's active configuration object and attaches it to the model, changing its name if necessary to be unique. The code is the same whether the object is a configuration set or configuration reference.

```
myConfigObj = getActiveConfigSet(gcs);
copiedConfig = myConfigObj.copy;
copiedConfig.Name = 'DevConfig';
attachConfigSet(gcs, copiedConfig, true);
```

More About

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

See Also

`attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Introduced before R2006a

attachConfigSetCopy

Copy configuration set or configuration reference and associate it with model

Syntax

```
myConfigObj = attachConfigSetCopy(model, configObj)

myConfigObj = attachConfigSetCopy(model, configObj, allowRename)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObj`

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

`allowRename`

Boolean that specifies how Simulink software handles a name conflict

Description

`attachConfigSetCopy` copies the configuration set or configuration reference (configuration object) specified by `configObj` and associates the copy with `model`. Simulink software returns the copied configuration object as `newConfigObj`.

You cannot attach a configuration object to a model if the configuration object has the same name as another configuration object attached to the same model. The optional Boolean argument `allowRename` determines how Simulink software handles a name conflict between configuration objects. If `allowRename` is `false` and the configuration object specified by `configObj` has the same name as a configuration object already attached to `model`, Simulink software generates an error. If `allowRename` is `true` and a name conflict occurs, Simulink software provides a unique name for the copy of `configObj` before associating it with `model`.

Examples

The following example creates a copy of `ModelA`'s active configuration object and attaches it to `ModelB`, changing the name if necessary to be unique. The code is the same whether the object is a configuration set or configuration reference.

```
myConfigObj = getActiveConfigSet('ModelA');
newConfigObj = attachConfigSetCopy('ModelB', myConfigObj, true);
```

More About

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

See Also

`attachConfigSet` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Introduced in R2006b

addLabel

Attach label to Simulink Project file

Syntax

```
addLabel(file,categoryName,labelName)
addLabel(file,categoryName,labelName,labelData)
```

Description

`addLabel(file,categoryName,labelName)` attaches the specified label `labelName` in the category `categoryName` to the file.

`addLabel(file,categoryName,labelName,labelData)` attaches the label with data `labelData`.

Examples

Attach a Label to a Project File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a particular file by name.

```
myfile = findFile(proj,'models/AnalogControl.mdl')
```

```
myfile =

  ProjectFile with properties:

      Path: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
  Revision: '2'
```

```
SourceControlStatus: Unmodified
```

Get the `Labels` property of the file.

```
myfile.Labels
```

```
ans =

  Label with properties:

           File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
           Data: []
       DataType: 'none'
           Name: 'Design'
   CategoryName: 'Classification'
```

Attach the label `'Artifact'` to the file.

```
addLabel(myfile,'Classification','Artifact')
```

```
ans =

  Label with properties:

           File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
           Data: []
       DataType: 'none'
           Name: 'Artifact'
   CategoryName: 'Classification'
```

Index into the Labels property to get the label attached to this file.

```
reviewlabel = myfile.Labels(1)
```

```
reviewlabel =

  Label with properties:

           File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
           Data: []
       DataType: 'none'
           Name: 'Artifact'
   CategoryName: 'Classification'
```

Detach the new label from the file.

```
removeLabel(myfile,reviewlabel)
```

Attach a Label to a Subset of Files

Attach the `'Classification'` category label `'Utility'` to all files in the project that have the `.m` file extension.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get the list of files.

```
files = proj.Files;
```

Loop through each file. If a file has the extension `.m`, attach the label `'Utility'`.

```
for fileIndex = 1:numel(files)
   file = files(fileIndex);
   [~, ~, fileExtension] = fileparts(file.Path);
   if strcmp(fileExtension,'.m')
       addLabel(file,'Classification','Utility');
   end
end
```

In the Simulink Project **Files** view, the **Classification** column displays the label `Utility` for each `.m` file in the `utilities` folder.

Attach a Label and Label Data to a File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Create a new category `'Review'`.

```
createCategory(proj,'Review','char');
```

For the new category, create a label `'To Review'`.

```
reviewCategory = findCategory(proj,'Review');
createLabel(reviewCategory,'To Review');
```

Get a particular file by name.

```
myfile = findFile(proj,'models/AnalogControl.mdl')
```

```
myfile =

  ProjectFile with properties:

      Path: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
  Revision: '2'
SourceControlStatus: Unmodified
```

Attach the label `'To Review'` and a 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:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
            Data: 'Whole team design review'
        DataType: 'char'
            Name: 'To Review'
    CategoryName: 'Review'
```

In the Simulink Project **Files** view, for the `AnalogControl.mdl` file, the **Review** column displays the `To Review` label with label data.

Alternatively, you can set or change label data using the data property.

```
mylabel = myfile.Labels(2);
mylabel.Data = 'Final review';
```

Input Arguments

`file` — File to attach label to

file object

File to attach the label to, specified as a file object. You can get the file object by examining the project's Files property (`proj.Files`), or use `findFile` to find a file by name. The file must be in the project.

`categoryName` — Name of category for label

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`.

See Also

Functions

`createLabel` | `findFile` | `findLabel` | `removeLabel` | `simulinkproject`

Introduced in R2013a

bdclose

Close any or all Simulink system windows unconditionally

Syntax

```
bdclose
bdclose('sys')

bdclose('all')
```

Description

`bdclose` with no arguments closes the current system window unconditionally and without confirmation. Any changes made to the system since it was last saved are lost.

`bdclose('sys')` closes the specified system window.

`bdclose('all')` closes all system windows.

Examples

This command closes the `vdp` system.

```
bdclose('vdp')
```

See Also

`close_system` | `new_system` | `open_system` | `save_system`

Introduced before R2006a

bdIsLibrary

Whether block diagram is a library

Syntax

```
isLibrary = bdIsLibrary(bdnames)
```

Description

`isLibrary = bdIsLibrary(bdnames)` returns whether the loaded block diagrams specified by `bdnames` are libraries.

Examples

Check Whether Block Diagrams Are Libraries

Load some block diagrams and get a handle to one of them.

```
load_system({'sf_car','hydlib','vdp'})
h = get_param('hydlib','Handle');
```

Check whether `sf_car` is a library. The returned value 0 indicates that it is not.

```
bdIsLibrary('sf_car')
```

```
ans =
 0
```

Check whether `hydlib` and `vdp` are libraries. The returned value shows that `hydlib` is a library and `vdp` is not.

```
bdIsLibrary({'hydlib','vdp'})
```

```
ans =
1 0
```

Using the handle to `hydlib`, check whether `hdlib` is a library. The value returned shows that it is.

```
bdIsLibrary(h)
```

```
ans =
1
```

Input Arguments

`bdnames` — Names or handles of loaded block diagrams

string | cell array of strings | double | array of doubles

Names or handles of loaded block diagrams, specified as a string, a cell array of strings, a double, or a double array. All strings are names of loaded block diagrams. All doubles are handles of loaded block diagrams.

Data Types: `char` | `cell` | `double`

Output Arguments

`isLibrary` — Logical array showing whether block diagrams are libraries

logical scalar | logical array

Logical array showing whether block diagrams are libraries, returned as a logical scalar or array (`1` for a library, `0` otherwise).

See Also

`bdIsLoaded` | `bdroot` | `find_system`

Introduced in R2015a

bdIsLoaded

Whether block diagram is in memory

Syntax

```
isLoaded = bdIsLoaded(bdnames)
```

Description

`isLoaded = bdIsLoaded(bdnames)` returns whether or not a block diagram is in memory. `bdnames` can be a string or a cell array of strings. All strings must be valid block diagram names. It is an error to supply a path to a block or subsystem.

`isLoaded` is a logical array with one entry for each block diagram name.

Examples

```
bdIsLoaded('sf_car')
```
returns a logical scalar.

```
bdIsLoaded({'sf_car','vdp'})
```
returns a 1-by-2 logical array.

See Also

`find_system` | `bdIsLibrary`

Introduced in R2008a

bdroot

Return name of top-level Simulink system

Syntax

```
bdroot
bdroot(obj)
bdroot(handle)
bdroot(sys)
```

Description

`bdroot` with no arguments returns the name of the current top-level system.

`bdroot(obj)`, where `obj` is a string specifying a system or block path name, returns the name of the top-level system containing the specified object name. The `bdroot` of an empty string generates an error. Prior to issuing `bdroot`, make sure that the top-level system is loaded.

`bdroot(handle)`, where `handle` is the numeric handle for a system or block, returns the numeric handle of the top-level system containing the specified object. Prior to issuing `bdroot`, make sure that the top-level system for each element in the cell array is loaded. If you specify a vector of handles, Simulink returns a list of handles of the top-level systems.

`bdroot(sys)`, where `sys` is a cell array of system names or a vector of system handles. Prior to issuing `bdroot`, make sure that the top-level system for each element in the cell array is loaded. If you specify a vector of handles, Simulink returns a list of handles of the top-level systems.

Examples

This command returns the name of the top-level system that contains the current block.

```
bdroot(gcb)
```

This command returns the name of the top-level system that contains the current system.

```
bdroot (gcs)
```

This command returns the numeric handle of the top-level system that contains the current block.

```
bdroot (gcbh)
```

If `gcbh` is a cell array of system names or handles, it returns a cell array containing the corresponding top-level system names or handles.

If `gcbh` is a numeric array of system handles, it returns a numeric array containing the corresponding top-level system handles.

See Also

`find_system` | `gcb` | `gcs` | `gcbh` | `load_system`

Introduced before R2006a

dlinmod

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`

Introduced in R2007a

close_system

Close Simulink system window or block dialog box

Syntax

```
close_system

close_system('sys')

close_system('sys', saveflag)

close_system('sys', 'newname')

close_system('sys', 'newname','ErrorIfShadowed', true)
```

Description

`close_system` with no arguments closes the current system or subsystem window. If the current system is the top-level system and it has been modified, `close_system` returns an error. The current system is defined in the description of the `gcs` command.

`close_system('sys')` closes the specified system, subsystem, or block window.

`close_system('sys')` unloads a model after specifying

- `load_system('sys')`.

`'sys'` can be a 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` | `load_system`

Introduced before R2006a

closeDialog

Close configuration parameters dialog

Syntax

```
closeDialog(configObj)
```

Arguments

configObj

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`closeDialog` closes an open configuration parameters dialog box. If *`configObj`* is a configuration set, the function closes the dialog box that displays the configuration set. If *`configObj`* is a configuration reference, the function closes the dialog box that displays the referenced configuration set, or generates an error if the reference does not specify a valid configuration set. If the dialog box is already closed, the function does nothing.

Examples

The following example closes a configuration parameters dialog box that shows the current parameters for the current model. The parameter values derive from the active configuration set or configuration reference (configuration object). The code is the same in either case; the only difference is which type of configuration object is currently active.

```
myConfigObj = getActiveConfigSet(gcs);
closeDialog(myConfigObj);
```

More About

- "Manage a Configuration Set"

- “Manage a Configuration Reference”

See Also

`attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Introduced in R2006b

close

Close Simulink Project

Syntax

```
close(proj)
```

Description

`close(proj)` closes the project `proj`.

Examples

Open and Close a Simulink Project

Open a specified project and get a project object to manipulate the project at the command line. For example,

```
proj = simulinkproject('C:/projects/project1/myproject.prj')
```

Close the project.

```
close(proj)
```

Input Arguments

`proj` — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

See Also

Functions

`simulinkproject`

Introduced in R2013a

coder.allowpcode

Package: coder

Control code generation from protected MATLAB files

Syntax

```
coder.allowpcode('plain')
```

Description

`coder.allowpcode('plain')` allows you to generate protected MATLAB code (P-code) that you can then compile into optimized MEX functions or embeddable C/C++ code. This function does not obfuscate the generated MEX functions or embeddable C/C++ code.

With this capability, you can distribute algorithms as protected P-files that provide code generation optimizations, providing intellectual property protection for your source MATLAB code.

Call this function in the top-level function before control-flow statements, such as `if`, `while`, `switch`, and function calls.

MATLAB functions can call P-code. When the `.m` and `.p` versions of a file exist in the same folder, the P-file takes precedence.

`coder.allowpcode` is ignored outside of code generation.

Examples

Generate optimized embeddable code from protected MATLAB code:

1. Write an function `p_abs` that returns the absolute value of its input:

```
function out = p_abs(in)   %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
```

```
coder.allowpcode('plain');
out = abs(in);
```

2 Generate protected P-code. At the MATLAB prompt, enter:

```
pcode p_abs
```

The P-file, `p_abs.p`, appears in the current folder.

3 Generate a MEX function for `p_abs.p`, using the `-args` option to specify the size, class, and complexity of the input parameter (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -args { int32(0) }
```

`codegen` generates a MEX function in the current folder.

4 Generate embeddable C code for `p_abs.p` (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -config:lib -args { int32(0) };
```

`codegen` generates C library code in the `codegen\lib\p_abs` folder.

See Also

`pcode` | `codegen`

Introduced in R2011a

coder.ceval

Package: coder

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.

When the LCC compiler creates a library, it adds a leading underscore to the library function names. If the compiler for the library was LCC and your code generation compiler is not LCC, you must add the leading underscore to the function name in a `coder.ceval` call. For example, `coder.ceval('_mylibfun')`. If the compiler for a library was not LCC, you cannot use LCC to generate code from MATLAB code that calls functions from that library. Those library function names do not have the leading underscore that the LCC compiler requires.

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 `double` and returns a value of type `double`.

```
#ifdef MATLAB_MEX_FILE
#include <tmwtypes.h>
#else
#include "rtwtypes.h"
#endif

double foo(double in1, double in2);
```

2. Write the C function `foo.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

double foo(double in1, double in2)
{
  return in1 + in2;
}
```

3. Write a function `callfoo` that calls `foo` using `coder.ceval`.

```
function y = callfoo  %#codegen
y = 0.0;
if coder.target('MATLAB')
    % Executing in MATLAB, call MATLAB equivalent of
    % C function foo
    y = 10 + 20;
else
    % Executing in generated code, call C function foo
    y = coder.ceval('foo', 10, 20);
end
end
```

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.

```
double callfoo(void)
{
  /*  Executing in generated code, call C function foo */
  return foo(10.0, 20.0);
```

```
}
```

In this case, you have not specified the type of the input arguments, that is, the type of the constants `10` and `20`. Therefore, the arguments are implicitly of double-precision, floating-point type by default, because the default type for constants in MATLAB is `double`.

Call 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 coder.target('MATLAB')
     % Executing in MATLAB, call function absval
     y = absval(y);
   else
     % Executing in the generated code.
     % Call the initialize function before calling the
     % C function for the first time
     coder.ceval('absval_initialize');

     % Call the generated C library function absval
     y = coder.ceval('absval',y);

     % Call the terminate function after
     % calling the C function for the last time
     coder.ceval('absval_terminate');
   end
   ```

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_mex
```

More About

- "Compilation Directive %#codegen"
- "External Code Integration"
- "Data Definition Basics"

See Also

| | | | | `coder.extrinsic` | `coder.opaque` | `coder.ref` | `coder.rref` | `coder.wref` | `coder.target` | |

Introduced in R2011a

coder.cinclude

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>
```

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 `myMult2.c` that doubles its input. Save it in a subfolder `mycfiles`.

```
#include "myMult2.h"
double myMult2(double u)
{
    return 2 * u;
```

```
}
```

Write the application header file `myMult2.h`. Save it in the subfolder `mycfiles`.

```
#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif
```

Write a MATLAB function that conditionally includes the application header file and calls the external C function.

```
function y = myfunc
%#codegen
    y = 21;
    if ~coder.target('MATLAB')
    % Running in generated code
        coder.cinclude('myMult2.h');
        y = coder.ceval('myMult2', y);
    else
    % Running in MATLAB
        y = y * 2;
    end
end
```

Compile the MATLAB function. Use the `-I` option to specify the path to the external header and C files.

```
codegen -config:lib myfunc -I mycfiles
```

Here is the generated C code:

```
/* Include files */
#include "rt_nonfinite.h"
#include "myfunc.h"
#include "myMult2.h"

/* Function Definitions */
double myfunc(void)
{
  /*  Running in generated code */
  return myMult2(21.0);
}

/* End of code generation (myfunc.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, `myMult2.h`)
- `rt_nonfinite.h` (if you do not specify `SupportNonFinite=false` using `coder.config` when you compile the entry-point function).

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`

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 ~coder.target('MATLAB')
   coder.cinclude('foo.h');
   coder.ceval('foo');
end
...
```

More About

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`

Introduced in R2013a

coder.const

Fold expressions into constants in generated code

Syntax

```
out = coder.const(expression)
[out1,...,outN] = coder.const(handle,arg1,...,argN)
```

Description

`out = coder.const(expression)` evaluates `expression` and replaces `out` with the result of the evaluation in generated code.

`[out1,...,outN] = coder.const(handle,arg1,...,argN)` evaluates the multi-output function having handle `handle`. It then replaces `out1,...,outN` with the results of the evaluation in the generated code.

Examples

Specify Constants in Generated Code

This example shows how to specify constants in generated code using `coder.const`.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generation software generates code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int k;
  for (k = 0; k < 10; k++) {
    y[k] = (double)((1 + k) * (1 + k)) + Shift;
  }
}
```

Replace the statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((1:10).^2)+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generation software creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int i0;
  static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
                                       49, 64, 81, 100 };

  for (i0 = 0; i0 < 10; i0++) {
    y[i0] = (double)iv0[i0] + Shift;
  }
}
```

Create Lookup Table in Generated Code

This example shows how to fold a user-written function into a constant in generated code.

Write a function `getsine` that takes an input `index` and returns the element referred to by `index` from a lookup table of sines. The function `getsine` creates the lookup table using another function `gettable`.

```
function y = getsine(index) %#codegen
  assert(isa(index, 'int32'));
  persistent tbl;
  if isempty(tbl)
           tbl = gettable(1024);
  end
  y = tbl(index);

function y = gettable(n)
      y = zeros(1,n);
      for i = 1:n
         y(i) = sin((i-1)/(2*pi*n));
      end
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

```
codegen -config:lib -launchreport getsine -args int32(0)
```

The generated code contains instructions for creating the lookup table.

Replace the statement:

```
tbl = gettable(1024);
```

with:

```
tbl = coder.const(gettable(1024));
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

The generated code contains the lookup table itself. `coder.const` forces the expression `gettable(1024)` to be evaluated during code generation. The generated code does not contain instructions for the evaluation. The generated code contains the result of the evaluation itself.

Specify Constants in Generated Code Using Multi-Output Function

This example shows how to specify constants in generated code using a multi-output function in a `coder.const` statement.

Write a function `MultiplyConst` that takes an input `factor` and multiplies every element of two vectors `vec1` and `vec2` with `factor`. The function generates `vec1` and `vec2` using another function `EvalConsts`.

```
function [y1,y2] = MultiplyConst(factor) %#codegen
  [vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
  y1=vec1.*factor;
  y2=vec2.*factor;

function [f1,f2]=EvalConsts(z,n)
  f1=z.^(2*n)/factorial(2*n);
  f2=z.^(2*n+1)/factorial(2*n+1);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generation software generates code for creating the vectors.

Replace the statement

```
[vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
```

with

```
[vec1,vec2]=coder.const(@EvalConsts,pi.*(1./2.^(1:10)),2);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generation software does not generate code for creating the vectors. Instead, it calculates the vectors and specifies the calculated vectors in generated code.

Read Constants by Processing XML File

This example shows how to call an extrinsic function using `coder.const`.

Write an XML file `MyParams.xml` containing the following statements:

```
<params>
    <param name="hello" value="17"/>
    <param name="world" value="42"/>
```

```
</params>
```

Save `MyParams.xml` in the current folder.

Write a MATLAB function `xml2struct` that reads an XML file. The function identifies the XML tag `param` inside another tag `params`.

After identifying `param`, the function assigns the value of its attribute `name` to the field name of a structure `s`. The function also assigns the value of attribute `value` to the value of the field.

```
function s = xml2struct(file)

s = struct();
doc = xmlread(file);
els = doc.getElementsByTagName('params');
for i = 0:els.getLength-1
    it = els.item(i);
    ps = it.getElementsByTagName('param');
    for j = 0:ps.getLength-1
        param = ps.item(j);
        paramName = char(param.getAttribute('name'));
        paramValue = char(param.getAttribute('value'));
        paramValue = evalin('base', paramValue);
        s.(paramName) = paramValue;
    end
end
```

Save `xml2struct` in the current folder.

Write a MATLAB function `MyFunc` that reads the XML file `MyParams.xml` into a structure `s` using the function `xml2struct`. Declare `xml2struct` as extrinsic using `coder.extrinsic` and call it in a `coder.const` statement.

```
function y = MyFunc(u) %#codegen
  assert(isa(u, 'double'));
  coder.extrinsic('xml2struct');
  s = coder.const(xml2struct('MyParams.xml'));
  y = s.hello + s.world + u;
```

Generate code for `MyFunc` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:dll -launchreport MyFunc -args 0
```

The code generation software executes the call to `xml2struct` during code generation. It replaces the structure fields `s.hello` and `s.world` with the values 17 and 42 in generated code.

Input Arguments

`expression` — MATLAB expression or user-written function

expression with constants | single-output function with constant arguments

MATLAB expression or user-defined single-output function.

The expression must have compile-time constants only. The function must take constant arguments only. For instance, the following code leads to a code generation error, because `x` is not a compile-time constant.

```
function y=func(x)
   y=coder.const(log10(x));
```

To fix the error, assign `x` to a constant in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args coder.Constant(10)
```

Example: `2*pi`, `factorial(10)`

`handle` — Function handle

function handle

Handle to built-in or user-written function.

Example: `@log`, `@sin`

Data Types: `function_handle`

`arg1,...,argN` — Arguments to the function with handle `handle`

function arguments that are constants

Arguments to the function with handle `handle`.

The arguments must be compile-time constants. For instance, the following code leads to a code generation error, because `x` and `y` are not compile-time constants.

```
function y=func(x,y)
```

```
y=coder.const(@nchoosek,x,y);
```

To fix the error, assign `x` and `y` to constants in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args {coder.Constant(10),coder.Constant(2)}
```

Output Arguments

`out` — Value of `expression`

value of the evaluated expression

Value of `expression`. In the generated code, MATLAB Coder replaces occurrences of `out` with the value of `expression`.

`out1,...,outN` — Outputs of the function with handle `handle`

values of the outputs of the function with handle `handle`

Outputs of the function with handle `handle`.MATLAB Coder evaluates the function and replaces occurrences of `out1,...,outN` with constants in the generated code.

More About

Tips

- The code generation software constant-folds expressions automatically when possible. Typically, automatic constant-folding occurs for expressions with scalars only. Use `coder.const` when the code generation software does not constant-fold expressions on its own.

Introduced in R2013b

coder.cstructname

Package: coder

Name structure in generated code

Syntax

```
coder.cstructname(var,'structName')
coder.cstructname(var,'structName','extern')
coder.cstructname(var,'structName','extern',Name,Value)
```

Description

`coder.cstructname(var,'structName')` specifies the name of the structure type that represents `var` in the generated C/C++ code. `var` is a structure or cell array variable. `structName` is the name for the structure type in the generated code. Call `coder.cstructname` before the first use of the variable. If `var` is a cell array element, call `coder.cstructname` after the first assignment to the element.

`coder.cstructname(var,'structName','extern')` declares an externally defined structure. It does not generate the definition of the structure type. Provide the definition in a custom include file.

`coder.cstructname(var,'structName','extern',Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Limitations

- You cannot use `coder.cstructname` with global variables.
- If `var` is a cell array or the field names of externally defined structures must be `f1`, `f2`, and so on.
- If `var` is a cell array element, call `coder.cstructname` after the first assignment to the element. For example:

```
...
x = cell(2,2);
x{1} = struct('a', 3);
coder.cstructname(x{1}, 'mytype');
...
```

Tips

- The code generation software represents a heterogeneous cell array as a structure in the generated C/C++ code. To specify the name of the generated structure type, use `coder.cstructname`. Using `coder.cstructname` with a cell array variable makes the cell array heterogeneous.
- To use `coder.cstructname` on arrays, use single indexing. For example, you cannot use `coder.cstructname(x(1,2))`. Instead, use single indexing, for example `coder.cstructname(x(n))`.
- If you use `coder.cstructname` on an array, it sets the name of the base type of the array, not the name of the array. Therefore, you cannot use `coder.cstructname` on the base element and then on the array. For example, the following code does not work. The second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previous `coder.cstructname`, `myStructName`.

  ```
  % Define scalar structure with field a
  myStruct = struct('a', 0);
  coder.cstructname(myStruct,'myStructName');
  % Define array of structure with field a
  myStructArray = repmat(myStruct,k,n);
  coder.cstructname(myStructArray,'myStructArrayName');
  ```
- If you are using custom structure types, specify the name of the header file that includes the external definition of the structure. Use the `HeaderFile` input argument.
- If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. To take advantage of target-specific function implementations that require data to be aligned, use the `Alignment` input argument.
- You can also use `coder.cstructname` to assign a name to a substructure, which is a structure that appears as a field of another structure. For more information, see “Assign a Name to a SubStructure” on page 2-66.

Input Arguments

`structName`

The name of the structure type in the generated code.

`var`

Structure or cell array variable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Alignment'`

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary. Hence it is not matched by CRL functions that require alignment.

`Alignment` must be either `-1` or a power of `2` that is not greater than `128`.

Default: -1

`'HeaderFile'`

Name of the header file that contains the external definition of the structure, for example, `'mystruct.h'`.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` option, MATLAB Coder includes that header file exactly at the point where it is required.

Must be a non-empty 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`. In this folder, the structure definition is in `topfun_types.h`.

```
typedef struct
{
    double a;
    double b;
} MyStruct;
```

Assign a Name to a Structure and Pass It to a Function

Assign the name `MyStruct` to the structure `var`. Pass the structure to a C function `use_struct`.

1. Create a C header file, `use_struct.h`, for a `use_struct`function that takes a parameter of type `MyStruct`. Define a structure of type `MyStruct` in the header file.

```
#ifdef MATLAB_MEX_FILE
#include <tmwtypes.h>
#else
#include "rtwtypes.h"
#endif

typedef struct MyStruct
{
    double s1;
    double s2;
} MyStruct;

void use_struct(struct MyStruct *my_struct);
```

2 Write the C function `use_struct.c`.

```
#include <stdio.h>
#include <stdlib.h>

#include "use_struct.h"

void use_struct(struct MyStruct *my_struct)
{
  double x = my_struct->s1;
  double y = my_struct->s2;
}
```

3 Write a `m_use_struct` compliant with MATLAB that declares a structure. Have the function assign the name `MyStruct` to the structure. Then, have the function call the C function `use_struct` using `coder.ceval`.

```
function m_use_struct   %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Declare a MATLAB structure
var.s1 = 1;
var.s2 = 2;

% Assign the name MyStruct to the structure variable.
% extern indicates this is an externally defined
% structure.
coder.cstructname(var, 'MyStruct', 'extern');

% Call the C function use_struct. The type of var
% matches the signature of use_struct.
```

```
% Use coder.rref to pass the the variable var by
% reference as a read-only input to the external C
% function use_struct
coder.ceval('use_struct', coder.rref(var));
```

4 Generate C library code for function `m_use_struct`, passing `use_struct.h` to include the structure definition.

```
codegen -config:lib m_use_struct use_struct.c use_struct.h
```

`codegen` generates C code in the default folder `codegen\lib\m_use_struct`. The generated header file `m_use_struct_types.h` in this folder does not contain a definition of the structure `MyStruct` because `MyStruct` is an external type.

Assign a Name to a SubStructure

Use `coder.cstructname` to assign a name to a substructure.

1 Define a MATLAB structure, `top`, that has another structure, `lower`, as a field.

```
% Define structure top with field lower,
% which is a structure with fields a and b
top.lower = struct('a',1,'b',1);
top.c = 1;
```

2 Define a function, `MyFunc`, which takes an argument, `TopVar`, as input. Mark the function for code generation using `%#codegen`.

```
function out = MyFunc(TopVar) %#codegen
```

3 Inside `MyFunc`, include the following lines

```
coder.cstructname(TopVar,'topType');
coder.cstructname(TopVar.lower,'lowerType');
```

4 So that `TopVar` has the same type as `top`, generate C code for `MyFunc` with an argument having the same type as `top`.

```
codegen -config:lib MyFunc -args coder.typeof(top)
```

In the generated C code, the field variable `TopVar.lower` is assigned the type name `lowerType`. For instance, the structure declaration of the variable `TopVar.lower` appears in the C code as:

```
typedef struct
{
    /* Definitions of a and b appear here */
} lowerType;
```

and the structure declaration of the variable `TopVar` appears as:

```
typedef struct
{
     lowerType lower;
    /* Definition of c appears here */
} topType;
```

Assign a Name to a Structure That Is an Element of a Cell Array

Write a function `struct_in_cell` that has a cell array `x{1}` that contains a structure. The `coder.cstructname` call follows the assignment to `x{1}`.

```
function z = struct_in_cell()
x = cell(2,2);
x{1} = struct('a', 3);
coder.cstructname(x{1}, 'mytype');
z = x{1};
end
```

Generate a static library for `struct_in_cell`.

```
codegen -config:lib struct_in_cell -report
```

The type for `a` has the name `mytype`.

```
 typedef struct {
  double a;
} mytype;
```

More About

- "Homogeneous vs. Heterogeneous Cell Arrays"

Introduced in R2011a

coder.extrinsic

Package: coder

Declare extrinsic function or functions

Syntax

```
coder.extrinsic('function_name');

coder.extrinsic('function_name_1', ... , 'function_name_n');

coder.extrinsic('-sync:on', 'function_name');

coder.extrinsic('-sync:on', 'function_name_1', ... ,
'function_name_n');

coder.extrinsic('-sync:off','function_name');

coder.extrinsic('-sync:off', 'function_name_1', ... ,
'function_name_n');
```

Arguments

function_name
function_name_1, ... , *function_name_n*

Declares *function_name* or *function_name_1* through *function_name_n* as extrinsic functions.

–*sync:on*

function_name or *function_name_1* through *function_name_n*.

Enables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *function_name* or *function_name_1* through *function_name_n*. If only a few extrinsic calls modify global data, turn off synchronization before and after all extrinsic function calls by setting the global synchronization mode to `At MEX-function entry and exit`. Use the *–sync:on* option to turn on synchronization for only the extrinsic calls that *do* modify global data.

–sync:off

Disables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *`function_name`* or *`function_name_1`* through *`function_name_n`*. If most extrinsic calls modify global data, but a few do not, you can use the *–sync:off* option to turn off synchronization for the extrinsic calls that *do not* modify global data.

Description

`coder.extrinsic` declares extrinsic functions. During simulation, the code 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 must 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 MATLAB issues a compiler error.

Examples

The following code declares the MATLAB function `patch` as extrinsic in the MATLAB local function `create_plot`.

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
%  and displays the triangle as a patch object.

c = sqrt(a^2 + b^2);

create_plot(a, b, color);

function create_plot(a, b, color)

%Declare patch as extrinsic
coder.extrinsic('patch');

x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');
```

By declaring `patch` as extrinsic, you instruct the code generation software not to compile or generate code for `patch`. Instead, the code generation software dispatches `patch` to MATLAB for execution.

More About

- "Call MATLAB Functions"
- "Controlling Synchronization for Extrinsic Function Calls"
- "Resolution of Function Calls for Code Generation"
- "Restrictions on Extrinsic Functions for Code Generation"

See Also

`coder.ceval` | `coder.screener`

Introduced in R2011a

coder.inline

Package: coder

Control inlining in generated code

Syntax

```
coder.inline('always')
coder.inline('never')
coder.inline('default')
```

Description

`coder.inline('always')` forces inlining 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. The code generation software does not inline entry-point functions.

`coder.inline('always')` does not inline functions called from `parfor`-loops. The code generation software does not inline functions into `parfor`-loops.

Examples

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;
```

More About

inlining

Technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/C++ code. Inlining can create opportunities for further optimization of the generated C/C++ code.

Introduced in R2011a

coder.load

Load compile-time constants from MAT-file or ASCII file into caller workspace

Syntax

```
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat',var1,...,varN)
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

Description

`S = coder.load(filename)` loads compile-time constants from `filename`.

- If `filename` is a MAT-file, then `coder.load` loads variables from the MAT-file into a structure array.
- If `filename` is an ASCII file, then `coder.load` loads data into a double-precision array.

`S = coder.load(filename,var1,...,varN)` loads only the specified variables from the MAT-file `filename`.

`S = coder.load(filename,'-regexp',expr1,...,exprN)` loads only the variables that match the specified regular expressions.

`S = coder.load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`S = coder.load(filename,'-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`S = coder.load(filename,'-mat',var1,...,varN)` treats `filename` as a MAT-file and loads only the specified variables from the file.

`S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)` treats `filename` as a MAT-file and loads only the variables that match the specified regular expressions.

Examples

Load compile-time constants from MAT-file

Generate code for a function `edgeDetect1` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect1` uses `coder.load` to load the edge detection kernel from a MAT-file at compile time.

Save the Sobel edge-detection kernel in a MAT-file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];

save sobel.mat k
```

Write the function `edgeDetect1`.

```
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

S = coder.load('sobel.mat','k');
H = conv2(double(originalImage),S.k, 'same');
V = conv2(double(originalImage),S.k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect1`.

```
codegen -report -config cfg edgeDetect1
```

`codegen` generates C code in the `codegen\lib\edgeDetect1` folder.

Load compile-time constants from ASCII file

Generate code for a function `edgeDetect2` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect2` uses `coder.load` to load the edge detection kernel from an ASCII file at compile time.

Save the Sobel edge-detection kernel in an ASCII file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];
save sobel.dat k -ascii
```

Write the function `edgeDetect2`.

```
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = coder.load('sobel.dat');
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect2`.

```
codegen -report -config cfg edgeDetect2
```

`codegen` generates C code in the `codegen\lib\edgeDetect2` folder.

Input Arguments

`filename` — Name of file

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.

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.

More About

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.
- "Regular Expressions"

See Also

`matfile` | `regexp` | `save`

Introduced in R2013a

coder.nullcopy

Package: coder

Declare uninitialized variables

Syntax

```
X = coder.nullcopy(A)
```

Description

`X = coder.nullcopy(A)` copies type, size, and complexity of `A` to `X`, but does not copy element values. Preallocates memory for `X` without incurring the overhead of initializing memory.

`coder.nullcopy` does not support MATLAB classes as inputs.

Use With Caution

Use this function with caution. See "How to Eliminate Redundant Copies by Defining Uninitialized Variables".

Examples

The following example shows how to declare variable `X` as a 1-by-5 vector of real doubles without performing an unnecessary initialization:

```
function X = foo

N = 5;
X = coder.nullcopy(zeros(1,N));
for i = 1:N
   if mod(i,2) == 0
      X(i) = i;
   else
```

```
      X(i) = 0;
   end
end
```

Using `coder.nullcopy` with `zeros` lets you specify the size of vector *X* without initializing each element to zero.

More About

- "Eliminate Redundant Copies of Variables in Generated Code"

Introduced in R2011a

coder.opaque

Declare variable in generated code

Syntax

```
y = coder.opaque(type)
y = coder.opaque(type,value)
y = coder.opaque(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.

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
```

Cast to and from Types of Variables Declared Using `coder.opaque`

This example shows how to cast to and from types of variables that are declared using `coder.opaque`. The function `castopaque` calls the C run-time function `strncmp` to compare at most `n` characters of the strings `s1` and `s2`. `n` is the number of characters in

the shorter of the strings. To generate the correct C type for the `strncmp` input `nsizet`, the function casts `n` to the C type `size_t` and assigns the result to `nsizet`. The function uses `coder.opaque` to declare `nsizet`. Before using the output `retval` from `strncmp`, the function casts `retval` to the MATLAB type `int32` and stores the results in `y`.

Write this MATLAB function:

```
function y = castopaque(s1,s2)

%  <0 - the first character that does not match has a lower value in s1 than in s2
%   0 - the contents of both strings are equal
%  >0 - the first character that does not match has a greater value in s1 than in s2
%
%#codegen

coder.cinclude('<string.h>');
n = min(numel(s1), numel(s2));

% Convert the number of characters to compare to a size_t

nsizet = cast(n,'like',coder.opaque('size_t','0'));

% The return value is an int
retval = coder.opaque('int');
retval = coder.ceval('strncmp', cstr(s1), cstr(s2), nsizet);

% Convert the opaque return value to a MATLAB value
y = cast(retval, 'int32');

%--------------
function sc = cstr(s)
% NULL terminate a MATLAB string for C
sc = [s, char(0)];
```

Generate the MEX function.

```
codegen castopaque -args {blanks(3), blanks(3)} -report
```

Call the MEX function with inputs `'abc'` and `'abc'`.

```
castopaque_mex('abc','abc')

ans =
```

```
     0
```

The output is `0` because the strings are equal.

Call the MEX function with inputs `'abc'` and `'abd'`.

```
castopaque_mex('abc','abd')
```

```
ans =

    -1
```

The output is `-1` because the third character `d` in the second string is greater than the third character `c` in the first string.

Call the MEX function with inputs `'abd'` and `'abc'`.

```
castopaque_mex('abd','abc')
```

```
ans =

     1
```

The output is `1` because the third character `d` in the first string is greater than the third character `c` in the second string.

In the MATLAB workspace, you can see that the type of `y` is `int32`.

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`.

Specify a `value` that has the type that `type` specifies. Otherwise, the generated code can produce unexpected results.

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`

More About

Tips

- Specify a `value` that has the type that `type` specifies. Otherwise, the generated code can produce unexpected results. For example, the following `coder.opaque` declaration can produce unexpected results.

  ```
  y = coder.opaque('int', '0.2')
  ```

- `coder.opaque` declares the type of a variable. It does not instantiate the variable. You can instantiate a variable by using it later in the MATLAB code. In the following example, assignment of `fp1` from `coder.ceval` instantiates `fp1`.

  ```
  % Declare fp1 of type FILE *
  fp1 = coder.opaque('FILE *');
  %Create the variable fp1
  fp1 = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
  ```

- In the MATLAB environment, `coder.opaque` returns the value specified in `value`. If `value` is not provided, it returns 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 2-84
- To avoid multiple inclusions of the same header file in generated code, enclose the header file in the conditional preprocessor statements `#ifndef` and `#endif`. For example:

  ```
  #ifndef MyHeader_h
  #define MyHeader_h
  <body of header file>
  #endif
  ```

- You can use the MATLAB `cast` function to cast a variable to or from a variable that is declared using `coder.opaque`. Use `cast` with `coder.opaque` only for numeric types.

  To cast a variable declared by `coder.opaque` to a MATLAB type, you can use the `B = cast(A,type)` syntax. For example:

  ```
  x = coder.opaque('size_t','0');
  ```

```
x1 = cast(x, 'int32');
```

You can also use the `B = cast(A,'like',p)` syntax. For example:

```
x = coder.opaque('size_t','0');
x1 = cast(x, 'like', int32(0));
```

To cast a MATLAB variable to the type of a variable declared by `coder.opaque`, you must use the `B = cast(A,'like',p)` syntax. For example:

```
x = int32(12);
x1 = coder.opaque('size_t', '0');
x2 = cast(x, 'like', x1));
```

Use `cast` with `coder.opaque` to generate the correct data types for:

- Inputs to C/C++ functions that you call using `coder.ceval`.
- Variables that you assign to outputs from C/C++ functions that you call using `coder.ceval`.

Without this casting, it is possible to receive compiler warnings during code generation.

See Also

`coder.ceval` | `coder.ref` | `coder.rref` | `coder.wref`

Introduced in R2011a

coder.ref

Package: coder

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 C function prototype for `my_fcn` must be as follows:

```
double my_fcn(double *u)
```

In this example, the generated code infers the type of the input `u` from the `codegen` argument.

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`.

See Also

`coder.ceval` | `coder.rref` | `coder.wref`

Introduced in R2011a

coder.rref

Package: coder

Pass argument by reference as read-only input

Syntax

```
[y =] coder.ceval('function_name', coder.rref(argI), ... un)
```

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), ... un)` 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 C function prototype for `foo` must be as follows:

```
double foo(const double *u)
```

In this example, the generated code infers the type of the input `u` from the `codegen` argument.

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`.

See Also

`coder.ceval` | `coder.opaque` | `coder.ref` | `coder.wref` | | |

Introduced in R2011a

coder.screener

Determine if function is suitable for code generation

Syntax

```
coder.screener(fcn)
coder.screener(fcn_1,...,fcn_n )
```

Description

`coder.screener(fcn)` analyzes the entry-point MATLAB function, `fcn`. It identifies unsupported functions and language features, such as recursion and nested functions, as code generation compliance issues. It displays the code generation compliance issues in a report. If `fcn` calls other functions directly or indirectly that are not MathWorks® functions, `coder.screener` analyzes these functions. It does not analyze MathWorks functions. It is possible that `coder.screener` does not detect all code generation issues. Under certain circumstances, it is is possible that `coder.screener` reports false errors.

`coder.screener(fcn_1,...,fcn_n )` analyzes entry-point functions (`fcn_1,...,fcn_n`).

Input Arguments

`fcn`

Name of entry-point MATLAB function that you want to analyze.

`fcn_1,...,fcn_n`

Comma-separated list of names of entry-point MATLAB functions that you want to analyze.

Examples

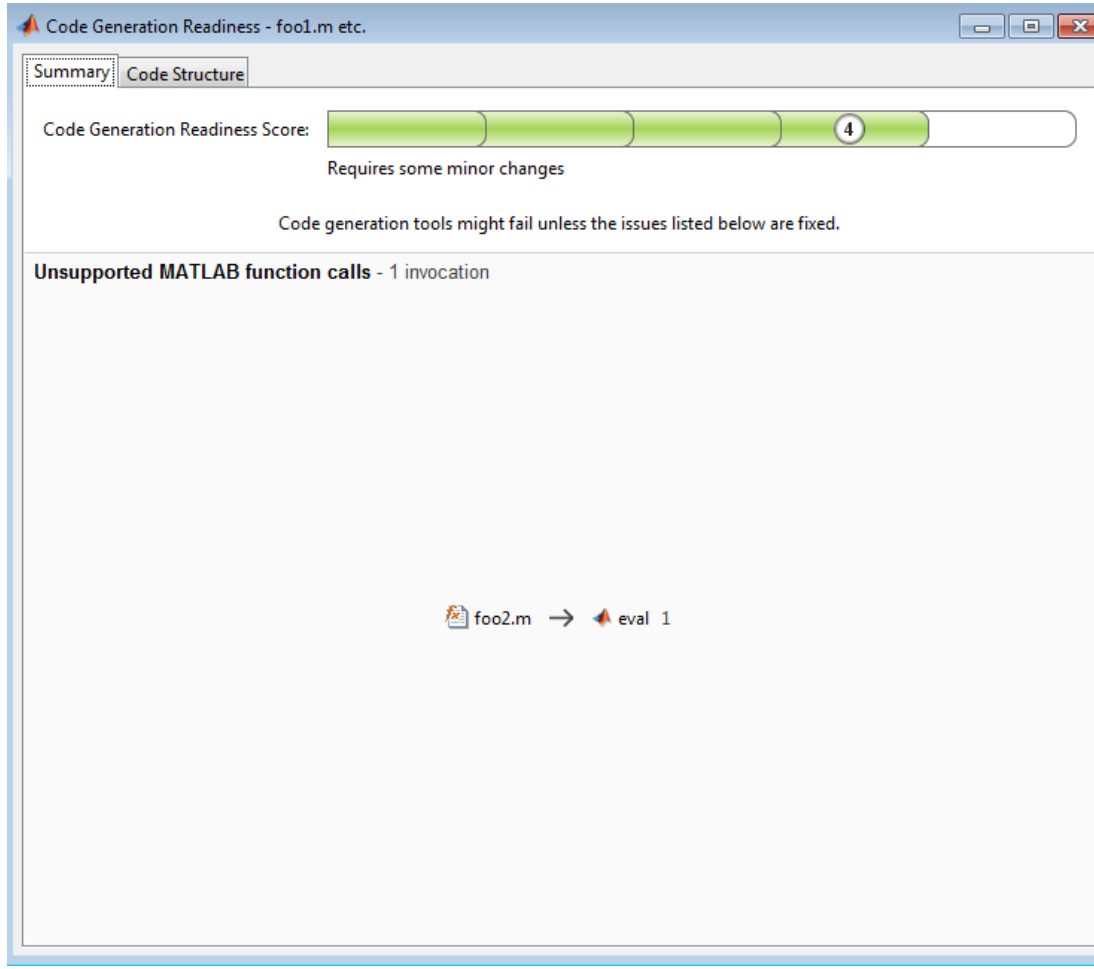
Identify Unsupported Functions

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`. Put `foo1` and `foo2` in separate files.

```
function out = foo1(in)
  out = foo2(in);
  disp(out);
end

function out = foo2(in)
  out = eval(in);
end

coder.screener('foo1')
```

The code generation readiness report displays a summary of the unsupported MATLAB function calls. The function `foo2` calls one unsupported MATLAB function.

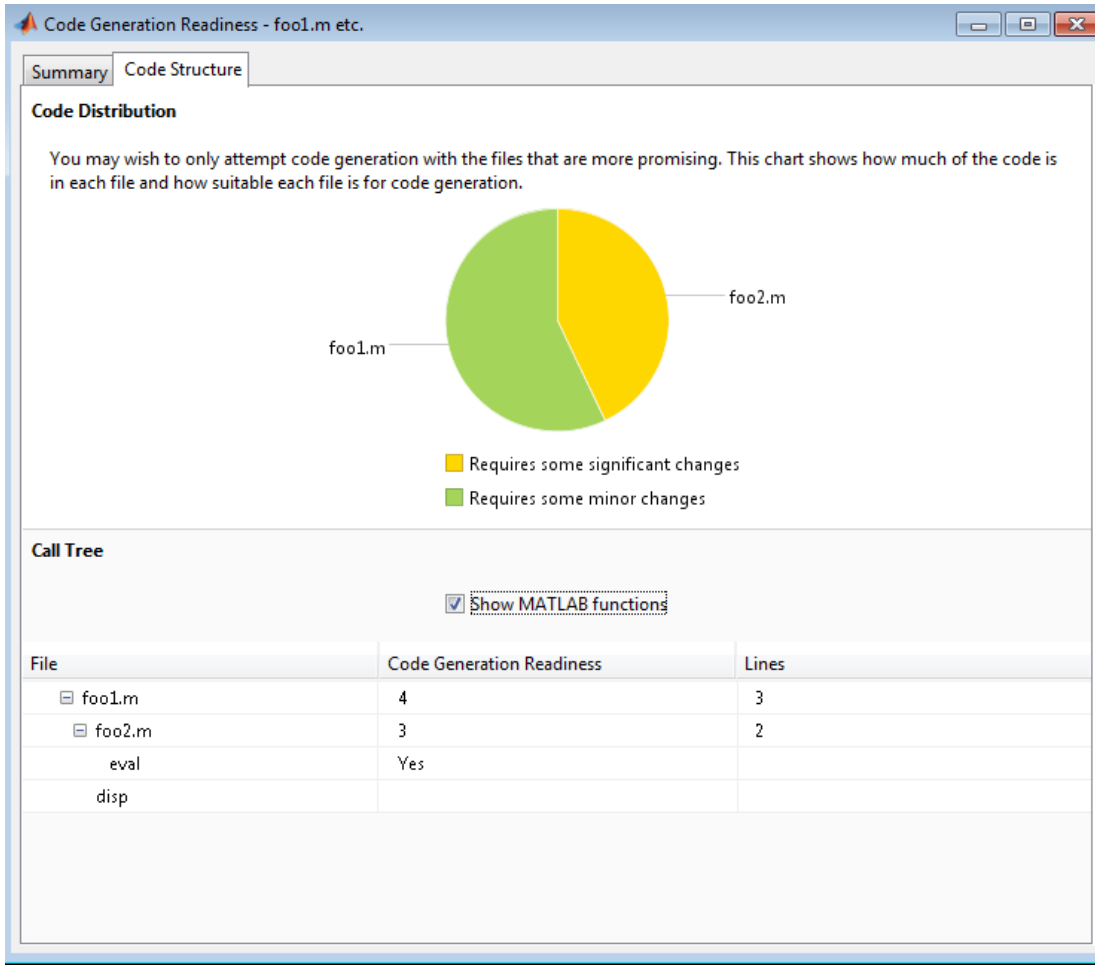


In the report, click the **Code Structure** tab and select the **Show MATLAB functions** check box.

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:

- Colors `foo1.m` green to indicate that it is suitable for code generation.
- Colors `foo2.m` yellow to indicate that it requires significant changes.

- Assigns `foo1.m` a code generation readiness score of 4 and `foo2.m` a score of 3. The score is based on a scale of 1–5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool does not detect issues.
- Displays a call tree.



The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function which code generation does not support. To generate a MEX function for `foo2.m`, modify the code to make the call to `eval` extrinsic.

```
function out = foo2(in)
  coder.extrinsic('eval');
  out = eval(in);
end
```

Rerun the code generation readiness tool.

```
coder.screener('foo1')
```

The report no longer flags that code generation does not support the `eval` function. When you generate a MEX function for `foo1`, the code generation software dispatches `eval` to MATLAB for execution. For standalone code generation, it does not generate code for it.

Identify Unsupported Data Types

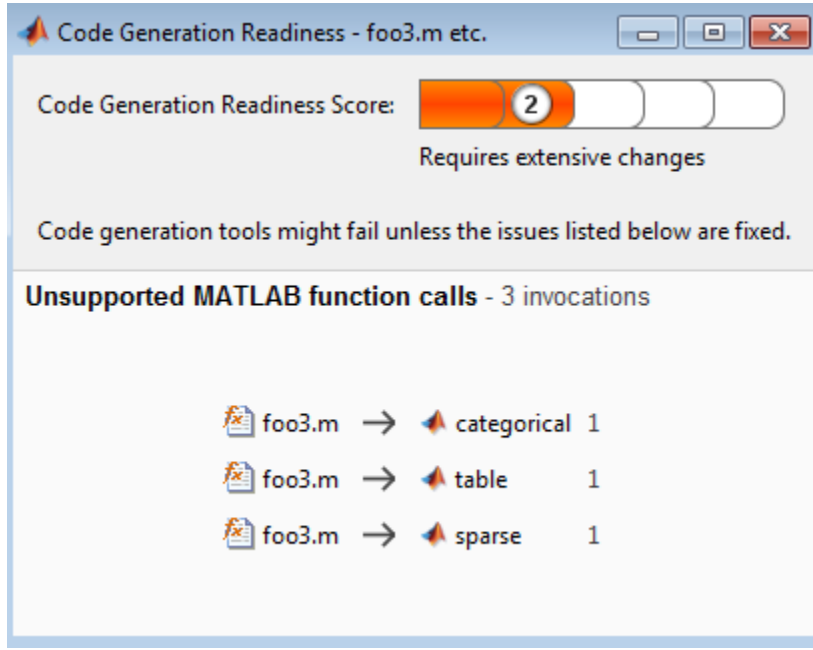
The `coder.screener` function identifies data types that code generation does not support.

Analyze the MATLAB function `foo3` that uses unsupported data types.

```
function [outSparse,outCategorical] = foo3(A,B,C)
    outSparse = sparse(A);
    outCategorical = categorical(B);
    outTable = table(C);
end
```

```
coder.screener('foo3')
```

The code generation readiness report displays a summary of the unsupported data types.



The report assigns the code a code readiness score of 2. This score indicates that the code requires extensive changes.

Before generating code, you must fix the reported issues.

Determine Code Generation Readiness for Multiple Entry-Point Functions

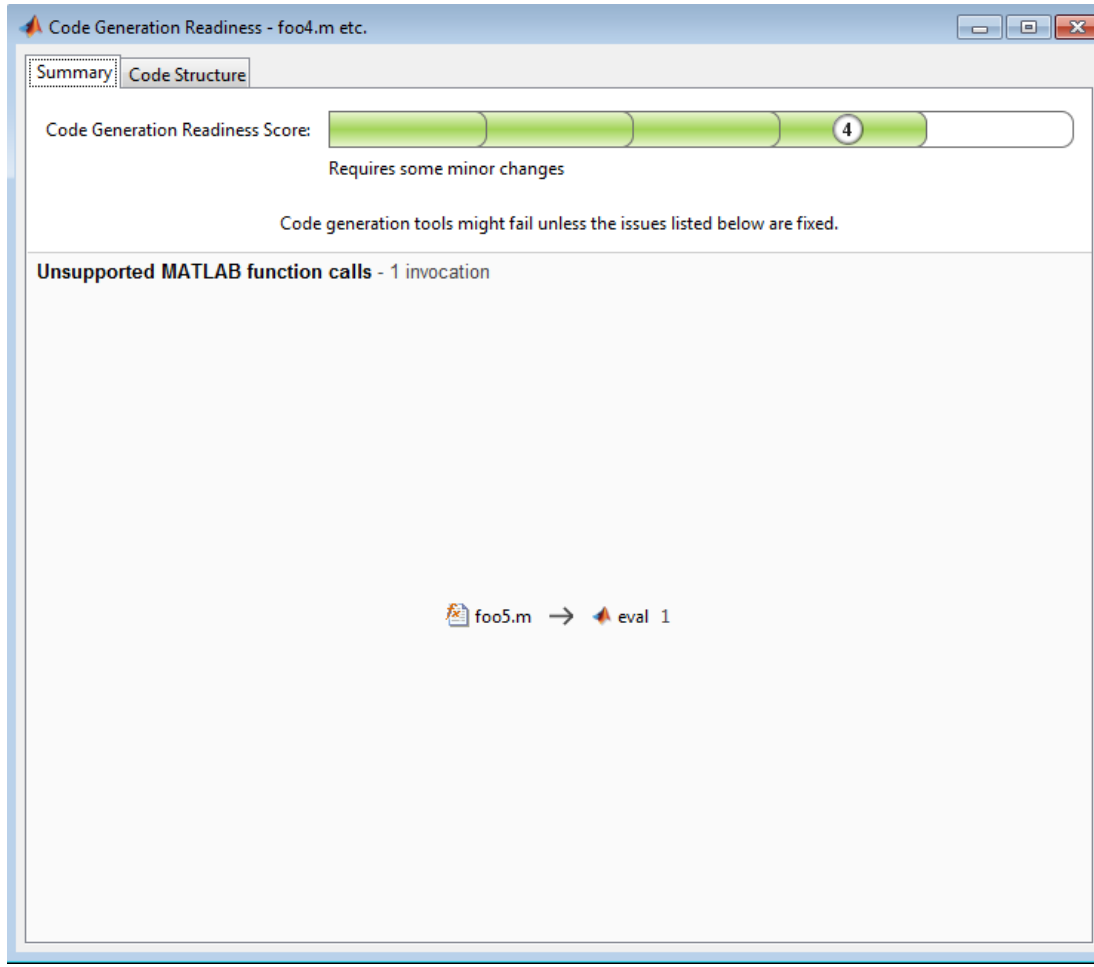
The `coder.screener` function identifies calls to functions that code generation does not support. It checks the entry-point functions `foo4` and `foo5`.

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')
```

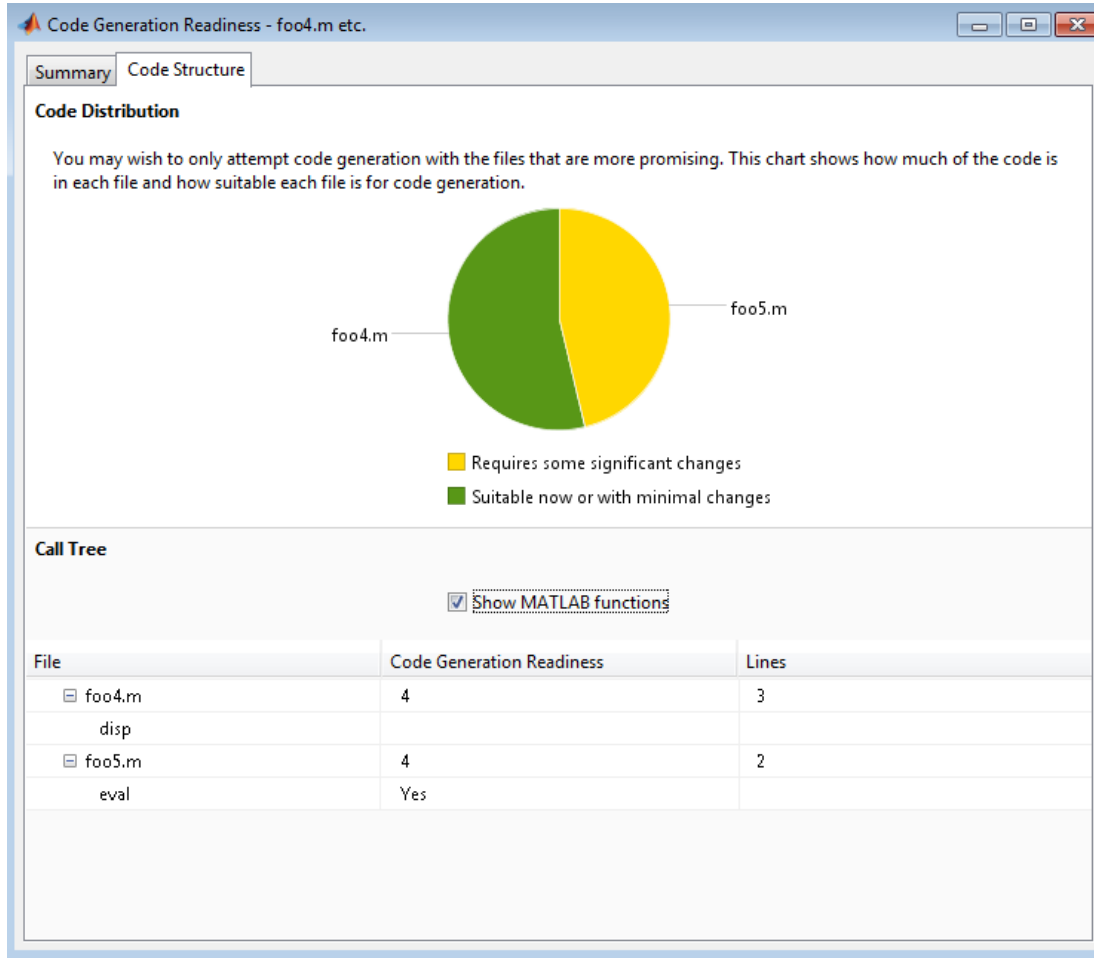
The code generation readiness report displays a summary of the unsupported MATLAB function calls. The function `foo5` calls one unsupported MATLAB function.



In the report, click the **Code Structure** tab. Select the **Show MATLAB functions** check box.

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:

- Colors `foo4.m` green to indicate that it is suitable for code generation.
- Colors `foo5.m` yellow to indicate that it requires significant changes.
- Assigns `foo4.m` a code generation readiness score of 4 and `foo5.m` a score of 4. The score is based on a scale of 1–5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool cannot detect issues.
- Displays a call tree.



Alternatives

- “Run the Code Generation Readiness Tool From the Current Folder Browser”

More About

Tips

- Before using `coder.screener`, fix issues that the Code Analyzer identifies.
- Before generating code, use `coder.screener` to check that a function is suitable for code generation. Fix all the issues that it detects.

- "Functions and Objects Supported for C and C++ Code Generation — Alphabetical List"
- "Functions and Objects Supported for C and C++ Code Generation — Category List"
- "Code Generation Readiness Tool"

Introduced in R2012b

coder.target

Determine if code generation target is specified target

Syntax

```
tf = coder.target(target)
```

Description

`tf = coder.target(target)` returns true (1) if the code generation target is `target`. Otherwise, it returns false (0).

If you generate code for MATLAB classes, MATLAB computes class initial values at class loading time before code generation. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns true.

Examples

Use coder.target to parameterize a MATLAB function

Parameterize a MATLAB function so that it works in MATLAB or generated code. When the function runs in MATLAB, it calls the MATLAB function `myabsval`. The generated code, however, calls a C library function `myabsval`.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)   %#codegen
y = abs(u);
```

Generate the C library for `myabsval.m`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

`codegen` creates the library `myabsval.lib` and header file `myabsval.h` in the folder `/codegen/lib/myabsval`. It also generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

```
function y = callmyabsval  %#codegen
y = -2.75;
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
  % Executing in MATLAB, call function myabsval
  y = myabsval(y);
else
  % Executing in the generated code.
  % Call the initialize function before calling the
  % C function for the first time
  coder.ceval('myabsval_initialize');

  % Call the generated C library function myabsval
  y = coder.ceval('myabsval',y);

  % Call the terminate function after
  % calling the C function for the last time
  coder.ceval('myabsval_terminate');
end
```

Convert `callmyabsval.m` to the MEX function `callmyabsval_mex`.

```
codegen -config:mex callmyabsval codegen/lib/myabsval/myabsval.lib...
     codegen/lib/myabsval/myabsval.h
```

Run the MATLAB function `callmyabsval` .

```
callmyabsval
```

```
ans =

    2.7500
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex
```

```
ans =
```

```
    2.7500
```

Input Arguments

`target` — code generation target

string

Code generation target specified as one of the following strings:

`'MATLAB'`	Running in MATLAB (not generating code)
`'MEX'`	Generating a MEX function
`'Sfun'`	Simulating a Simulink model
`'Rtw'`	Generating a LIB, DLL, or EXE target
`'HDL '`	Generating an HDL target
`'Custom'`	Generating a custom target

Example: `tf = coder.target('MATLAB')`

Data Types: `char`

See Also

`coder.ceval`

Introduced in R2011a

coder.unroll

Package: coder

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 getrand(double y[8])
{
  /*  Turn off inlining to make  */
  /*  generated code easier to read */
  /*  Set flag variable dounroll to repeat loop body */
  /*  only for fewer than 10 iterations */
  /*  Declare size, class, and complexity */
  /*  of variable y by assignment */
  memset(&y[0], 0, sizeof(double) << 3);

  /*  Loop body begins */
  y[2] = b_rand();
  y[4] = b_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 b_getrand(double y[50])
{
  int i;
  int b_i;

  /*  Turn off inlining to make  */
  /*  generated code easier to read */
```

```
/*  Set flag variable dounroll to repeat loop body */
/*  only for fewer than 10 iterations */
/*  Declare size, class, and complexity */
/*  of variable y by assignment */
memset(&y[0], 0, 50U * sizeof(double));

/*  Loop body begins */
for (i = 0; i < 25; i++) {
  b_i = (i << 1) + 1;
  if ((b_i > 2) && (b_i < 48)) {
    y[b_i - 1] = b_rand();
  }
}
```

More About

- "Using Logicals in Array Indexing"

See Also

`| | for | |`

Introduced in R2011a

coder.updateBuildInfo

Update build information object `RTW.BuildInfo`

Syntax

```
coder.updateBuildInfo('addCompileFlags',options)
coder.updateBuildInfo('addLinkFlags',options)
coder.updateBuildInfo('addDefines',options)
coder.updateBuildInfo(___,group)

coder.updateBuildInfo('addLinkObjects',filename,path)
coder.updateBuildInfo('addLinkObjects',filename,path, priority,
precompiled)
coder.updateBuildInfo('addLinkObjects',filename,path, priority,
precompiled,linkonly)
coder.updateBuildInfo(___,group)

coder.updateBuildInfo('addNonBuildFiles',filename)
coder.updateBuildInfo('addSourceFiles',filename)
coder.updateBuildInfo('addIncludeFiles',filename)
coder.updateBuildInfo(___,path)
coder.updateBuildInfo(___,path,group)

coder.updateBuildInfo('addSourcePaths',path)
coder.updateBuildInfo('addIncludePaths',path)
coder.updateBuildInfo(___,group)
```

Description

`coder.updateBuildInfo('addCompileFlags',options)` adds compiler options to the build information object.

`coder.updateBuildInfo('addLinkFlags',options)` adds link options to the build information object.

`coder.updateBuildInfo('addDefines',options)` adds preprocessor macro definitions to the build information object.

`coder.updateBuildInfo(___,group)` assigns a group name to `options` for later reference.

`coder.updateBuildInfo('addLinkObjects',filename,path)` adds a link object from a file to the build information object.

`coder.updateBuildInfo('addLinkObjects',filename,path, priority, precompiled)` specifies if the link object is precompiled.

`coder.updateBuildInfo('addLinkObjects',filename,path, priority, precompiled,linkonly)` specifies if the object is to be built before being linked or used for linking alone. If the object is to be built, it specifies if the object is precompiled.

`coder.updateBuildInfo(___,group)` assigns a group name to the link object for later reference.

`coder.updateBuildInfo('addNonBuildFiles',filename)` adds a nonbuild-related file to the build information object.

`coder.updateBuildInfo('addSourceFiles',filename)` adds a source file to the build information object.

`coder.updateBuildInfo('addIncludeFiles',filename)` adds an include file to the build information object.

`coder.updateBuildInfo(___,path)` adds the file from specified path.

`coder.updateBuildInfo(___,path,group)` assigns a group name to the file for later reference.

`coder.updateBuildInfo('addSourcePaths',path)` adds a source file path to the build information object.

`coder.updateBuildInfo('addIncludePaths',path)` adds an include file path to the build information object.

`coder.updateBuildInfo(___,group)` assigns a group name to the path for later reference.

Examples

Add Multiple Compiler Options

Add the compiler options `-Zi` and `-Wall` during code generation for function, `func`.

Anywhere in the MATLAB code for `func`, add the following line:

```
coder.updateBuildInfo('addCompileFlags','-Zi -Wall');
```

Generate code for `func` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport func
```

Add Source File Name

Add a source file to the project build information while generating code for a function, `calc_factorial`.

1. Write a header file `fact.h` that declares a C function `factorial`.

   ```
   double factorial(double x);
   ```

   `fact.h` will be included as a header file in generated code. This inclusion ensures that the function is declared before it is called.

   Save the file in the current folder.

2. Write a C file `fact.c` that contains the definition of `factorial`. `factorial` calculates the factorial of its input.

   ```
   #include "fact.h"

         double factorial(double x)
         {
             int i;
             double fact = 1.0;
             if (x == 0 || x == 1) {
                return 1.0;
            } else {
                for (i = 1; i <= x; i++) {
                    fact *= (double)i;
                }
                return fact;
   ```

```
        }
    }
```

`fact.c` is used as a source file during code generation.

Save the file in the current folder.

3 Write a MATLAB function `calc_factorial` that uses `coder.ceval` to call the external C function `factorial`.

Use `coder.updateBuildInfo` with option `'addSourceFiles'` to add the source file `fact.c` to the build information. Use `coder.cinclude` to include the header file `fact.h` in the generated code.

```
function y = calc_factorial(x) %#codegen

  coder.cinclude('fact.h');
  coder.updateBuildInfo('addSourceFiles', 'fact.c');

  y = 0;
  y = coder.ceval('factorial', x);
```

4 Generate code for `calc_factorial` using the `codegen` command.

```
codegen -config:dll -launchreport calc_factorial -args 0
```

Add Link Object

Add a link object `LinkObj.lib` to the build information while generating code for a function `func`. For this example, you must have a link object `LinkObj.lib` saved in a local folder, for example, `c:\Link_Objects`.

Anywhere in the MATLAB code for `func`, add the following lines:

```
libPriority = '';
libPreCompiled = true;
libLinkOnly = true;
libName = 'LinkObj.lib';
libPath = 'c:\Link_Objects';
coder.updateBuildInfo('addLinkObjects', libName, libPath, ...
     libPriority, libPreCompiled, libLinkOnly);
```

Generate a MEX function for `func` using the `codegen` command. Open the Code Generation Report.

```
codegen -launchreport func
```

Add Include Paths

Add an include path to the build information while generating code for a function, `adder`. Include a header file, `adder.h`, existing on the path.

When header files do not reside in the current folder, to include them, use this method:

1. Write a header file `mysum.h` that contains the declaration for a C function `mysum`.

   ```
   double mysum(double, double);
   ```

   Save it in a local folder, for example `c:\coder\myheaders`.
2. Write a C file `mysum.c` that contains the definition of the function `mysum`.

   ```
   #include "mysum.h"

   double mysum(double x, double y)
    {
     return(x+y);
    }
   ```

   Save it in the current folder.
3. Write a MATLAB function `adder` that adds the path `c:\coder\myheaders` to the build information.

   Use `coder.cinclude` to include the header file `mysum.h` in the generated code.

   ```
   function y = adder(x1, x2) %#codegen
      coder.updateBuildInfo('addIncludePaths','c:\coder\myheaders');
      coder.updateBuildInfo('addSourceFiles','mysum.c');
        %Include the source file containing C function definition
      coder.cinclude('mysum.h');
      y = 0;
      if coder.target('MATLAB')
         % This line ensures that the function works in MATLAB
           y = x1 + x2;
      else
          y = coder.ceval('mysum', x1, x2);
       end
   end
   ```
4. Generate code for `adder` using the `codegen` command.

```
codegen -config:lib -launchreport adder -args {0,0}
```

Input Arguments

`options` — Build options

string

Build options, specified as a string. The string must be a compile-time constant.

Depending on the leading argument, `options` specifies the relevant build options to be added to the project's build information.

Leading Argument	Values in `options`
`'addCompileFlags'`	Compiler options
`'addLinkFlags'`	Link options
`'addDefines'`	Preprocessor macro definitions

The function adds the options to the end of an option vector.

Example: `coder.updateBuildInfo('addCompileFlags','-Zi -Wall')`

`group` — Group name

string

Name of user-defined group, specified as a string. The string must be a compile-time constant.

The `group` option assigns a group name to the parameters in the second argument.

Leading Argument	Second Argument	Parameters Named by `group`
`'addCompileFlags'`	`options`	Compiler options
`'addLinkFlags'`	`options`	Link options
`'addLinkObjects'`	`filename`	Name of file containing linkable objects
`'addNonBuildFiles'`	`filename`	Name of nonbuild-related file
`'addSourceFiles'`	`filename`	Name of source file
`'addSourcePaths'`	`path`	Name of source file path

You can use `group` to:

- Document the use of specific parameters.
- Retrieve or apply multiple parameters together as one group.

`filename` — File name

string

File name, specified as a string. The string must be a compile-time constant.

Depending on the leading argument, `filename` specifies the relevant file to be added to the project's build information.

Leading Argument	File Specified by `filename`
`'addLinkObjects'`	File containing linkable objects
`'addNonBuildFiles'`	Nonbuild-related file
`'addSourceFiles'`	Source file

The function adds the file name to the end of a file name vector.

`path` — Full path name

string

Full path name, specified as a string. The string must be a compile-time constant.

Depending on the leading argument, `path` specifies the relevant path name to be added to the project's build information.

Leading Argument	Path Specified by `path`
`'addLinkObjects'`	Path to linkable objects
`'addNonBuildFiles'`	Path to nonbuild-related files
`'addSourceFiles'`, `'addSourcePaths'`	Path to source files

The function adds the path to the end of a path name vector.

`priority` — Relative priority of link object

`''`

Priority of link objects.

This feature applies only when several link objects are added. Currently, only a single link object file can be added for every `coder.updateBuildInfo` statement. Therefore, this feature is not available for use.

To use the succeeding arguments, include `''` as a placeholder argument.

`precompiled` — Variable indicating if link objects are precompiled

logical value

Variable indicating if the link objects are precompiled, specified as a logical value. The value must be a compile-time constant.

If the link object has been prebuilt for faster compiling and linking and exists in a specified location, specify `true`. Otherwise, the MATLAB Coder build process creates the link object in the build folder.

If `linkonly` is set to `true`, this argument is ignored.

Data Types: `logical`

`linkonly` — Variable indicating if objects must be used for linking only

logical value

Variable indicating if objects must be used for linking only, specified as a logical value. The value must be a compile-time constant.

If you want that the MATLAB Coder build process must not build or generate rules in the makefile for building the specified link object, specify `true`. Instead, when linking the final executable, the process should just include the object. Otherwise, rules for building the link object are added to the makefile.

You can use this argument to incorporate link objects for which source files are not available.

If `linkonly` is set to `true`, the value of `precompiled` is ignored.

Data Types: `logical`

Introduced in R2013b

coder.varsize

Package: coder

Declare variable-size array

Syntax

```
coder.varsize('var1', 'var2', ...)
coder.varsize('var1', 'var2', ..., ubound)
coder.varsize('var1', 'var2', ..., ubound, dims)
coder.varsize('var1', 'var2', ..., [], dims)
```

Description

`coder.varsize('var1', 'var2', ...)` declares one or more variables as variable-size data, allowing subsequent assignments to extend their size. Each `'varn'` must be a quoted string that represents a variable or structure field. If the structure field belongs to an array of structures, use colon (`:`) as the index expression to make the field variable-size for all elements of the array. For example, the expression `coder.varsize('data(:).A')` declares that the field `A` inside each element of `data` is variable sized.

`coder.varsize('var1', 'var2', ..., ubound)` declares one or more variables as variable-size data with an explicit upper bound specified in *`ubound`*. The argument *`ubound`* must be a constant, integer-valued vector of upper bound sizes for every dimension of each `'varn'`. If you specify more than one `'varn'`, each variable must have the same number of dimensions.

`coder.varsize('var1', 'var2', ..., ubound, dims)` declares one or more variables as variable size with an explicit upper bound and a mix of fixed and varying dimensions specified in *`dims`*. The argument *`dims`* is a logical vector, or double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` in *`dims`* have fixed size; dimensions that correspond to ones or `true` vary in size. If you specify more than one variable, each fixed dimension must have the same value across all `'varn'`.

`coder.varsize('var1', 'var2', ..., [], dims)` declares one or more variables as variable size with a mix of fixed and varying dimensions. The empty vector `[]` means that you do not specify an explicit upper bound.

When you do *not* specify *`ubound`*, the upper bound is computed for each `'varn'` in generated code.

When you do *not* specify *`dims`*, dimensions are assumed to be variable except the singleton ones. A singleton dimension is a dimension for which `size(A,dim)` = 1.

You must add the `coder.varsize` declaration before each `'varn'` is used (read). You can add the declaration before the first assignment to each `'varn'`. However, for a cell array element, the `coder.varsize` declaration must follow the first assignment to the element. For example:

```
...
x = cell(3, 3);
x{1} = [1 2];
coder.varsize('x{1}');
...
```

You cannot use `coder.varsize` outside the MATLAB code intended for code generation. For example, the following code does not declare the variable, `var`, as variable-size data:

```
coder.varsize('var',10);
codegen -config:lib MyFile -args var
```

Instead, include the `coder.varsize` statement inside `MyFile` to declare `var` as variable-size data.

Examples

Develop a Simple Stack That Varies in Size up to 32 Elements as You Push and Pop Data at Run Time.

Write primary function `test_stack` to issue commands for pushing data on and popping data from a stack.

```
function test_stack %#codegen
    % The directive %#codegen indicates that the function
```

```
    % is intended for code generation
    stack('init', 32);
    for i = 1 : 20
        stack('push', i);
    end
    for i = 1 : 10
        value = stack('pop');
        % Display popped value
        value
    end
end
```

Write local function `stack` to execute the push and pop commands.

```
function y = stack(command, varargin)
    persistent data;
    if isempty(data)
        data = ones(1,0);
    end
    y = 0;
    switch (command)
    case {'init'}
        coder.varsize('data', [1, varargin{1}], [0 1]);
        data = ones(1,0);
    case {'pop'}
        y = data(1);
        data = data(2:size(data, 2));
    case {'push'}
        data = [varargin{1}, data];
    otherwise
        assert(false, ['Wrong command: ', command]);
    end
end
```

The variable `data` is the stack. The statement `coder.varsize('data', [1, varargin{1}], [0 1])` declares that:

- `data` is a row vector
- Its first dimension has a fixed size
- Its second dimension can grow to an upper bound of 32

Generate a MEX function for `test_stack`:

```
codegen -config:mex test_stack
```

`codegen` generates a MEX function in the current folder.

Run `test_stack_mex` to get these results:

```
value =
    20

value =
    19

value =
    18

value =
    17

value =
    16

value =
    15

value =
    14

value =
    13

value =
    12

value =
    11
```

At run time, the number of items in the stack grows from zero to 20, and then shrinks to 10.

Declare a Variable-Size Structure Field.

Write a function `struct_example` that declares an array `data`, where each element is a structure that contains a variable-size field:

```
function y=struct_example() %#codegen
```

```
d = struct('values', zeros(1,0), 'color', 0);
data = repmat(d, [3 3]);
coder.varsize('data(:).values');

for i = 1:numel(data)
    data(i).color = rand-0.5;
    data(i).values = 1:i;
end

y = 0;
for i = 1:numel(data)
    if data(i).color > 0
        y = y + sum(data(i).values);
    end;
end
```

The statement `coder.varsize('data(:).values')` marks as variable-size the field `values` inside each element of the matrix `data`.

Generate a MEX function for `struct_example`:

```
codegen -config:mex struct_example
```

Run `struct_example`.

Each time you run `struct_example` you get a different answer because the function loads the array with random numbers.

Make a Cell Array Variable Size

Write the function `make_varsz_cell` that defines a local cell array variable `c` whose elements have the same class, but different sizes. Use `coder.varsize` to indicate that `c` has variable size.

```
function y =  make_varsz_cell()
c = {1 [2 3]};
coder.varsize('c', [1 3], [0 1]);
y = c;
end
```

Generate a C static library.

```
codegen -config:lib make_varsz_cell -report
```

In the report, view the MATLAB variables.

`c` is a 1x:3 homogeneous cell array whose elements are 1x:2 double.

- "Incompatibilities with MATLAB in Variable-Size Support for Code Generation"

Limitations

- If you use the cell function to create a `cell` array, you cannot use `coder.varsize` with that cell array.
- If you use `coder.varsize` with a cell array element, the `coder.varsize` declaration must follow the first assignment to the element. For example:

```
...
x = cell(3, 3);
x{1} = [1 2];
coder.varsize('x{1}');
...
```

- You cannot use `coder.varsize` with global variables.
- You cannot use `coder.varsize` with MATLAB class properties.

More About

Tips

- If you use input variables (or result of a computation using input variables) to specify the size of an array, it is declared as variable-size in the generated code. Do not use `coder.varsize` on the array again, unless you also want to specify an upper bound for its size.
- Using `coder.varsize` on an array without explicit upper bounds causes dynamic memory allocation of the array. This dynamic memory allocation can reduce the speed of generated code. To avoid dynamic memory allocation, use the syntax `coder.varsize('`*var_1*`', '`*var_2*`', ..., `*ubound*`)` to specify an upper bound for the array size (if you know it in advance).
- A cell array can be variable size only if it is homogeneous. When you use `coder.varsize` with a cell array, the code generation software tries to make the cell array homogeneous. It tries to find a class and size that apply to all elements of the cell array. For example, if the first element is double and the second element is 1x2 double, all elements can be represented as 1x:2 double. If the code generation software

cannot find a common class and size, code generation fails. For example, suppose that the first element of a cell array is char and the second element is double. The code generation software cannot find a class that can represent both elements.

- "Homogeneous vs. Heterogeneous Cell Arrays"

Introduced in R2011a

coder.wref

Package: coder

Pass argument by reference as write-only output

Syntax

```
[y =] coder.ceval('function_name', coder.wref(arg0), ... un);
```

Arguments

arg0

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(arg0), ... un);` passes the variable *`arg0`* 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);
coder.ceval('init', coder.wref(y));
```

The C function prototype for `init` must be as follows:

```
void init(double *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` explicitly—in this case to a matrix of type `double`.
- The generated code collapses matrices to a single dimension.

See Also

`coder.ceval` | `coder.ref` | `coder.rref`

Introduced in R2011a

getHardwareImplementation

Class: coder.BuildConfig
Package: coder

Get handle of copy of hardware implementation object

Syntax

```
hw = bldcfg.getHardwareImplementation()
```

Description

`hw = bldcfg.getHardwareImplementation()` returns the handle of a copy of the hardware implementation object.

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`hw`

Handle of copy of hardware implementation object.

getStdLibInfo

Class: coder.BuildConfig
Package: coder

Get standard library information

Syntax

```
[linkLibPath,linkLibExt,execLibExt,libPrefix]=
bldcfg.getStdLibInfo()
```

Description

`[linkLibPath,linkLibExt,execLibExt,libPrefix]= bldcfg.getStdLibInfo()` returns strings representing the:

- Standard MATLAB architecture-specific library path
- Platform-specific library file extension for use at link time
- Platform-specific library file extension for use at run time
- Standard architecture-specific library name prefix

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`linkLibPath`

Standard MATLAB architecture-specific library path specified as a string. The string can be empty.

`linkLibExt`

Platform-specific library file extension for use at link time, specified as a string. The value is one of `'.lib'`,`'.dylib'`,`'.so'`, `''`.

`execLibExt`

Platform-specific library file extension for use at run time, specified as a string. the value is one of `'.dll'`,`'.dylib'`,`'.so'`, `''`.

`linkPrefix`

Standard architecture-specific library name prefix, specified as a string. The string can be empty.

getTargetLang

Class: coder.BuildConfig
Package: coder

Get target code generation language

Syntax

```
lang = bldcfg.getTargetLang()
```

Description

`lang = bldcfg.getTargetLang()` returns a string containing the target code generation language.

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`lang`

A string containing the target code generation language. The value is 'C' or 'C++'.

getToolchainInfo

Class: coder.BuildConfig
Package: coder

Returns handle of copy of toolchain information object

Syntax

```
tc = bldcfg.getToolchainInfo()
```

Description

`tc = bldcfg.getToolchainInfo()` returns a handle of a copy of the toolchain information object.

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`tc`

Handle of copy of toolchain information object.

isCodeGenTarget

Class: coder.BuildConfig
Package: coder

Determine if build configuration represents specified target

Syntax

```
tf = bldcfg.isCodeGenTarget(target)
```

Description

`tf = bldcfg.isCodeGenTarget(target)` returns true (1) if the code generation target of the current build configuration represents the code generation target specified by `target`. Otherwise, it returns false (0).

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

`target`

Code generation target specified as a string or cell array of strings.

Specify	For code generation target
`'rtw'`	C/C++ dynamic Library, C/C++ static library, or C/C ++ executable
`'sfun'`	S-function (Simulation)
`'mex'`	MEX-function

Specify `target` as a cell array of strings to test if the code generation target of the build configuration represents one of the targets specified in the cell array.

For example:

```
...
mytarget = {'sfun','mex'};
tf = bldcfg.isCodeGenTarget(mytarget);
...
```

tests whether the build context represents an S-function target or a MEX-function target.

Output Arguments

`tf`

The value is true (1) if the code generation target of the build configuration represents the code generation target specified by `target`. Otherwise, the value is false (0).

See Also

`coder.target`

isMatlabHostTarget

Class: coder.BuildConfig
Package: coder

Determine if hardware implementation object target is MATLAB host computer

Syntax

```
tf = bldcfg.isMatlabHostTarget()
```

Description

`tf = bldcfg.isMatlabHostTarget()` returns true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, it returns false (0).

Input Arguments

`bldcfg`

`coder.BuildConfig` object.

Output Arguments

`tf`

Value is true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, the value is false (0).

coder.ExternalDependency.getDescriptiveName

Class: coder.ExternalDependency
Package: coder

Return descriptive name for external dependency

Syntax

```
extname = coder.ExternalDependency.getDescriptiveName(bldcfg)
```

Description

`extname = coder.ExternalDependency.getDescriptiveName(bldcfg)` returns the name that you want to associate with an "external dependency" on page 2-137. The code generation software uses the external dependency name for error messages.

Input Arguments

`bldcfg`

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-137

You can use this information when you want to return different names based on the build context.

Output Arguments

`extname`

External dependency name returned as a string.

Definitions

external dependency

External code interface represented by a class derived from a `coder.ExternalDependency` class. The external code can be a library, object files, or C/C++ source.

build context

Information used by the build process including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

Examples

Return external dependency name

Define a method that always returns the same name.

```
function myextname = getDescriptiveName(~)
   myextname = 'MyLibrary'
end
```

Return external library name based on the code generation target

Define a method that uses the build context to determine the name.

```
function myextname = getDescriptiveName(context)
    if context.isMatlabHostTarget()
        myextname = 'MyLibary_MatlabHost';
    else
        myextname = 'MyLibrary_Local';
    end
end
```

coder.ExternalDependency.isSupportedContext

Class: coder.ExternalDependency
Package: coder

Determine if build context supports external dependency

Syntax

```
tf = coder.ExternalDependency.isSupportedContext(bldcfg)
```

Description

`tf = coder.ExternalDependency.isSupportedContext(bldcfg)` returns true (1) if you can use the "external dependency" on page 2-139 in the current "build context" on page 2-139 . You must provide this method in the class definition for a class that derives from `coder.ExternalDependency`.

If you cannot use the "external dependency" on page 2-139 in the current "build context" on page 2-139, display an error message and stop code generation. The error message must describe why you cannot use the external dependency in this build context. If the method returns false (0), the code generation software uses a default error message. The default error message uses the name returned by the `getDescriptiveName` method of the `coder.ExternalDependency` class.

Use `coder.BuildConfig` methods to determine if you can use the external dependency in the current build context.

Input Arguments

`bldcfg`

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-139.

Output Arguments

`tf`

Value is true (1) if the build context supports the external dependency.

Definitions

external dependency

External code interface represented by a class derived from `coder.ExternalDependency` class. The external code can be a library, object file, or C/C++ source.

build context

Information used by the build process including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

Examples

Report error when build context does not support external library

This method returns true(1) if the code generation target is a MATLAB host target. Otherwise, the method reports an error and stops code generation.

Write `isSupportedContext` method.

```
function tf = isSupportedContext(ctx)
    if  ctx.isMatlabHostTarget()
        tf = true;
    else
```

```
            error('adder library not available for this target');
        end
    end
```

coder.ExternalDependency.updateBuildInfo

Class: coder.ExternalDependency
Package: coder

Update build information

Syntax

```
coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg)
```

Description

`coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg)` updates the build information object whose handle is `buildInfo`. After code generation, the build information object has standard information. Use this method to provide additional information required to link to external code. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-142.

You must provide this method in the class definition for a class that derives from `coder.ExternalDependency`.

Input Arguments

`buildInfo`

Handle of build information object.

`bldcfg`

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the "build context" on page 2-142.

Definitions

build context

Information used by the build process including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

convertToSlDataset

Convert contents of MAT-file to `Simulink.SimulationData.Dataset` object

Syntax

```
success=convertToSLDataset(source,destination)
success=convertToSLDataset(source,destination,datasetname)
```

Description

`success=convertToSLDataset(source,destination)` converts the contents of a MAT-file (`source`) to a destination MAT-file (`destination`).

`success=convertToSLDataset(source,destination,datasetname)` names the dataset `datasetname`.

When converting structure signal data, the function names the signal using the value contained in the label field of the structure signal field, such as: `mySignal.signal(1).label`.

This function ignores time expressions in source.

Examples

Save Signals to Dataset in file2.mat

Save signals from `file1.mat` to a dataset named `file1` in `file2.mat`.

```
success=convertToSLDataset('file1.mat','file2.mat')
```

Save Signals to Dataset Named myDataset in file2.mat

Save signals from `file1.mat` to a dataset named `myDataset` in `file2.mat`.

```
success=convertToSLDataset('file1.mat','file2.mat','myDataset')
```

Input Arguments

`source` — MAT-file

string

MAT–file that contains Simulink inputs.

`destination` — MAT-file

string

MAT–file to contain `Simulink.SimulationData.Dataset` converted from contents of `source`.

`datasetname` — Data set name

string

Data set name for new `Simulink.SimulationData.Dataset` object.

Output Arguments

`success` — Outcome of conversion

binary

Outcome of conversion, specified as binary:

- 1

  Conversion is successful.
- 0

  Conversion is not successful.

Introduced in R2016a

createCategory

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.

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 = 'Maintenance responsibility';
disp(label)
```

```
Label with properties:

            File: [1x80 char]
            Data: 'Maintenance responsibility'
        DataType: 'char'
            Name: 'Tom'
    CategoryName: 'Engineers'
```

Create a New Category of Labels with Datatype Double

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a new category of labels.

```
createCategory(proj,'Coverage','double')
```

```
category =

  Category with properties:

                Name: 'Coverage'
            DataType: 'double'
    LabelDefinitions: []
```

Find out what you can do with the new category.

```
category = findCategory(proj, 'Coverage');
methods(category)
```

```
Methods for class slproject.Category:

findLabel  removeLabel  createLabel
```

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.

More About

Tips

After you create a new category, you can create labels in the new category. See `createLabel`.

See Also

Functions
`createLabel` | `simulinkproject`

Introduced in R2013a

createLabel

Define Simulink Project label

Syntax

```
createLabel(category,newLabelName)
```

Description

`createLabel(category,newLabelName)` creates a new label, `newLabelName`, in a category. Use this syntax if you previously got a `category` by accessing a `Categories` property, e.g., using a command like `proj.Categories(1)`.

Use `addLabel` instead to create and attach a new label in an existing category using a single step.

Use `createCategory` first if you want to make a new category of labels.

Examples

Create a New Label

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = simulinkproject;
```

Examine the first existing category.

```
cat = proj.Categories(1)
```

```
cat =

  Category with properties:

              Name: 'Classification'
          DataType: 'none'
```

```
    LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat,'Future');
```

Create a New Category of Labels for File Ownership

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = simulinkproject;
```

Create creates a new category of labels called `Engineers` which can be used to denote file ownership in a project. These labels have the char `datatype` for attaching 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 = 'Maintenance responsibility';
disp(label)
```

```
Label with properties:

            File: [1x80 char]
            Data: 'Maintenance responsibility'
        DataType: 'char'
```

```
          Name: 'Tom'
CategoryName: 'Engineers'
```

Input Arguments

`category` — Category

category object

Category for the new label, specified as a category object. Get the category by accessing a `Categories` property, e.g. with a command like `proj.Categories(1)`, or use `findCategory`. To create a new category, use `createCategory`.

`newLabelName` — The name of the new label to define

string

The name of the new label to define, specified as a string.

See Also

Functions

`addLabel` | `createCategory`

Introduced in R2013a

delete_block

Delete blocks from Simulink system

Syntax

```
delete_block(blockArg)
```

Description

`delete_block(blockArg)` deletes the specified blocks from a system. Open the system before you delete blocks.

Examples

Delete Block Using Full Path Name

Delete the block Mu from the `vdp` system.

```
open_system('vdp')
delete_block('vdp/Mu')
```

Delete Block Using Block Handle

Delete the block Out2 from the `vdp` system using the block handle.

Open the `vdp` system.

```
open_system('vdp')
```

Interactively select the block Out1. Get the block's handle and assign it to the variable `Out1_handle`. Delete the block using the handle.

```
Out1_handle = get_param(gcb,'Handle');
delete_block(Out1_handle)
```

Delete Blocks Using Vector of Handles

Delete three blocks from the `vdp` system.

Open the vdp system. Add three blocks and assign their handles to variables.

```
open_system('vdp')
Constant_handle = add_block('built-in/Constant','vdp/MyConstant');
Gain_handle = add_block('built-in/Gain','vdp/MyGain');
Outport_handle = add_block('built-in/Outport','vdp/MyOutport');
```

Delete the blocks you added using a vector of handles.

```
delete_block([Constant_handle Gain_handle Outport_handle])
```

Input Arguments

`blockArg` — Blocks to delete

full path name | handle | vector of handles | 1-D cell array of handles or block path names

Blocks to delete, specified as the full block path name, a handle, a vector of handles, or a 1-D cell array of handles or block path names.

Example: `'vdp/Mu'`

Example: `[handle1 handle2]`

Example: `{'vdp/Mu' 'vdp/Out1' 'vdp/Out2'}`

See Also

`add_block`

Introduced before R2006a

delete_line

Delete line from Simulink 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`

Introduced before R2006a

delete_param

Delete system parameter added via `add_param` command

Syntax

```
delete_param('sys','parameter1','parameter2',...)
```

Description

This command deletes parameters that were added to the system using the `add_param` command. The command displays an error message if a specified parameter was not added with the `add_param` command.

Examples

The following example

```
add_param('vdp','DemoName','VanDerPolEquation','EquationOrder','2')
delete_param('vdp','DemoName')
```

adds the parameters `DemoName` and `EquationOrder` to the `vdp` system, then deletes `DemoName` from the system.

See Also

`add_param`

Introduced before R2006a

dependencies.fileDependencyAnalysis

Find model file dependencies

Syntax

```
files = dependencies.fileDependencyAnalysis('modelname')
[files, missing] = dependencies.fileDependencyAnalysis('modelname')
[files, missing, depfile] = dependencies.fileDependencyAnalysis('
modelname')
[files, missing, depfile, manifestfile] =
dependencies.fileDependencyAnalysis('modelname', 'manifestfile')
```

Description

`files = dependencies.fileDependencyAnalysis('modelname')` returns `files`, a cell array of 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`.

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`. If there is only one dependency, the return is a string. If there are no dependencies, the return is empty.

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 “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`.

More About

Tips

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.

- “What Are Model Dependencies?”

See Also

`dependencies.toolboxDependencyAnalysis`

Introduced in R2012a

dependencies.toolboxDependencyAnalysis

Find toolbox dependencies

Syntax

```
names = dependencies.toolboxDependencyAnalysis(files_in)
[names, folders] = dependencies.toolboxDependencyAnalysis(files_in)
```

Description

`names = dependencies.toolboxDependencyAnalysis(files_in)` returns `names`, a cell array of toolbox names required by the files in `files_in`.

`[names, folders] = dependencies.toolboxDependencyAnalysis(files_in)` returns toolbox names and also a cell array of the toolbox folders.

Input Arguments

`files_in`

Cell array of strings containing `.m`, `.mdl`, or `.slx` files on the MATLAB path. Simulink model names (without file extension) are also allowed.

Default: [ ]

Output Arguments

`names`

Cell array of toolbox names required by the files in `files_in`.

`folders`

(Optional) Cell-array of the required toolbox folders.

Examples

The following code reports the detectable required toolboxes for the model `vdp`:

```
files_in={'vdp'};
names = dependencies.toolboxDependencyAnalysis(files_in)
```

```
names =

    'MATLAB'    'Simulink'    'Simulink Coder'
```

To find all detectable toolbox dependencies of your model *and* the files it depends on:

1. Call `fileDependencyAnalysis` on your model.

   For example:

   ```
   files = dependencies.fileDependencyAnalysis('mymodel')
   ```

   ```
   files =
       'C:\Work\foo.m'
       'C:\Work\mymodel.mdl'
   ```

2. Call `toolboxDependencyAnalysis` on the `files` output of step 1.

   For example:

   ```
   tbxes = dependencies.toolboxDependencyAnalysis(files)
   ```

   ```
   tbxes =
   [1x24 char]    'MATLAB'    'Simulink Coder'    'Simulink'
   ```

   To view long product names examine the `tbxes` cell array as follows:

   ```
   tbxes{:}
   ```

   ```
   ans =
   Image Processing Toolbox

   ans =
   MATLAB

   ans =
   Simulink Coder

   ans =
   ```

```
Simulink
```

Alternatives

You can interactively run dependency analysis from the Simulink project. See “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`.

More About

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 2-160.

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.

- “What Are Model Dependencies?”

See Also

`dependencies.fileDependencyAnalysis`

Introduced in R2012a

detachConfigSet

Dissociate configuration set or configuration reference from model

Syntax

```
detachConfigSet(model, configObjName)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObjName`

The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`detachConfigSet` detaches the configuration set or configuration reference (configuration object) specified by `configObjName` from `model`. If no such configuration object is attached to the model, an error occurs.

Examples

The following example detaches the configuration object named `DevConfig` from the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
detachConfigSet(gcs, 'DevConfig');
```

More About

- "Manage a Configuration Set"

- “Manage a Configuration Reference”

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Introduced in R2006a

removeLabel(was detachLabelFromFile)

REMOVE — RENAMED TO REMOVELABEL — consolidate with existing removeLabel page. was: Detach label from Simulink Project file

Syntax

```
removeLabel(file,labelDefinition)
```

Description

`removeLabel(file,labelDefinition)` detaches the specified label `labelDefinition` from the file. Before you can detach the label, you need to get the label from the `file.Label` property or by using `findLabel`.

Examples

Detach a Label from a File

Remove a label from a particular project file.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a particular file by name.

```
myfile = findFile(proj,'models/AnalogControl.mdl')
```

```
myfile =

  ProjectFile with properties:

      Path: [1x86 char]
    Labels: [1x1 slproject.Label]
```

Get the `Labels` property of the file.

```
myfile.Labels
```

```
ans =

  Label with properties:

           File: 'C:\work\airframe\models\AnalogControl.mdl'
           Data: []
       DataType: 'none'
           Name: 'Design'
   CategoryName: 'Classification'
```

Attach the label `'To Review'` to the file.

```
addLabel(myfile,'Review','To Review')
```

Get the label you want to remove. Index into the Labels property to get the second label attached to the file.

```
 labeltoremove = myfile.Labels(2)
```

```
labeltoremove =

  Label with properties:

           File: [1x86 char]
           Data: []
       DataType: 'char'
           Name: 'To Review'
   CategoryName: 'Review'
```

Remove the label from the file.

```
removeLabel(myfile,labeltoremove)
myfile.Labels
```

```
ans =

  Label with properties:

           File: [1x86 char]
           Data: []
       DataType: 'none'
```

```
          Name: 'Design'
CategoryName: 'Classification'
```

Input Arguments

`file` — File to detach label from

file object

File to detach the label from, specified as a file object. You can get the file object by examining the project's Files property (`proj.Files`), or use `findFile` to find a file by name. The file must be within the root folder.

`labelDefinition` — Label to detach

label definition object

Name of the label to detach, specified as a label definition object returned by the `file.Label` property or `findLabel`.

See Also

Functions

`addLabel` | `createLabel` | `findFile` | `findLabel` | `simulinkproject`

disableimplicitsignalresolution

Convert model to use only explicit signal resolution

Syntax

```
retVal = disableimplicitsignalresolution('model')
retVal = disableimplicitsignalresolution('model', displayOnly)
```

Description

`retVal = disableimplicitsignalresolution('model')` inputs a model, reports all signals and states that implicitly resolve to signal objects, and converts the model to resolve only signals and states that explicitly require it. The report and any changes are limited to the model itself; they do not include blocks that are library links.

Before executing this function, ensure that all relevant Simulink data objects are defined in the base workspace. 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

More About

- "Data Validity Diagnostics Overview"
- "Symbol Resolution"

See Also

Simulink.Signal

Introduced in R2007a

docblock

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>" &'');')
```

More About

- "Use a Simulink DocBlock to Add a Comment"

See Also

DocBlock

Introduced in R2007a

export

Export Simulink Project to zip

Syntax

```
export(proj,zipFileName)
export(proj,zipFileName,definitionType)
```

Description

`export(proj,zipFileName)` exports the project `proj` to a zip file specified by `zipFileName`. The zip archive preserves the project files, structure, labels, and shortcuts, and does not include any source control information. You can use the zip archive to send the project to customers, suppliers, or colleagues who do not have access to your source control repository. Recipients can create a new project from the zip archive by clicking **New** in the Simulink Project Tool, and then in the start page, clicking **Archive**.

`export(proj,zipFileName,definitionType)` exports the project using the specified `definitionType` for the project definition files, single or multiple. If you do not specify `definitionType`, the project's current setting is used. Use the `definitionType` export option if you want to change project definition file management from the type selected when the project was created. You can control project definition file management in the preferences.

Examples

Export a Project to a Zip File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Export the project to a zip file.

```
export(proj,'airframeproj.zip')
```

- “Archive Projects in Zip Files”

Input Arguments

`proj` — Project
project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

`zipFileName` — Zip file name or path
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'`

Data Types: `char`

`definitionType` — Definition file type
slproject.DefinitionFiles.SingleFile | slproject.DefinitionFiles.MultiFile

Definition file type, specified as `slproject.DefinitionFiles.SingleFile` or `slproject.DefinitionFiles.MultiFile`. 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)`

Introduced in R2013a

findCategory

Get Simulink Project category of labels

Syntax

```
category = findCategory(proj,categoryName)
```

Description

`category = findCategory(proj,categoryName)` returns the project category specified by `categoryName`. You need to get a category before you can use `createLabel` or `removeLabel`.

Examples

Get a Category of Project Labels

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Use `findCategory` to get a category of labels by name.

```
category = findCategory(proj,'Classification')
```

```
category =

  Category with properties:

               Name: 'Classification'
           DataType: 'none'
    LabelDefinitions: [1x8 slproject.LabelDefinition]]
```

Alternatively, you can examine categories by index. Get the first category.

```
proj.Categories(1)
```

```
ans =
```

```
  Category with properties:

              Name: 'Classification'
          DataType: 'none'
    LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Find out what you can do with the category.

```
methods(category)
```

```
Methods for class slproject.Category:

createLabel          findLabel  removeLabel
```

Input Arguments

`proj` — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

`categoryName` — Name of category

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.

See Also

Functions

`createLabel` | `removeLabel` | `simulinkproject`

Introduced in R2013a

findFile

Get Simulink Project file by name

Syntax

```
file = findFile(proj,fileName)
```

Description

`file = findFile(proj,fileName)` returns a specific project file by name. You need to get a file before you can query labels, or use `addLabel` or `removeLabel`.

Examples

Find a File By Name

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Use `findFile` to get a file by name. You need to know the path if it is inside subfolders under the project root.

```
myfile = findFile(proj,'models/AnalogControl.mdl')
```

```
myfile =

  ProjectFile with properties:

      Path: [1x86 char]
    Labels: [1x1 slproject.Label]
  Revision: '2'
SourceControlStatus: Unmodified
```

Alternatively, you can examine files by index. Get the first file.

```
file = proj.Files(1);
```

Find out what you can do with the file.

```
methods(file)
```

```
Methods for class slproject.ProjectFile:

addLabel    removeLabel  findLabel
```

Input Arguments

`proj` — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

`fileName` — Path of file

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.

See Also

Functions

`addLabel` | `findCategory` | `findLabel` | `removeLabel` | `simulinkproject`

Introduced in R2013a

findLabel

Get Simulink Project file label

Syntax

```
label = findLabel(file,categoryName,labelName)
label = findLabel(file,labelDefinition)
label = findLabel(category,labelName)
```

Description

`label = findLabel(file,categoryName,labelName)` returns the label and its attached data for the label `labelName` in the category `categoryName` that is attached to the specified `file`. Use this syntax when you know the label name and category.

`label = findLabel(file,labelDefinition)` returns the file label and its attached data for the label name and category specified by `labelDefinition`. Use this syntax if you previously got a `labelDefinition` by accessing a `Labels` property, e.g., using a command like `myfile.Labels(1)`.

`label = findLabel(category,labelName)` returns the label definition of the label in this category specified by `labelName`. Returns an empty array if the label is not found.

Examples

Find Files with the Label `Utilility`

Find all project files with a particular label.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get the list of project files.

```
files = proj.Files;
```

Loop through each file. If the file has the extension `.m`, attach the label `Utility`.

```
for fileIndex = 1:numel(files)
   file = files(fileIndex);
   [~, ~, fileExtension] = fileparts(file.Path);
   if strcmp(fileExtension,'.m')
      addLabel(file,'Classification','Utility');
   end
end
```

Find all the files with the label `Utility` and add them to a list returned in `utility_files_to_review`.

```
utility_files_to_review = {};
for jj=1:numel(files)
   this_file = files(jj);

   label = findLabel(this_file,'Classification','Utility');

   if ( ~isempty(label))
      % This is a file labeled 'Utility'. Add to the
      % list of utility files.
      utility_files_to_review = [utility_files_to_review; this_file];
   end
end
```

Find a Label by Name or Definition

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a particular file by name.

```
myfile = findFile(proj,'models/AnalogControl.mdl');
```

Get a label by name.

```
label = findLabel(myfile,'Classification','Design');
```

Alternatively, examine the `Labels` property of the file to get an array of Label objects, one for each label attached to the file.

```
labels = myfile.Labels
```

Index into the Labels property to get the label attached to the particular file.

```
labeldefinition = myfile.Labels(1)
```

After you get the label definition from the Labels property, you can use it with `findLabel`.

```
label = findLabel(myfile,labeldefinition);
```

Find Labels by Name or Definition

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a category.

```
category = proj.Categories(1)
```

```
category =

  Category with properties:

               Name: 'Classification'
           DataType: 'none'
    LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Get a label definition.

```
ld = findLabel(category,'Design')
```

```
ld =

  LabelDefinition with properties:

           Name: 'Design'
    CategoryName: 'Classification'
```

Input Arguments

`file` — File to search labels of

file object

File to search the labels of, specified as a file object. You can get the file object by examining the project's Files property (`proj.Files`), or use `findFile` to get a file by name. The file must be in the project.

`categoryName` — Name of category
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.

See Also

Functions
`addLabel` | `createLabel` | `findFile` | `simulinkproject`

Introduced in R2013a

findLabelDefinition(renamed to findLabel)

Get Simulink Project label definition

Syntax

```
labelDefinition = findLabelDefinition(category,labelName)
```

Description

`labelDefinition = findLabelDefinition(category,labelName)` returns the label definition of the label in this category specified by `labelName`. Returns an empty array if the label is not found.

Examples

Find Labels by Name or Definition

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a category.

```
category = proj.Categories(1)
```

```
category =

  Category with properties:

               Name: 'Review'
           DataType: 'char'
    LabelDefinitions: [1x4 slproject.LabelDefinition]
```

Get a label definition.

```
ld = findLabelDefinition(category,'To Review')

ld =

  LabelDefinition with properties:

           Name: 'To Review'
   CategoryName: 'Review'
```

Alternatively, get a file and examine the `Labels` property to get an array of Label objects, one for each label attached to the file.

```
myfile = findFile(proj,'models/AnalogControl.mdl');
labels = myfile.Labels
```

Index into the Labels property to get the second label attached to the particular file.

```
labeldefinition = myfile.Labels(1)
```

After you get the label definition from the Labels property, you can use it with `findLabel`.

```
label = findLabel(myfile,labeldefinition);
```

Alternatively, get a particular file by name, and then get one of its labels by name.

```
myfile = findFile(proj,'models/AnalogControl.mdl');
label = findLabel(myfile,'Review','To Review');
```

Input Arguments

`labelName` — Name of label

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.

See Also

Functions

`addLabel` | `createLabel` | `findCategory` | `simulinkproject`

Introduced in R2013a

find_mdlrefs

Find Model blocks and referenced models at all levels or at top level only

Syntax

```
[refMdls,mdlBlks] = find_mdlrefs(system)
[refMdls,mdlBlks] = find_mdlrefs(system,Name,Value)
[refMdls,mdlBlks] = find_mdlrefs(system,allLevels)
```

Description

`[refMdls,mdlBlks] = find_mdlrefs(system)` finds all referenced models and Model blocks contained by the subsystem or model reference hierarchy that `system` is the top level of.

`[refMdls,mdlBlks] = find_mdlrefs(system,Name,Value)` finds referenced models and Model blocks with additional options specified by one or more `Name,Value` pair arguments.

`[refMdls,mdlBlks] = find_mdlrefs(system,allLevels)` specifies the levels of the system to search.

Tip The `find_mdlrefs` function provides two different ways to specify the levels of the system to search. Both techniques give the same results, but only the name and value technique allows you to control inclusion of protected and variant models in `refMdls`.

Examples

Find Referenced Models in Model Reference Hierarchy

Find referenced models and Model blocks for all models referenced by the specified model. Include all model reference variants.

```
open_system('sldemo_mdlref_variants_enum');
[myModels,myModelBlks] = find_mdlrefs('sldemo_mdlref_variants_enum',...
```

```
'AllLevels',true,'Variants','AllVariants')

myModels =

    'sldemo_mrv_linear_controller'
    'sldemo_mrv_nonlinear_controller'
    'sldemo_mrv_sig_filter1_production'
    'sldemo_mrv_sig_filter1_prototype'
    'sldemo_mrv_sig_filter2_production'
    'sldemo_mrv_sig_filter2_prototype'
    'sldemo_mrv_sig_filter3_production'
    'sldemo_mrv_sig_filter3_prototype'
    'sldemo_mdlref_variants_enum'


myModelBlks =

    'sldemo_mdlref_variants_enum/Controller'
    'sldemo_mdlref_variants_enum/Filter1'
    'sldemo_mdlref_variants_enum/Filter2'
    'sldemo_mdlref_variants_enum/Filter3'
```

- “Set up Model Variants”
- “Protected Model”

Input Arguments

`system` — System to search

string | handle

System to search, specified as a string or a handle.

- The string can be the path to a Model block, subsystem, or a model in a model reference hierarchy.
- The handle can be for a Model block, subsystem, or model in a model reference hierarchy.

`allLevels` — Levels to search

`true` (default) | `false`

Levels to search, specified as `true` or `false`.

- `true` — Search all Model blocks in the model reference hierarchy for which the system is the top model.
- `false` — Search only the top-level system.

Data Types: `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `refModels = find_mdlrefs(top_model,'Variants',true)`

`'AllLevels'` — Levels to search

`true` (default) | `false`

Levels to search, specified as a `true` or `false`.

- `true` — Search all Model blocks in the model reference hierarchy for which the system is the top model.
- `false` — Search only the top-level system.

Data Types: `logical`

`'IncludeProtectedModels'` — Include protected models in search results

`false` (default) | `true`

Include protected models in search, specified as `true` or `false`. This setting does not affect the list of Model blocks returned.

Data Types: `logical`

`'Variants'` — Include variants in search

`'ActivePlusCodeVariants'` (default) | `'ActiveVariants'` | `'AllVariants'`

Include variants in search, specified as `'ActivePlusCodeVariants'`, `'ActiveVariants'`, or `'AllVariants'`.

- `'ActivePlusCodeVariants'` — Include all variants for `Model Variants` blocks for which you select the **Generate preprocessor conditionals** option.

- `'ActiveVariants'` — Include the active variant for `Model Variants` blocks.
- `'AllVariants'` — Include all variants for `Model Variants` blocks.

`'IncludeCommented'` — Include commented blocks in search
`'off'` (default) | `'on'`

Include commented blocks in search, specified as `'off'` or `'on'`.

Output Arguments

`refMdls` — Names of referenced models
cell array of strings

Names of referenced models, returned as a cell array of strings. The last element is the system you specified in the `system` input argument or the parent model of that system.

`mdlBlks` — Names of Model blocks
cell array of strings

Names of Model blocks, returned as a cell array of strings.

More About

- "Model Reference"

See Also

`find_system` | `Model`

Introduced before R2006a

find_system

Find systems, blocks, lines, ports, and annotations

Syntax

```
Objects = find_system
Objects = find_system(System)
Objects = find_system(Name,Value)
Objects = find_system(System,Name,Value)
```

Description

`Objects = find_system` returns loaded systems and their blocks, including subsystems.

`Objects = find_system(System)` returns the specified system and its blocks.

`Objects = find_system(Name,Value)` returns loaded systems and the objects in those systems that meet the criteria specified by one or more `Name,Value` pair arguments. You can use this syntax to specify search constraints and to search for specific parameter values. Specify the search constraints before the parameter and value pairs.

`Objects = find_system(System,Name,Value)` returns the objects in the specified system that meet the specified criteria.

Input Arguments

`System` — System to search

path name | cell array of path names | handle | vector of handles

System to search, specified as the full system path name, a cell array of system path names, a handle, or a vector of handles.

Example: `'MyModel/Subsystem1'`

Example: `{'vdp','fuelsys'}`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

When you use the `find_system` function, `Name,Value` pair arguments can include search constraints and parameter name and value pairs. You can specify search constraints in any order, but they must precede the parameter name and value pairs.

See "Model Parameters" on page 6-2 and "Block-Specific Parameters" on page 6-101 for the lists of model and block parameters.

Example: `'SearchDepth','0','LookUnderMasks','none','BlockType','Goto'` searches in loaded systems, excluding masked subsystems, for Goto blocks.

The table lists the possible search constraint pairs. Braces indicate default values.

Name	Value Type	Description
`'BlockDialogParams'`	string	Search block dialog box parameters for the specified value. This pair, like parameter and value pairs, must follow the other search constraint pairs.
`'CaseSensitive'`	`{'on'} | 'off'`	If `'on'`, search considers case when matching search strings.
`'FindAll'`	`'on' | {'off'}`	If `'on'`, search includes lines, ports, and annotations within systems. `find_system` returns a vector of handles when this option is set to `'on'`, regardless of how you specify `System`.
`'FollowLinks'`	`'on' | {'off'}`	If `'on'`, search follows links into library blocks. If you do not specify a system to search, `find_system` includes loaded libraries in the results, whether you set `'FollowLinks'` to `'on'` or `'off'`. You can use `'FollowLinks'` with

Name	Value Type	Description
		`'LookUnderMasks'` to update library links in subsystems. See “Update Library Links in a Subsystem” on page 2-197.
`'IncludeCommented'`	`'on' | {'off'}`	Specify whether to include commented blocks in the search.
`'LoadFullyIfNeeded'`	`{'on'} | 'off'`	If `'on'`, attempts to load any partially loaded models. If `'off'`, disables model loading. Use this pair, for example, to prevent load warnings.
`'LookUnderMasks'`	`{'graphical'}`	Search includes masked subsystems that have no workspaces and no dialogs.
	`'none'`	Search skips masked subsystems.
	`'functional'`	Search includes masked subsystems that do not have dialogs.
	`'all'`	Search includes all masked subsystems.
`'RegExp'`	`'on' | {'off'}`	If `'on'`, search treats search expressions as regular expressions. To learn more about MATLAB regular expressions, see “Regular Expressions”.
`'SearchDepth'`	positive integer string	Restricts the search depth to the specified level (`'0'` for loaded systems only, `'1'` for blocks and subsystems of the top-level system, `'2'` for the top-level system and its children, etc.). The default is all levels.
`'Variants'` This search constraint applies only to variant subsystems and model variants.	`{'ActiveVariants'}`	Search in only the active variant subsystems.
	`'AllVariants'`	Search in all variants.

Name	Value Type	Description
	`'ActivePlusCodeVariants'`	Search all variants if any generate preprocessor conditionals. Otherwise, search only the active variant.

Output Arguments

`Objects` — Matching objects

cell array of path names | vector of handles

Matching objects found, returned as:

- A cell array of path names if you specified `System` as a path name or cell array of path names, or if you did not specify a system
- A vector of handles if you specified `System` as a handle or vector of handles

Examples

Find Loaded Systems and Their Blocks

Return the names of all loaded systems and their blocks.

```
load_system('vdp')
find_system
```

Returns loaded systems and libraries, including `vdp`.

Find Specific System and Its Blocks

Return `vdp` system and its blocks.

```
load_system({'vdp','fuelsys'})
find_system('vdp')
```

```
ans =

    'vdp'
```

```
    'vdp/Fcn'
    'vdp/More Info'
    'vdp/More Info/Model Info'
    'vdp/Mu'
    'vdp/Mu1'
    'vdp/Mux'
    'vdp/Product'
    'vdp/Scope'
    'vdp/Sum'
    'vdp/x1'
    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'
```

Return Names of Loaded Models

Return the names of only the loaded models, i.e., block diagrams. This command returns library names as well, because libraries are treated as models.

```
vdp
open_bd = find_system('type','block_diagram')
```

```
 ans =
    'vdp'
    'simulink'
```

Search Children of Subsystem

Return 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')
```

```
ans =
  'sldemo_clutch/Unlocked/Goto'
  'sldemo_clutch/Unlocked/Goto1'
```

Search Using Multiple Criteria

Search in the `vdp` system and return the names of all `Gain` blocks whose **Gain** value is set to `1`.

```
vdp
find_system('vdp','BlockType','Gain','Gain','1')

ans =
  'vdp/Mu'
```

Return Lines and Annotations as Handles

Get the handles of all lines and annotations in the `vdp` system. With `'FindAll'`, the function returns handles regardless of how you specify the system to search.

```
vdp
L = find_system('vdp','FindAll','on','type','line')
A = find_system('vdp','FindAll','on','type','annotation')

L =

   1.0e+03 *

    2.0740
    2.0730
    2.0720
    2.0710
    2.0700
    2.0690
    2.0680
    2.0670
    2.0660
    2.0650
    2.0640
    2.0630
    2.0620
    2.0610
    2.0600
    2.0590
    2.0580


A =

   1.0e+03 *

    2.0760
    2.0750
```

Search for Specific Block Parameter Value

Find any block dialog box parameters with a value of `0` in the `vdp` and `fuelsys` systems.

```
load_system({'vdp','fuelsys'})
find_system({'vdp','fuelsys'},'BlockDialogParams','0')
```

```
ans =

    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'
    'fuelsys/Constant2'
    'fuelsys/Constant4'
    'fuelsys/Constant5'
    'fuelsys/engine …'
    'fuelsys/engine …'
    'fuelsys/engine …'
    'fuelsys/engine …'
    .
    .
    .
```

Search Using Regular Expressions

Find all blocks in the top level of the currently loaded systems with a block dialog parameter value that starts with `3`.

```
load_system({'fuelsys','vdp'})
find_system('SearchDepth','1','regexp','on','BlockDialogParams','^3')
```

```
ans =
  'fuelsys/Nominal…'
  'vdp/Scope'
```

Regular Expression Search for Partial Match

When you search using regular expressions, you can specify a part of the string you want to match to return all objects that contain that string.

Find all the inport and outport blocks in the `sldemo_clutch` model.

```
sldemo_clutch
```

```
find_system('sldemo_clutch','regexp','on','blocktype','port')
```

Update Library Links in a Subsystem

In this example, `myModel` contains a single subsystem, which is a library link. After the model was last opened, a `Gain` block was added to the corresponding subsystem in the library.

Open the model. Use `find_system` with `'FollowLinks'` set to `'off'`. The command does not follow the library links into the subsystem and returns only the subsystem at the top level.

```
open_system('myModel')
find_system(bdroot,'LookUnderMasks','on','FollowLinks', 'off')
```

```
ans =

    'myModel'
    'myModel/Subsystem'
```

Use `find_system` with `'FollowLinks'` set to `'on'`. `find_system` updates the library links and returns the block in the subsystem.

```
find_system(bdroot,'LookUnderMasks','on','FollowLinks','on')
```

```
Updating Link: myModel/Subsystem/Gain
Updating Link: myModel/Subsystem/Gain

ans =

    'myModel'
    'myModel/Subsystem'
    'myModel/Subsystem/Gain'
```

Return Values as Handles

Provide the system to `find_system` as a handle. Search for block dialog box parameters with a value of `0`.

If you make multiple calls to `get_param` for the same block, then using the block handle is more efficient than specifying the full block path as a string.

```
vdp
```

```
sys = get_param('vdp','Handle');
find_system(sys,'BlockDialogParams','0')

ans =
  14.0001
  15.0001
  16.0001
```

More About

- “Regular Expressions”
- “Model Parameters” on page 6-2
- “Block-Specific Parameters” on page 6-101

See Also

`find_mdlrefs` | `get_param` | `getSimulinkBlockHandle` | `set_param`

Introduced before R2006a

fixdt

Create `Simulink.NumericType` object describing fixed-point or floating-point data type

Syntax

```
a = fixdt(Signed, WordLength)
a = fixdt(Signed, WordLength, FractionLength)
a = fixdt(Signed, WordLength, TotalSlope, Bias)
a = fixdt(Signed, WordLength, SlopeAdjustmentFactor, FixedExponent,
Bias)
a = fixdt(DataTypeNameString)
a = fixdt(..., 'DataTypeOverride', 'Off')
[DataType,IsScaledDouble] = fixdt(DataTypeNameString)
[DataType,IsScaledDouble] = fixdt(DataTypeNameString,
'DataTypeOverride', 'Off')
```

Description

`a = fixdt(Signed, WordLength)` returns a `Simulink.NumericType` object describing a fixed-point data type with unspecified scaling. The scaling would typically be determined by another block parameter. `Signed` can be `0` (false) for unsigned or `1` (true) for signed.

`a = fixdt(Signed, WordLength, FractionLength)` returns a `Simulink.NumericType` object describing a fixed-point data type with binary point scaling. `FractionLength` can be greater than `WordLength`. For more information, see "Binary Point Interpretation".

`a = fixdt(Signed, WordLength, TotalSlope, Bias)` or `a = fixdt(Signed, WordLength, SlopeAdjustmentFactor, FixedExponent, Bias)` returns a `Simulink.NumericType` object describing a fixed-point data type with slope and bias scaling.

`a = fixdt(DataTypeNameString)` returns a `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type name. The data type name can be either the name of a built-in Simulink data

type or the name of a fixed-point data type that conforms to the naming convention for fixed-point names established by the Fixed-Point Designer product. For more information, see "Fixed-Point Data Type and Scaling Notation" in the Fixed-Point Designer documentation.

`a = fixdt(..., 'DataTypeOverride', 'Off')` returns a `Simulink.NumericType` object with its `DataTypeOverride` parameter set to `Off`. The default value for this property is `Inherit`. You can specify the `DataTypeOverride` parameter after any combination of other input parameters.

`[DataType,IsScaledDouble] = fixdt(DataTypeNameString)` returns a `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type name and a flag that indicates whether the specified data type name was the name of a scaled double data type.

`[DataType,IsScaledDouble] = fixdt(DataTypeNameString, 'DataTypeOverride', 'Off')` returns:

- A `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type name. The `DataTypeOverride` parameter of the `Simulink.NumericType` object is set to `Off`.
- A flag that indicates whether the specified data type name was the name of a scaled double data type.

Examples

Return a `Simulink.NumericType` object describing a fixed-point data type with unspecified scaling:

```
a = fixdt(1,16)

a =

Simulink.NumericType
    DataTypeMode: 'Fixed-point: unspecified scaling'
      Signedness: 'Signed'
      WordLength: 16
         IsAlias: false
      HeaderFile: ''
     Description: ''
```

Return a `Simulink.NumericType` object describing a fixed-point data type with binary point scaling :

```
a = fixdt(1,16,2)

a =

Simulink.NumericType
    DataTypeMode: 'Fixed-point: binary point scaling'
      Signedness: 'Signed'
      WordLength: 16
  FractionLength: 2
         IsAlias: false
      HeaderFile: ''
     Description: ''
```

Return a `Simulink.NumericType` object describing a fixed-point data type with slope and bias scaling:

```
a = fixdt(1, 16, 2^-2, 4)

a =

Simulink.NumericType
    DataTypeMode: 'Fixed-point: slope and bias scaling'
      Signedness: 'Signed'
      WordLength: 16
           Slope: 0.25
            Bias: 4
         IsAlias: false
      HeaderFile: ''
     Description: ''
```

Return a `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type name:

```
[DataType,IsScaledDouble] = fixdt('ufix8')

DataType =

Simulink.NumericType
    DataTypeMode: 'Fixed-point: binary point scaling'
      Signedness: 'Unsigned'
      WordLength: 8
  FractionLength: 0
```

```
         IsAlias: false
      HeaderFile: ''
     Description: ''
IsScaledDouble =

     0
```

Return a `Simulink.NumericType` object with its `DataTypeOverride` property set to `Off`:

```
 a = fixdt(0, 8, 2, 'DataTypeOverride', 'Off')

a =

Simulink.NumericType
    DataTypeMode: 'Fixed-point: binary point scaling'
      Signedness: 'Unsigned'
      WordLength: 8
  FractionLength: 2
DataTypeOverride: Off
         IsAlias: false
      HeaderFile: ''
     Description: ''
```

See Also

`float` | `sint` | `ufix` | `sfix` | `sfrac` | `ufrac` | `uint`

Introduced before R2006a

fixpt_evenspace_cleanup

Modify breakpoints of lookup table to have even spacing

Syntax

```
xdata_modified = fixpt_evenspace_cleanup(xdata,xdt,xscale)
```

Description

`xdata_modified = fixpt_evenspace_cleanup(xdata,xdt,xscale)` modifies breakpoints of a lookup table to have even spacing after quantization. By adjusting breakpoints to have even spacing after quantization, Simulink Coder generated code can exclude breakpoints from memory.

`xdata` is the breakpoint vector of a lookup table to make evenly spaced, such as `0:0.005:1`. `xdt` is the data type of the breakpoints, such as `sfix(16)`. `xscale` is the scaling of the breakpoints, such as `2^-12`. Using these three inputs, `fixpt_evenspace_cleanup` returns the modified breakpoints in `xdata_modified`.

This function works only with nontunable data and considers data to have even spacing relative to the scaling slope. For example, the breakpoint vector [0 2 5], which has spacing value 2 and 3, appears to have uneven spacing. However, the difference between the maximum spacing 3 and the minimum spacing 2 equals 1. If the scaling slope is 1 or greater, a spacing variation of 1 represents a 1-bit change or less. In this case, the `fixpt_evenspace_cleanup` function considers a spacing variation of 1 bit or less to be even.

Modifications to breakpoints can change the numerical behavior of a lookup table. To check for changes, test the model using simulation, rapid prototyping, or other appropriate methods.

Examples

Modify breakpoints of a lookup table to have even spacing after quantization:

```
xdata = 0:0.005:1;
```

```
xdt = sfix(16);
xscale = 2^-12;
xdata_modified = fixpt_evenspace_cleanup(xdata,xdt,xscale)
```

See Also

fixdt | fixpt_interp1 | fixpt_look1_func_approx | fixpt_look1_func_plot | sfix | ufix

Introduced before R2006a

fixpt_interp1

Implement 1-D lookup table

Syntax

```
y = fixpt_interp1(xdata,ydata,x,xdt,xscale,ydt,yscale,rndmeth)
```

Description

`y = fixpt_interp1(xdata,ydata,x,xdt,xscale,ydt,yscale,rndmeth)` implements a one-dimensional lookup table to find output `y` for input `x`. If `x` falls between two `xdata` values (breakpoints), `y` is the result of interpolating between the corresponding `ydata` values. If `x` is greater than the maximum value in `xdata`, `y` is the maximum `ydata` value. If `x` is less than the minimum value in `xdata`, `y` is the minimum `ydata` value.

If the input data type `xdt` or the output data type `ydt` is floating point, `fixpt_interp1` performs the interpolation using floating-point calculations. Otherwise, `fixpt_interp1` uses integer-only calculations. These calculations handle the input scaling `xscale` and the output scaling `yscale` and obey the rounding method `rndmeth`.

Input Arguments

`xdata`

Vector of breakpoints for the lookup table, such as `linspace(0,8,33)`.

`ydata`

Vector of table data that correspond to the breakpoints for the lookup table, such as `sin(xdata)`.

`x`

Vector of input values for the lookup table to process, such as `linspace(-1,9,201)`.

xdt

Data type of input `x`, such as `sfix(8)`.

xscale

Scaling for input `x`, such as `2^-3`.

ydt

Data type of output `y`, such as `sfix(16)`.

yscale

Scaling for output `y`, such as `2^-14`.

rndmeth

Rounding mode supported by fixed-point Simulink blocks:

`'Ceiling'`	Round to the nearest representable number in the direction of positive infinity.
`'Floor'` (default)	Round to the nearest representable number in the direction of negative infinity.
`'Nearest'`	Round to the nearest representable number.
`'Toward Zero'`	Round to the nearest representable number in the direction of zero.

Examples

Interpolate outputs for `x` using a 1-D lookup table that approximates the sine function:

```
xdata = linspace(0,8,33).';
ydata = sin(xdata);
% Define input x as a vector of 201 evenly
% spaced points between -1 and 9 (includes
% values both lower and higher than the range
% of breakpoints in xdata)
x = linspace(-1,9,201).';
```

```
% Interpolate output values for x
y = fixpt_interp1(xdata,ydata,x,sfix(8),2^-3,sfix(16),...
 2^-14,'Floor')
```

See Also

`fixpt_evenspace_cleanup` | `fixpt_look1_func_approx` | `fixpt_look1_func_plot`

Introduced before R2006a

fixpt_look1_func_approx

Optimize fixed-point approximation of nonlinear function by interpolating lookup table data points

Syntax

```
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax)
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[])
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax)
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...
xmin,xmax,xdt,xscale,ydtydt,yscale,rndmeth,errmax,nptsmax,spacing)
```

Description

`[xdata,ydata,errworst] = fixpt_look1_func_approx('func',... xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax)` returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error and maximum number of points that you specify.

`[xdata,ydata,errworst] = fixpt_look1_func_approx('func',... xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[])` returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error that you specify.

`[xdata,ydata,errworst] = fixpt_look1_func_approx('func',... xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax)` returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum number of points that you specify.

`[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...`

xmin,*xmax*,*xdt*,*xscale*,*ydtydt*,*yscale*,*rndmeth*,*errmax*,*nptsmax*,*spacing*) returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error, maximum number of points, and breakpoint spacing that you specify.

In each case, `fixpt_look1_func_approx` interpolates between lookup table data points to optimize the fixed-point approximation. The inputs *xmin* and *xmax* specify the range over which to approximate the breakpoints. The inputs *xdt*, *xscale*, *ydt*, *yscale*, and *rndmeth* follow conventions used by fixed-point Simulink blocks.

The inputs *errmax*, *nptsmax*, and *spacing* are optional. Of these inputs, you must specify at least *errmax* or *nptsmax*. If you omit one of those two inputs, you must use brackets, `[]`, in place of the omitted input. `fixpt_look1_func_approx` ignores that requirement for the lookup table.

If you do not specify spacing, and more than one spacing satisfies *errmax* and *nptsmax*, `fixpt_look1_func_approx` chooses in this order: power-of-2 spacing, even spacing, uneven spacing. This behavior applies when you specify both *errmax* and *nptsmax*, but not when you specify just one of the two.

Input Arguments

`func`

Function of `x` for which to approximate breakpoints. Enclose this expression in single quotes, for example, `'sin(2*pi*x)'`.

`xmin`

Minimum value of `x`.

`xmax`

Maximum value of `x`.

`xdt`

Data type of `x`.

xscale

Scaling for the `x` values.

ydt

Data type of `y`.

yscale

Scaling for the `y` values.

rndmeth

Rounding mode supported by fixed-point Simulink blocks:

`'Ceiling'`	Round to the nearest representable number in the direction of positive infinity.
`'Floor'` (default)	Round to the nearest representable number in the direction of negative infinity.
`'Nearest'`	Round to the nearest representable number.
`'Toward Zero'`	Round to the nearest representable number in the direction of zero.

errmax

Maximum acceptable error between the ideal function and the approximation given by the lookup table.

nptsmax

Maximum number of points for the lookup table.

spacing

Spacing of breakpoints for the lookup table:

`'even'`	Even spacing
`'pow2'`	Even, power-of-2 spacing

`'unrestricted'` (default) Uneven spacing

If you specify...	The breakpoints of the lookup table...
`errmax` and *`nptsmax`*	Meet both criteria, if possible. The *`errmax`* requirement has higher priority than *`nptsmax`*. If the breakpoints cannot meet both criteria with the specified spacing, *`nptsmax`* does not apply.
`errmax` only	Meet the error criteria, and `fixpt_look1_func_approx` returns the fewest number of points.
`nptsmax` only	Meet the points criteria, and `fixpt_look1_func_approx` returns the smallest worst-case error.

Output Arguments

`xdata`

Vector of breakpoints for the lookup table.

`ydata`

Vector of values from applying the ideal function to the breakpoints.

`errworst`

Worst-case error, which is the maximum absolute error between the ideal function and the approximation given by the lookup table.

Examples

Approximate a fixed-point sine function using a lookup table:

```
func = 'sin(2*pi*x)';
% Define the range over which to optimize breakpoints
```

```
xmin = 0;
xmax = 0.25;
% Define the data type and scaling for the inputs
xdt = ufix(16);
xscale = 2^-16;
% Define the data type and scaling for the outputs
ydt = sfix(16);
yscale = 2^-14;
% Specify the rounding method
rndmeth = 'Floor';
% Define the maximum acceptable error
errmax = 2^-10;
% Choose even, power-of-2 spacing for breakpoints
spacing = 'pow2';
% Create the lookup table
[xdata,ydata,errworst] = fixpt_look1_func_approx(func,...
 xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

More About

- "Use Lookup Table Approximation Functions"

See Also

`fixpt_evenspace_cleanup` | `fixpt_interp1` | `fixpt_look1_func_plot`

Introduced before R2006a

fixpt_look1_func_plot

Plot fixed-point approximation function for lookup table

Syntax

```
fixpt_look1_func_plot(xdata,ydata,'func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)
errworst = fixpt_look1_func_plot(xdata,ydata,'func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)
```

Description

`fixpt_look1_func_plot(xdata,ydata,'func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)` plots a lookup table approximation function and the error from the ideal function.

`errworst = fixpt_look1_func_plot(xdata,ydata,'func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)` plots a lookup table approximation function and the error from the ideal function. The output `errworst` is the maximum absolute error.

You can use `fixpt_look1_func_approx` to generate `xdata` and `ydata`, the breakpoints and table data for the lookup table, respectively. `fixpt_look1_func_approx` applies the ideal function to the breakpoints in `xdata` to produce `ydata`. While this method is the easiest way to generate `ydata`, you can choose other values for `ydata` as input for `fixpt_look1_func_plot`. Choosing different values for `ydata` can, in some cases, produce a lookup table with a smaller maximum absolute error.

Input Arguments

xdata

Vector of breakpoints for the lookup table.

ydata

Vector of values from applying the ideal function to the breakpoints.

func

Function of `x` for which to approximate breakpoints. Enclose this expression in single quotes, for example, `'sin(2*pi*x)'`.

xmin

Minimum value of `x`.

xmax

Maximum value of `x`.

xdt

Data type of `x`.

xscale

Scaling for the `x` values.

ydt

Data type of `y`.

yscale

Scaling for the `y` values.

rndmeth

Rounding mode supported by fixed-point Simulink blocks:

`'Ceiling'`	Round to the nearest representable number in the direction of positive infinity.
`'Floor'` (default)	Round to the nearest representable number in the direction of negative infinity.

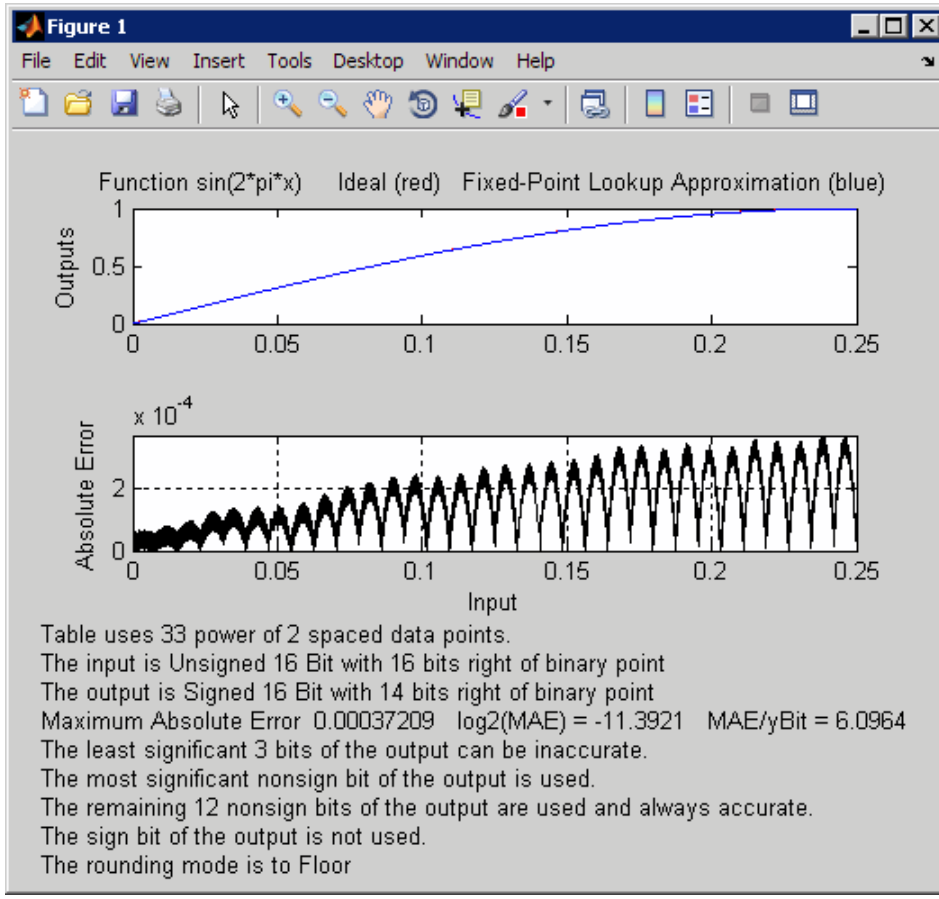
`'Nearest'`	Round to the nearest representable number.
`'Toward Zero'`	Round to the nearest representable number in the direction of zero.

Examples

Plot a fixed-point approximation of the sine function using data points generated by `fixpt_look1_func_approx`:

```
func = 'sin(2*pi*x)';
% Define the range over which to optimize breakpoints
xmin = 0;
xmax = 0.25;
% Define the data type and scaling for the inputs
xdt = ufix(16);
xscale = 2^-16;
% Define the data type and scaling for the outputs
ydt = sfix(16);
yscale = 2^-14;
% Specify the rounding method
rndmeth = 'Floor';
% Define the maximum acceptable error
errmax = 2^-10;
% Choose even, power-of-2 spacing for breakpoints
spacing = 'pow2';
% Generate data points for the lookup table
[xdata,ydata,errworst]=fixpt_look1_func_approx(func,...
 xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
% Plot the sine function (ideal and fixed-point) & errors
fixpt_look1_func_plot(xdata,ydata,func,xmin,xmax,...
 xdt,xscale,ydt,yscale,rndmeth);
```

`fixpt_look1_func_plot` plots the fixed-point sine function, using generated data points, and plots the error between the ideal function and the fixed-point function. The maximum absolute error and the number of points required appear on the plot. The error drops to zero at a breakpoint, but increases between breakpoints due to curvature differences between the ideal function and the line drawn between breakpoints.



The lookup table requires 33 points to achieve a maximum absolute error of `2^-11.3922`.

More About

- "Use Lookup Table Approximation Functions"

See Also

`fixpt_evenspace_cleanup` | `fixpt_interp1` | `fixpt_look1_func_approx`

Introduced before R2006a

fixpt_set_all

Set property for each fixed-point block in subsystem

Syntax

`fixpt_set_all(`*`SystemName`*`,`*`fixptPropertyName`*`,`*`fixptPropertyValue`*`)`

Description

`fixpt_set_all(`*`SystemName`*`,`*`fixptPropertyName`*`,`*`fixptPropertyValue`*`)` sets the property `fixptPropertyName` of every applicable block in the model or subsystem *`SystemName`* to the value *`fixptPropertyValue`*

Examples

Set each fixed-point block in a model `Filter_1` to round towards the floor and saturate upon overflow:

```
% Round towards the floor
fixpt_set_all('Filter_1','RndMeth','Floor')

% Saturate upon overflow
fixpt_set_all('Filter_1','DoSatur','on')
```

Introduced before R2006a

fixptbestexp

Exponent that gives best precision for fixed-point representation of value

Syntax

```
out = fixptbestexp(RealWorldValue, TotalBits, IsSigned)
out = fixptbestexp(RealWorldValue, FixPtDataType)
```

Description

`out = fixptbestexp(RealWorldValue, TotalBits, IsSigned)` returns the exponent that gives the best precision for the fixed-point representation of *`RealWorldValue`*. *`TotalBits`* specifies the number of bits for the fixed-point number. *`IsSigned`* specifies whether the fixed-point number is signed: `1` indicates the number is signed and `0` indicates the number is not signed.

`out = fixptbestexp(RealWorldValue, FixPtDataType)` returns the exponent that gives the best precision based on the data type *`FixPtDataType`*.

Examples

Get the exponent that gives the best precision for the real-world value 4/3 using a signed, 16-bit number:

```
out = fixptbestexp(4/3,16,1)
out =
   -14
```

Alternatively, specify the fixed-point data type:

```
out = fixptbestexp(4/3,sfix(16))
out =
   -14
```

This shows that the maximum precision representation of 4/3 is obtained by placing 14 bits to the right of the binary point:

01.01010101010101

You can specify the precision of this representation in fixed-point blocks by setting the scaling to `2^-14` or `2^fixptbestexp(4/3,16,1)`.

See Also

`fixptbestprec`

Introduced before R2006a

fixptbestprec

Determine maximum precision available for fixed-point representation of value

Syntax

```
out = fixptbestprec(RealWorldValue,TotalBits,IsSigned)
out = fixptbestprec(RealWorldValue,FixPtDataType)
```

Description

`out = fixptbestprec(RealWorldValue,TotalBits,IsSigned)` determines the maximum precision for the fixed-point representation of the real-world value specified by `RealWorldValue`. You specify the number of bits for the fixed- point number with `TotalBits`, and you specify whether the fixed-point number is signed with `IsSigned`. If `IsSigned` is `1`, the number is signed. If `IsSigned` is `0`, the number is not signed. The maximum precision is returned to `out`.

`out = fixptbestprec(RealWorldValue,FixPtDataType)` determines the maximum precision based on the data type specified by `FixPtDataType`.

Examples

Example 1

The following command returns the maximum precision available for the real-world value 4/3 using a signed, 8-bit number:

```
out = fixptbestprec(4/3,8,1)
out =
   0.015625
```

Alternatively, you can specify the fixed-point data type:

```
out = fixptbestprec(4/3,sfix(8))
out =
```

```
0.015625
```

This value means that the maximum precision available for 4/3 is obtained by placing six bits to the right of the binary point since 2^{-6} equals 0.015625:

01.010101

Example 2

You can use the maximum precision as the scaling in fixed-point blocks. This enables you to use `fixptbestprec` to perform a type of autoscaling if you would like to designate a known range of your simulation. For example, if your known range is -13 to 22, and you are using a safety margin of 30%:

```
knownMax = 22;
knownMin = -13;
localSafetyMargin = 30;
slope = max( fixptbestprec( (1+localSafetyMargin/100)* ...
 [knownMax,knownMin], sfix(16) ) );
```

The variable `slope` can then be used in the expression that you specify for the **Output data type** parameter in a block mask. Be sure to select the **Lock output data type setting against changes by the fixed-point tools** check box in the same block to prevent the Fixed-Point Tool from overriding the scaling. If you know the range, you can use this technique in place of relying on a model simulation to provide the range to the autoscaling tool, as described in `autofixexp` in the Fixed-Point Designer documentation.

See Also

`fixptbestexp`

Introduced before R2006a

float

Create `Simulink.NumericType` object describing floating-point data type

Syntax

```
a = float('single')
a = float('double')
```

Description

`a = float('single')` returns a `Simulink.NumericType` object that describes the data type of an IEEE single (32 total bits, 8 exponent bits).

`a = float('double')` returns a `Simulink.NumericType` object that describes the data type of an IEEE double (64 total bits, 11 exponent bits).

Note: `float` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `float('single')` with `fixdt('single')` and `float('double')` with `fixdt('double')`.

Examples

Define an IEEE single data type.

```
>> a = float('single')

a =

  NumericType with properties:

    DataTypeMode: 'Single'
         IsAlias: 0
       DataScope: 'Auto'
      HeaderFile: ''
     Description: ''
```

See Also

`fixdt` | `Simulink.NumericType` | `sfix` | `sfrac` | `sint` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

frameedit

Edit print frames for Simulink and Stateflow block diagrams

Syntax

```
frameedit
frameedit filename
```

Description

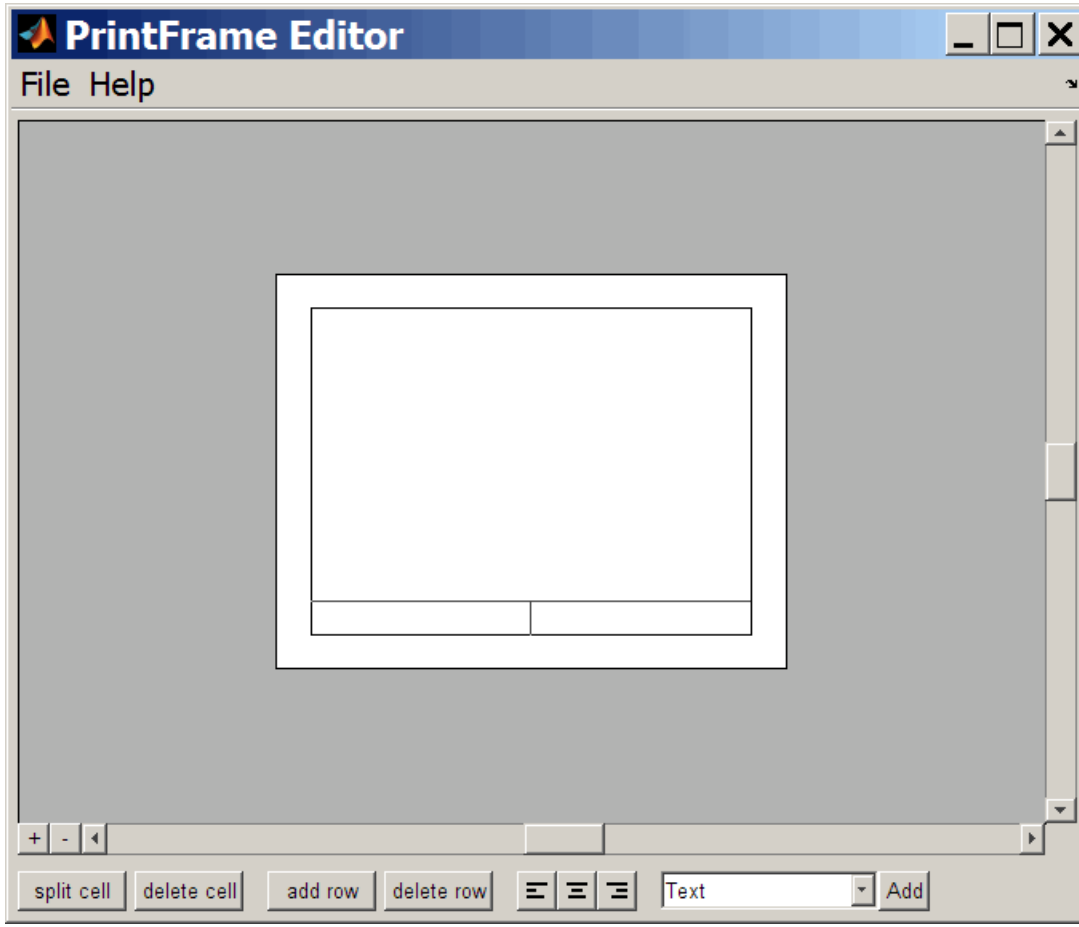
`frameedit` starts the PrintFrame Editor, a graphical user interface you use to create borders for Simulink and Stateflow block diagrams. With no argument, `frameedit` opens the **PrintFrame Editor** window with a new file.

`frameedit filename` opens the **PrintFrame Editor** window with the specified filename, where `filename` is a figure file (`.fig`) previously created and saved using `frameedit`.

More About

Tips

This illustrates the main features of the PrintFrame Editor.



Closing the PrintFrame Editor

To close the **PrintFrame Editor** window, click the close box in the upper right corner, or select **Close** from the **File** menu.

Printing Simulink Block Diagrams with Print Frames

Select **Print** from the Simulink **File** menu. Check the **Frame** box and supply the filename for the print frame you want to use. Click **OK** in the **Print** dialog box.

Getting Help for the PrintFrame Editor

For further instructions on using the PrintFrame Editor, select **PrintFrame Editor Help** from the **Help** menu in the PrintFrame Editor.

Introduced in R2008b

fxptdlg

Start Fixed-Point Tool

Syntax

```
fxptdlg('modelname')
```

Description

`fxptdlg('modelname')` starts the Fixed-Point Tool for the Simulink model specified by `modelname`. You can also access this tool by the following methods:

- From the Simulink **Analysis** menu, select **Fixed-Point Tool**.
- From a subsystem context (right-click) menu, select **Fixed-Point Tool**.

In conjunction with Fixed-Point Designer software, the Fixed-Point Tool provides convenient access to:

- Model and subsystem parameters that control the signal logging, fixed-point instrumentation mode, and data type override. (see “Model Parameters” on page 6-2)
- Plotting capabilities that enable you to plot data that resides in the MATLAB workspace, namely, simulation results associated with `Scope`, `To Workspace`, and root-level `Outport` blocks, in addition to logged signal data (see “Signal Logging” in the Simulink User's Guide)
- An interactive automatic data typing feature that proposes fixed-point data types for appropriately configured objects in your model, and then allows you to selectively accept and apply the data type proposals

You can launch the Fixed-Point Tool for any system or subsystem, and the tool controls the object selected in its **System under design** pane. If Fixed-Point Designer software is installed, the Fixed-Point Tool **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. From the Simulink model **Analysis** menu, select **Fixed Point Tool**.

   The Fixed-Point Tool opens.
2. Under **System under design**, select the system you want to convert.
3. Under **Configure model settings**, click **Advanced settings**. In the Advanced Settings dialog box:

   - 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`.

   Click **Apply** and close the dialog box.
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');
```

See Also

Fixed-Point Tool | “Propose Fraction Lengths Using Simulation Range Data”

Introduced before R2006a

gcb

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`

Introduced before R2006a

gcbh

Get handle of current block

Syntax

```
gcbh
```

Description

`gcbh` returns the handle of the current block in the current system.

You can use this command to identify or address blocks that have no parent system. The command should be most useful to blockset authors.

Examples

This command returns the handle of the most recently selected block.

```
gcbh

ans =

   281.0001
```

See Also

`gcb` | `getSimulinkBlockHandle`

Introduced before R2006a

gcs

Get pathname of current system

Syntax

```
gcs
```

Description

`gcs` returns the full pathname of the current system.

The current system is one of these:

- During editing, the current system is the system or subsystem most recently clicked.
- During simulation of a system that contains S-Function blocks, the current system is the system or subsystem containing the S-Function block that is currently being evaluated.
- During callbacks, the current system is the system containing any block whose callback routine is being executed.
- During evaluation of 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`

Introduced before R2006a

get_param

Get parameter names and values

Syntax

```
ParamValue = get_param(Object,Parameter)
```

Description

`ParamValue = get_param(Object,Parameter)` returns the name or value of the specified parameter for the specified model or block object. Open or load the Simulink model first.

Tip If you make multiple calls to `get_param` for the same block, then specifying the block using a numeric handle is more efficient than using the full block path. Use `getSimulinkBlockHandle` to get a block handle.

For parameter names, see:

- “Model Parameters” on page 6-2
- “Block-Specific Parameters” on page 6-101
- “Common Block Properties” on page 6-86

Examples

Get a Block Parameter Value and a Model Parameter Value

Load the `vdp` model.

```
load_system('vdp');
```

Get the value for the `Expression` block parameter.

```
BlockParameterValue = get_param('vdp/Fcn','Expression')
```

```
BlockParameterValue =
    1 - u*u
```

Get the value for the `SolverType` model parameter.

```
SolverType = get_param('vdp','SolverType')
```

```
SolverType =
   Variable-step
```

Get Root Parameter Names and Values

Get a list of global parameter names by finding the difference between the Simulink root parameter names and the model parameter names.

```
RootParameterNames = fieldnames(get_param(0,'ObjectParameters'));
load_system('vdp')
ModelParameterNames = fieldnames(get_param('vdp','ObjectParameters'));
GlobalParameterNames = setdiff(RootParameterNames,ModelParameterNames)
```

```
GlobalParameterNames =
          'AutoSaveOptions'
          'CacheFolder'
          'CallbackTracing'
          'CharacterEncoding'
                 . . .
          'CurrentSystem'
```

Get the value of a global parameter.

```
GlobalParameterValue = get_param(0,'CurrentSystem')
```

```
GlobalParameterValue =
          vdp
```

Get Model Parameter Names and Values

Get a list of model parameters for the `vdp` model .

```
load_system('vdp')
ModelParameterNames = get_param('vdp','ObjectParameters')
```

```
ModelParameterNames =
              Name: [1x1 struct]
```

```
              Tag: [1x1 struct]
      Description: [1x1 struct]
             Type: [1x1 struct]
           Parent: [1x1 struct]
           Handle: [1x1 struct]
                      . . .
          Version: [1x1 struct]
```

Get the current value of the `ModelVersion` model parameter for the `vdp` model.

```
ModelParameterValue = get_param('vdp','ModelVersion')
```

```
ModelParameterValue =
          1.6
```

Get All Block Parameter Names and Values

Get a list of block paths and names for the `vdp` model.

```
load_system('vdp')
BlockPaths = find_system('vdp','Type','Block')
```

```
BlockPaths =
    'vdp/Fcn'
    'vdp/More Info'
    'vdp/More Info/Model Info'
    'vdp/Mu'
    'vdp/Mux'
    'vdp/Product'
    'vdp/Scope'
    'vdp/Sum'
    'vdp/x1'
    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'
```

Get a list of block dialog parameters for the `Fcn` block.

```
BlockDialogParameters = get_param('vdp/Fcn','DialogParameters')
```

```
BlockDialogParameters =
               Expr: [1x1 struct]
         SampleTime: [1x1 struct]
```

Get the value for the `Expr` block parameter.

```
BlockParameterValue = get_param('vdp/Fcn','Expr')

BlockParameterValue =
    1 - u*u
```

Get a Block Parameter Value Using a Block Handle

If you make multiple calls to `get_param` for the same block, then using the block handle is more efficient than specifying the full block path as a string, e.g., `'vdp/Fcn'`.

You can use the block handle in subsequent calls to `get_param` or `set_param`. If you examine the handle, you can see that it contains a double. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Instead, assign the handle to a variable and use that variable name to specify a block.

Use `getSimulinkBlockHandle` to load the `vdp` model if necessary (by specifying `true`), and get a handle to the `FCN` block.

```
fcnblockhandle = getSimulinkBlockHandle('vdp/Fcn',true);
```

Use the block handle with `get_param` and get the value for the `Expr` block parameter.

```
BlockParameterValue = get_param(fcnblockhandle,'Expression')

BlockParameterValue =
    1 - u*u
```

Display Block Types for all Blocks in a Model

Get a list of block paths and names for the `vdp` model.

```
load_system('vdp')
BlockPaths = find_system('vdp','Type','Block')

BlockPaths =
    'vdp/Fcn'
    'vdp/More Info'
    'vdp/More Info/Model Info'
    'vdp/Mu'
    'vdp/Mux'
    'vdp/Product'
    'vdp/Scope'
    'vdp/Sum'
```

```
    'vdp/x1'
    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'
```

Get the value for the `BlockType` parameter for each of the blocks in the `vdp` model.

```
BlockTypes = get_param(BlockPaths,'BlockType')
```

```
BlockTypes =
    'Fcn'
    'SubSystem'
    'SubSystem'
    'Gain'
    'Mux'
    'Product'
    'Scope'
    'Sum'
    'Integrator'
    'Integrator'
    'Outport'
    'Outport'
```

- “Associating User Data with Blocks”
- “Use MATLAB Commands to Change Workspace Data”

Input Arguments

`Object` — Name or handle of a model or block, or root

handle | string | cell array of strings | 0

Handle or name of a model or block, or root, specified as a numeric handle or a string, a cell array of strings for multiple blocks, or `0` for root. A numeric handle must be a scalar. You can also get parameters of lines and ports, but you must use numeric handles to specify them.

Tip If you make multiple calls to `get_param` for the same block, then specifying a block using a numeric handle is more efficient than using the full block path. Use `getSimulinkBlockHandle` to get a block handle. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than

MATLAB displays. Assign the handle to a variable and use that variable name to specify a block.

Specify `0` to get root parameter names, including global parameters and model parameters for the current Simulink session.

- Global parameters include Editor preferences and Simulink Coder parameters.
- Model parameters include configuration parameters, Simulink Coder parameters, and Simulink Code Inspector™ parameters.

Example: `'vdp/Fcn'`

`Parameter` — Parameter of model or block, or root

string

Parameter of model or block, or root, specified as a string or `0` for root. The table shows special cases.

Specified Parameter	Result
`'ObjectParameters'`	Returns a structure array with the parameter names of the specified object (model, block, or root) as separate fields in the structure.
`'DialogParameters'`	Returns a structure array with the block dialog box parameter names as separate fields in the structure. If the block has a mask, the function instead returns the mask parameters.
Parameter name, e.g., `'BlockType'`. Specify any model or block parameter, or block dialog box parameter.	Returns the value of the specified model or block parameter. If you specified multiple blocks as a cell array, returns a cell array with the values of the specified parameter common to all blocks. All of the specified blocks in the cell array must contain the parameter, otherwise the function returns an error.

Example: `'ObjectParameters'`

Data Types: `char`

Output Arguments

`ParamValue` — The name or value of the specified parameter for the specified model or block, or root

any data type, depending on the parameter

The name or value of the specified parameter for the specified model or block, or root. If you specify multiple objects, the output is a cell array of objects. The table shows special cases.

Specified Parameter	ParamValue Returned
`'ObjectParameters'`	A structure array with the parameter names of the specified object (model, block, or root) as separate fields in the structure.
`'DialogParameters'`	A structure array with the block dialog box parameter names as separate fields in the structure. If the block has a mask, the function instead returns the mask parameters.
Parameter name, e.g., `'BlockType'`	The value of the specified model or block parameter. If multiple blocks are specified as a cell array, returns a cell array with the values of the specified parameter common to all blocks.

If you get the root parameters by specifying `get_param(0,'ObjectParameters')`, then the output `ParamValue` is a structure array with the root parameter names as separate fields in the structure. Each parameter field is a structure containing these fields:

- Type — Parameter type values are: `'boolean'`, `'string'`, `'int'`, `'real'`, `'point'`, `'rectangle'`, `'matrix'`, `'enum'`, `'ports'`, or `'list'`
- Enum — Cell array of enumeration string values that applies only to `'enum'` parameter types

- Attributes — Cell array of strings defining the attributes of the parameter. Values are: `'read-write'`, `'read-only'`, `'read-only-if-compiled'`, `'write-only'`, `'dont-eval'`, `'always-save'`, `'never-save'`, `'nondirty'`, or `'simulation'`

More About

- "Model Parameters" on page 6-2
- "Block-Specific Parameters" on page 6-101
- "Common Block Properties" on page 6-86

See Also

`bdroot` | `find_system` | `gcb` | `gcs` | `getSimulinkBlockHandle` | `set_param`

Introduced before R2006a

getActiveConfigSet

Get model's active configuration set or configuration reference

Syntax

```
myConfigObj = getActiveConfigSet(model)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

Description

`getActiveConfigSet` returns the configuration set or configuration reference (configuration object) that is the active configuration object of `model`.

Examples

The following example returns the active configuration object of the current model. The code is the same whether the object is a configuration set or configuration reference.

```
myConfigObj = getActiveConfigSet(gcs);
```

More About

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Introduced before R2006a

getCallbackAnnotation

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., `[]`.

Introduced before R2006a

getConfigSet

Get one of model's configuration sets or configuration references

Syntax

```
myConfigObj = getConfigSet(model, configObjName)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObjName`

The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`getConfigSet` returns the configuration set or configuration reference (configuration object) that is attached to `model` and is named `configObjName`. If no such object exists, an error occurs.

Examples

The following example returns the configuration object that is named `DevConfig` and attached to the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
myConfigObj = getConfigSet(gcs, 'DevConfig');
```

More About

- "Manage a Configuration Set"

- “Manage a Configuration Reference”

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Introduced before R2006a

getConfigSets

Get names of all of model's configuration sets or configuration references

Syntax

```
myConfigObjNames = getConfigSets(model)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

Description

`getConfigSets` returns a cell array of 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)
```

More About

- "Manage a Configuration Set"
- "Manage a Configuration Reference"

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `openDialog` | `setActiveConfigSet`

Introduced before R2006a

getfullname

Get pathname of block or line

Syntax

```
path=getfullname(handle)
```

Description

`path=getfullname(handle)` returns the full pathname of the block or line specified by `handle`.

Examples

`getfullname(gcb)` returns the pathname of the block currently selected in the model editor's window.

The following code returns the pathname of the line currently selected in the model editor's window.

```
line = find_system(gcs, 'SearchDepth', 1, 'FindAll', 'on', ...
      'Type', 'line', 'Selected', 'on');
path = getfullname(line);
```

See Also

`gcb` | `find_system`

Introduced in R2007a

getInputString

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`.

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.

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.

More About

- "Map Root Inport Signal Data"

See Also

`getRootInportMap`

Introduced in R2013a

getRootInportMap

Create custom object to map signals to root-level inports

Syntax

```
map = getRootInportMap('Empty');
map = getRootInportMap(model,mdl,Name,Value);
map = getRootInportMap(inputmap,map,Name,Value);
```

Description

`map = getRootInportMap('Empty');` creates an empty map object, *map*. Use this map object to set up an empty custom mapping object. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards.

`map = getRootInportMap(model,mdl,Name,Value);` creates a map object for `model`, `mdl`, with block names and signal names specified. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards.

`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 the example file `BlockNameIgnorePrefixMap.m`, located in *`matlabroot`*`/help/toolbox/simulink/examples`, for an example of a custom mapping algorithm.

Input Arguments

`Empty`

Create an empty map object.

Default: none

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'model'`

Name of model to associate with the root inport map.

Default: None

`'blockName'`

Block names of root-level input ports. The tool assigns data to ports according to the name of the root-inport block. If the tool finds a data element whose name matches the name of a root-inport block, it maps the data to the corresponding port.

The value for this argument can be:
Block name of root-level input ports.
Cell array containing multiple block names of root-level input ports.

Default: None

`'signalName'`

Signal names to be mapped. The tool assigns data to ports according to the name of the signal on the port. If the tool finds a data element whose name matches the name of a signal at a port, it maps the data to the corresponding port.

The value for this argument can be:
Signal name to be mapped.
Cell array containing multiple signal names of signals to be mapped.

Default: None

`'inputmap'`

Name of mapping object to override.

Default: None

Output Arguments

`map`

Custom object that you can use to map data to root-level input port.

Examples

Empty Mapping Object

Create an empty custom mapping object.

```
map = getRootInportMap('Empty')
```

```
map =

  1x0 InputMap array with properties:

    Type
    DataSourceName
    Destination
```

Simple Mapping Object

Create a simple mapping object using a MATLAB time series object.

Create a time series object, `signalIn1`.

```
signalIn1 = timeseries((1:10)')
```

```
Common Properties:
            Name: 'unnamed'
            Time: [10x1 double]
        TimeInfo: [1x1 tsdata.timemetadata]
            Data: [10x1 double]
        DataInfo: [1x1 tsdata.datametadata]
```

Create a mapping object for the time series object for the model, `ex_minportsOnlyModel`.

```
modelFile = fullfile(matlabroot,'help','toolbox','simulink',...
'examples','ex_minportsOnlyModel');
load_system(modelFile);
map = getRootInportMap('model','ex_minportsOnlyModel',...
'blockName','In1','signalname','signalIn1')

map =

  InputMap with properties:

             Type: 'Inport'
    DataSourceName: 'signalIn1'
       Destination: [1x1 Simulink.iospecification.Destination]
```

Mapping Object with Vectors

Create a mapping object using vectors of block names and signal names for the model `ex_minportsOnlyModel`.

Create a mapping object of vectors.

```
modelFile = fullfile(matlabroot,'help','toolbox','simulink',...
'examples','ex_minportsOnlyModel');
load_system(modelFile);
map = getRootInportMap('model','ex_minportsOnlyModel',...
'blockName',{'In1' 'In2'}, ...
'signalname',{'signalIn1' 'signalIn2'})

map =

  1x2 InputMap array with properties:

    Type
    DataSourceName
    Destination
```

Overriding Maps

Create a mapping object that contains the signal *var2*, then override *var2* with *var1*.

Create a mapping object of vectors.

```
% Load the model and define variables
modelFile = fullfile(matlabroot,'help','toolbox','simulink',...
'examples','ex_minportsOnlyModel');
```

```
load_system(modelFile);
modelValue = 'ex_minportsOnlyModel';
blockNameValue = 'In1';
signalNameValue = 'var2';
portType = 'Inport';

% Define var1 and override var2 with var1
signalNameToOverload = 'var1';
mapToOverload = getRootInportMap('model',modelValue,...
'blockName',blockNameValue,...
 'signalName',signalNameToOverload)

mapToOverload =

  InputMap with properties:

              Type: 'Inport'
    DataSourceName: 'var1'
       Destination: [1x1 Simulink.iospecification.Destination]
```

- "Create Custom Mapping File Function"

More About

Tips

- Load the model before running this function.
- If your custom mapping mode similar to an existing Simulink mapping mode, consider using the `getSlRootInportMap` function instead.

See Also

`getInputString` | `getSlRootInportMap`

Introduced in R2012b

getSimulinkBlockHandle

Get block handle from block path

Syntax

```
handle = getSimulinkBlockHandle(path)
handle = getSimulinkBlockHandle(path,'true')
```

Description

`handle = getSimulinkBlockHandle(path)` returns the numeric handle of the block specified by `path`, if it exists in a loaded model or library. Returns `-1` if the block is not found. Library links are resolved where necessary.

Use the numeric handle returned by `getSimulinkBlockHandle` to manipulate the block in subsequent calls to `get_param` or `set_param`. This approach is more efficient than making multiple calls to these functions using the full block path. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Assign the handle to a variable and use that variable name to specify a block. The handle applies only to the current MATLAB session.

Use `getSimulinkBlockHandle` to check whether a block path is valid. This approach is more efficient than calling `get_param` inside a `try` statement.

`handle = getSimulinkBlockHandle(path,'true')` attempts to load the model or library containing the specified block `path`, and then checks if the block exists. No error is returned if the model or library is not found. Any models or libraries loaded this way remain in memory even if the function does not find a block with the specified path.

Examples

Get the Handle of a Block

Get the handle of the `Pilot` block.

```
load_system('f14')
handle = getSimulinkBlockHandle('f14/Pilot')
```

```
handle =

  562.0004
```

You can use the handle in subsequent calls to `get_param` or `set_param`.

Load the Model and Get the Block Handle

Load the model `f14` if necessary (by specifying `true`), and get the handle of the `Pilot` block.

```
handle = getSimulinkBlockHandle('f14/Pilot',true)
```

```
handle =

  562.0004
```

You can use the handle in subsequent calls to `get_param` or `set_param`.

Check If a Model Contains a Specific Block

Check whether the model `f14` is loaded and contains a block named `Pilot`. Valid handles are always greater than zero. If the function does not find the block, it returns `-1`.

```
valid_block_path = getSimulinkBlockHandle('f14/Pilot') > 0
```

```
valid_block_path =

     0
```

The model contains the block but the model is not loaded, so this returns `0` because it cannot find the block.

Using `getSimulinkBlockHandle` to check whether a block path is valid is more efficient than calling `get_param` inside a `try` statement.

Input Arguments

`path` — Block path name

string | cell array of strings

Block path name, specified as a string or a cell array of strings.

Example: `'f14/Pilot'`

Data Types: `char`

Output Arguments

`handle` — Numeric handle of a block

double | array of doubles

Numeric handle of a block, returned as a double or an array of doubles. Valid handles are always greater than zero. If the function does not find the block, it returns `-1`. If the `path` input is a cell array of strings, then the output is a numeric array of handles.

Data Types: `double`

See Also

`get_param` | `set_param`

Introduced in R2015a

getSlRootInportMap

Create custom object to map signals to root-level inports using Simulink mapping mode

Syntax

```
inputMap = getSlRootInportMap('model',modelname,'MappingMode',
mappingmode,'SignalName',signalname,'SignalValue',signalvalue)
[inputMap, hasASignal] = getSlRootInportMap('model',
modelname,'MappingMode',mappingmode,'SignalName',
signalname,'SignalValue',signalvalue)
```

```
inputMap = getSlRootInportMap('model',
modelname,'MappingMode','Custom','CustomFunction',
customfunction,'SignalName',signalname,'SignalValue',signalvalue)
[inputMap,hasASignal] = getSlRootInportMap('model',
modelname,'MappingMode','Custom','CustomFunction',
customfunction,'SignalName',signalname,'SignalValue',signalvalue)
```

Description

`inputMap = getSlRootInportMap('model',modelname,'MappingMode', mappingmode,'SignalName',signalname,'SignalValue',signalvalue)` creates a root inport map using one of the Simulink mapping modes. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards.

`[inputMap, hasASignal] = getSlRootInportMap('model', modelname,'MappingMode',mappingmode,'SignalName', signalname,'SignalValue',signalvalue)` returns a vector of logical values specifying whether or not the root inport map has a signal associated with it.

`inputMap = getSlRootInportMap('model', modelname,'MappingMode','Custom','CustomFunction', customfunction,'SignalName',signalname,'SignalValue',signalvalue)` creates a root inport map using a custom mapping mode specified in `customfunction`. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards.

`[inputMap,hasASignal] = getSlRootInportMap('model', modelname,'MappingMode','Custom','CustomFunction', customfunction,'SignalName',signalname,'SignalValue',signalvalue)` returns a vector of logical values specifying whether or not the root inport map has a signal associated with it.

To map signals to root-level inports using custom mapping modes, you can use `getSlRootInport` with the Root Inport Mapping dialog box custom mapping capability.

Examples

Create inport map using Simulink mapping mode

Create a vector of inport maps using a built-in mapping mode.

```
Throttle = timeseries(ones(10,1)*10);
           Brake    = timeseries(zeros(10,1));
           inputMap = getSlRootInportMap('model','slexAutotransRootInportsExample',...
               'MappingMode','BlockName', ...
               'SignalName',{'Throttle' 'Brake'},...
               'SignalValue',{Throttle Brake});
```

Create inport map using custom function

Create a vector of inport maps using a custom function

```
port1    = timeseries(ones(10,1)*10);
           port2    = timeseries(zeros(10,1));
           inputMap = getSlRootInportMap('model','slexAutotransRootInportsExample',...
               'MappingMode','Custom', ...
               'CustomFunction','slexCustomMappingMyCustomMap',...
               'SignalName',{'port1' 'port2'},...
               'SignalValue',{port1 port2});
```

Input Arguments

`modelname` — Model name

string

Specify the model to associate with the root inport map.

Data Types: `char`

`mappingmode` — Simulink mapping mode

string

Specify the mapping mode to use with model name and data source. Possible string values are:

`'Index'`	Assign sequential index numbers, starting at 1, to the data in the MAT-file, and map this data to the corresponding inport.
`'BlockName'`	Assign data to ports according to the name of the root-inport block. If the block name of a data element matches the name of a root-inport block, map the data to the corresponding port.
`'SignalName'`	Assign data to ports according to the name of the signal on the port. If the signal name of a data element matches the name of a signal at a port, map the data to the corresponding port.
`'BlockPath'`	Assign data to ports according to the block path of the root-inport block. If the block path of a data element matches the block path of a root-inport block, map the data to the corresponding port.
`'Custom'`	Apply mappings according to the definitions in a custom file.

Data Types: `char`

`customfunction` — Custom function file name

string

Specify name of file that implements a custom method to map signals to root-level ports. This function must be on the MATLAB path.

Data Types: `char`

`signalname` — signal name

scalar | cell array of strings

Specify the signal name(s) of the signal to associate with the root inport map.

Data Types: `char` | `cell`

`signalvalue` — signal value

scalar | cell arrays

Specify the values of the signals to map to the root inport map. For the list of supported data types for the values, see “Choose a Base Workspace and MAT-File Format”.

Output Arguments

`inputMap` — input map

scalar | vector

Mapping object that defines the mapping of input signals to root-level ports.

`hasASignal` — signal presence indicator

scalar | vector

A vector of logical values with the same length as `inputMap`. If the value is true the inputMap has a signal associated with it. If the value is false the inputMap does not have a signal associated with it and will use a ground value as an input

Data Types: `logical`

More About

Tips

- Load the model before running this function.
- If your custom mapping mode is not similar to an existing Simulink mapping mode, consider using the `getRootInportMap` function instead.
- "Map Root Inport Signal Data"

See Also

`getRootInportMap`

Introduced in R2013b

getVariable

Get value of variable from workspace

Syntax

```
variableValue = getVariable(workspaceHandle,variableName)
variableValue = workspaceHandle.getVariable(variableName)
```

Description

`variableValue = getVariable(workspaceHandle,variableName)` returns the value of the variable. If the variable does not exist in the workspace, an error occurs.

`variableValue = workspaceHandle.getVariable(variableName)` is an alternative syntax.

Input Arguments

`workspaceHandle`

Handle to the workspace containing the variable.

`variableName`

Name of the variable containing the value.

Output Arguments

`variableValue`

Value of the variable.

Examples

Get Value of Workspace Variable

Get the value of the workspace variable K, which is defined in model `mdl`.

```
wksp = get_param(mdl,'ModelWorkspace')
value = getVariable(wksp,'K')
```

```
value =
   5
```

See Also

`get_param`

Introduced in R2012a

hasVariable

Determine if variable exists in workspace

Syntax

```
variableExists = hasVariable(workspaceHandle,variableName)
variableExists = workspaceHandle.hasVariable(variableName)
```

Description

`variableExists = hasVariable(workspaceHandle,variableName)` returns `1` if the variable exists in the workspace, and `0` if not.

`variableExists = workspaceHandle.hasVariable(variableName)` is an alternative syntax.

Input Arguments

`workspaceHandle`

Handle to the workspace.

`variableName`

Name of the variable.

Output Arguments

`variableExists`

Boolean value that indicates whether the variable exists in the workspace (`1` if true and `0` if false).

Examples

Determine Existence of Variable

Determine if the variable K exists in the workspace for model `mdl`.

```
wksp = get_param(mdl,'ModelWorkspace')
exists = hasVariable(wksp,'K')
```

```
exists =
   1
```

See Also

`get_param`

Introduced in R2012a

hdllib

Display blocks that are compatible with HDL code generation

Syntax

```
hdllib
hdllib('off')
hdllib('html')
hdllib('librarymodel')
```

Description

`hdllib` displays the blocks that are supported for HDL code generation, and for which you have a license, in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this Library Browser view.

If you close and reopen the Library Browser in the same MATLAB session, the Library Browser continues to show only the blocks supported for HDL code generation. To show all blocks, regardless of HDL code generation compatibility, at the command prompt, enter `hdllib('off')`.

`hdllib('off')` displays all the blocks for which you have a license in the Library Browser, regardless of HDL code generation compatibility.

`hdllib('html')` creates a library of blocks that are compatible with HDL code generation. It generates two additional HTML reports: a categorized list of blocks (`hdlblklist.html`) and a table of blocks and their HDL code generation parameters (`hdlsupported.html`).

To run `hdllib('html')`, you must have an HDL Coder license.

`hdllib('librarymodel')` displays blocks that are compatible with HDL code generation in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this library.

The default library name is `hdlsupported`. After you generate the library, you can save it to a folder of your choice.

To keep the library current, you must regenerate it each time that you install a new software release.

To run `hdllib('librarymodel')`, you must have an HDL Coder license.

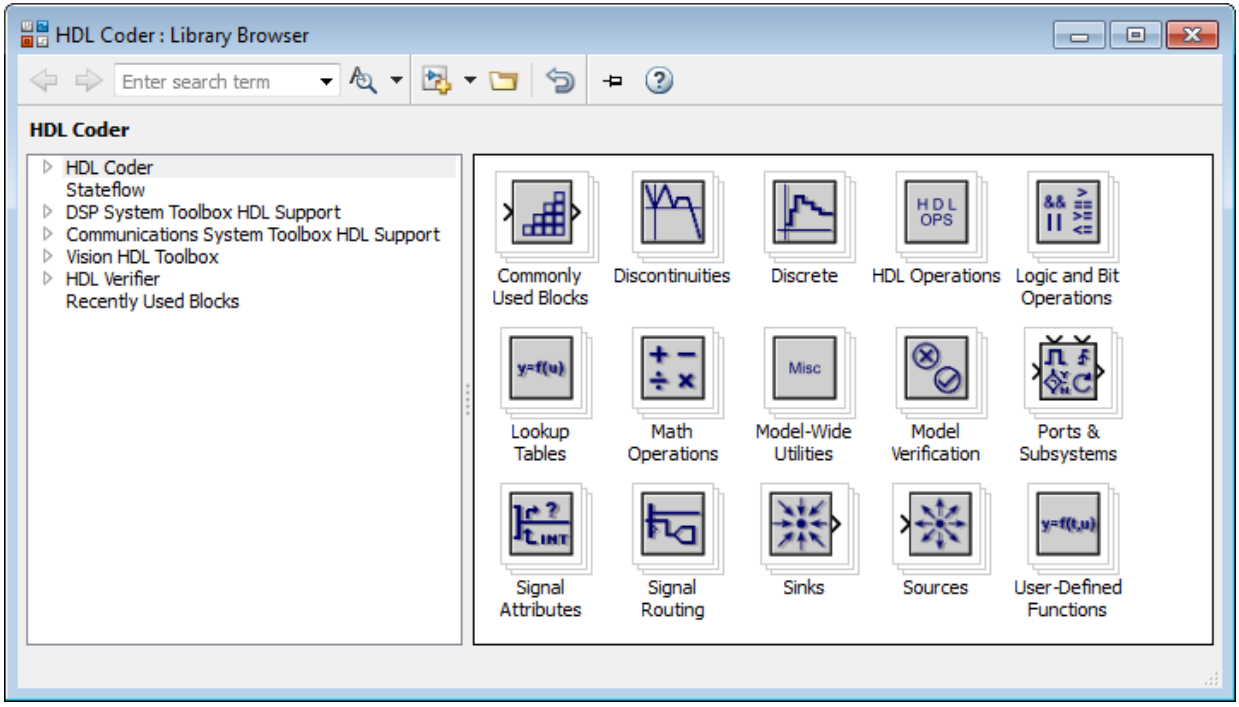
Examples

Display Supported Blocks in the Library Browser

To display blocks that are compatible with HDL code generation in the Library Browser:

```
hdllib
```

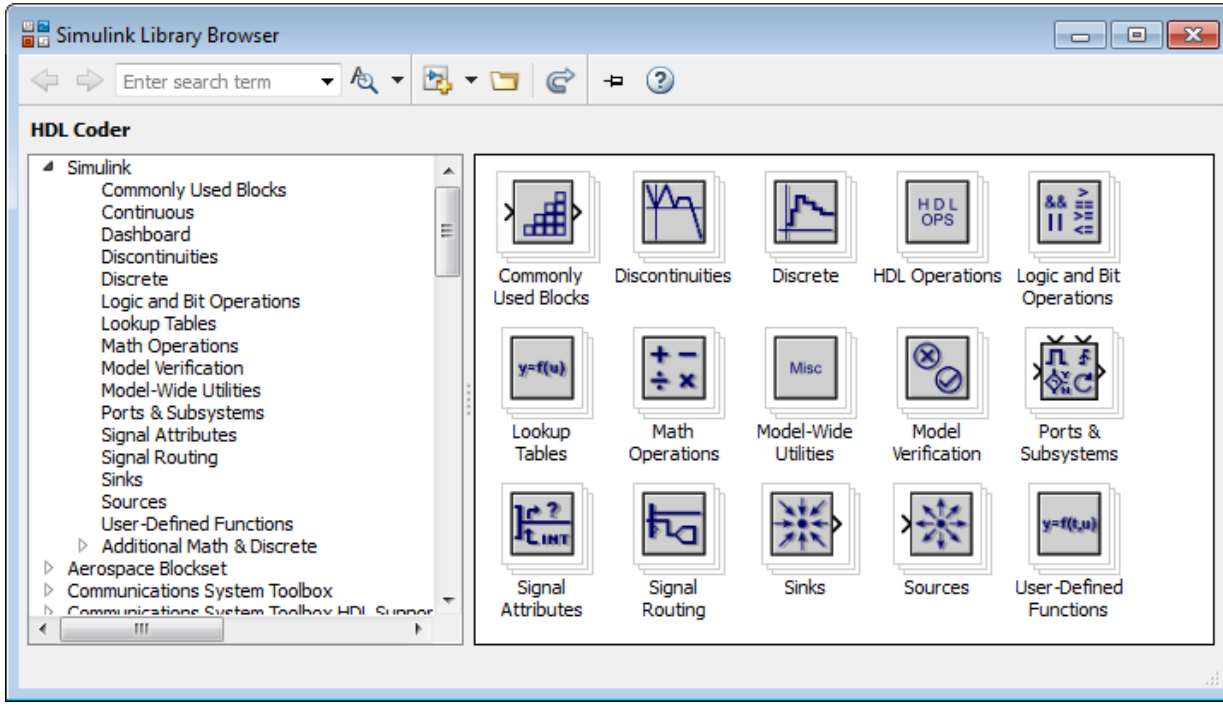
```
### Generating view of HDL Coder compatible blocks in Library Browser.
### To restore the Library Browser to the default Simulink view, enter "hdllib off".
```



Display All Blocks in the Library Browser

To display all blocks in the Library Browser, regardless of HDL code generation compatibility:

```
hdllib('off')

### Restoring Library Browser to default view; removing the HDL Coder compatibility fil
```

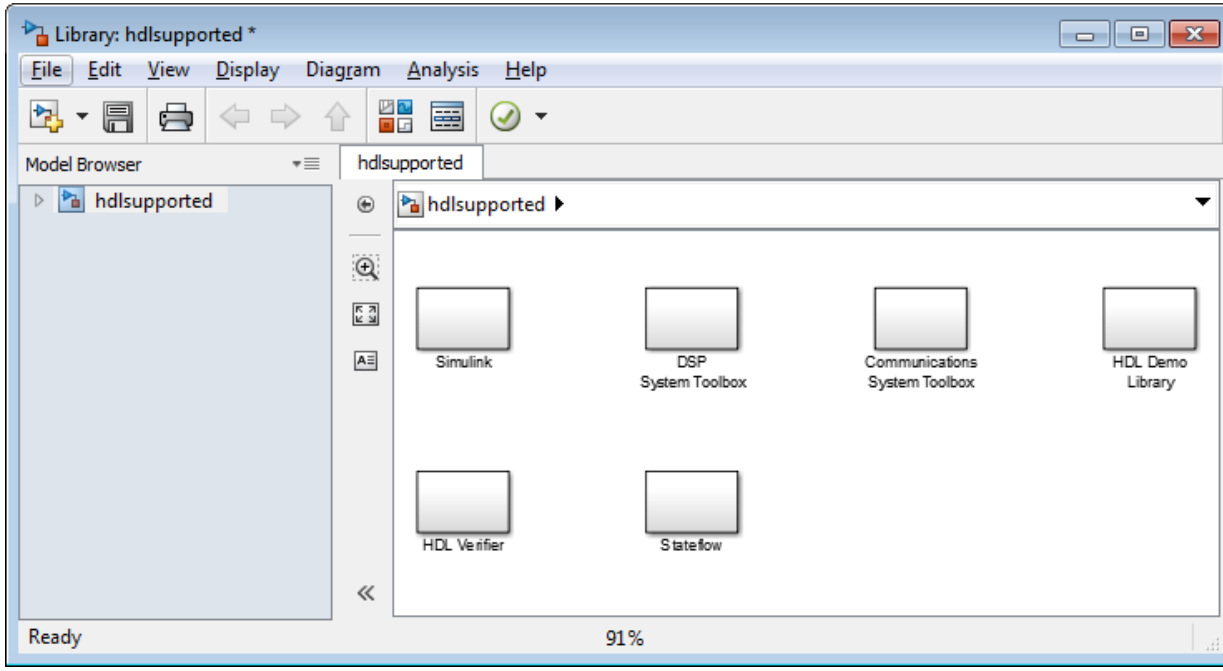


Create a Supported Blocks Library and HTML Reports

To create a library and HTML reports showing the blocks that are compatible with HDL code generation:

```
hdllib('html')

### HDL supported block list hdlblklist.html
### HDL implementation list hdlsupported.html
```

The `hdlsupported` library opens. To view the reports, click the `hdlblklist.html` and `hdlsupported.html` links.

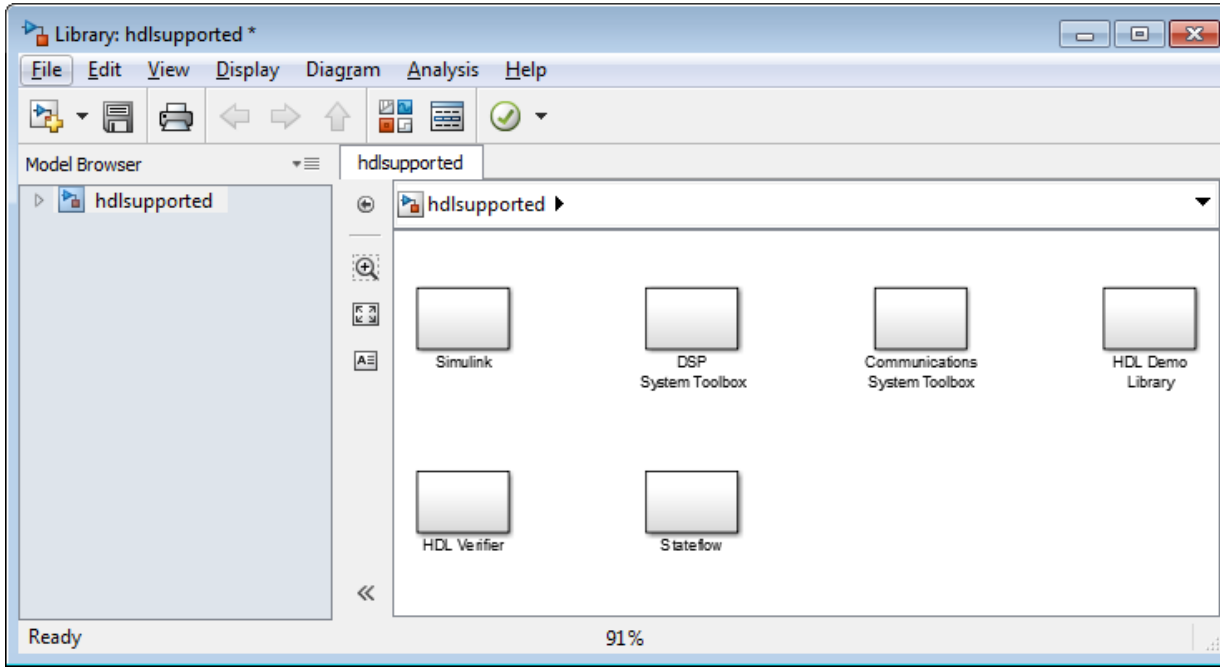


Create a Supported Blocks Library

To create a library that contains blocks that are compatible with HDL code generation:

```
hdllib('librarymodel')
```

The `hdlsupported` block library opens.



- “Show Blocks Supported for HDL Code Generation”
- “View HDL-Specific Block Documentation”
- “Prepare Simulink Model For HDL Code Generation”

See Also

“Supported Blocks”

Introduced in R2006b

hilite_system

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.

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 2-281.

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.

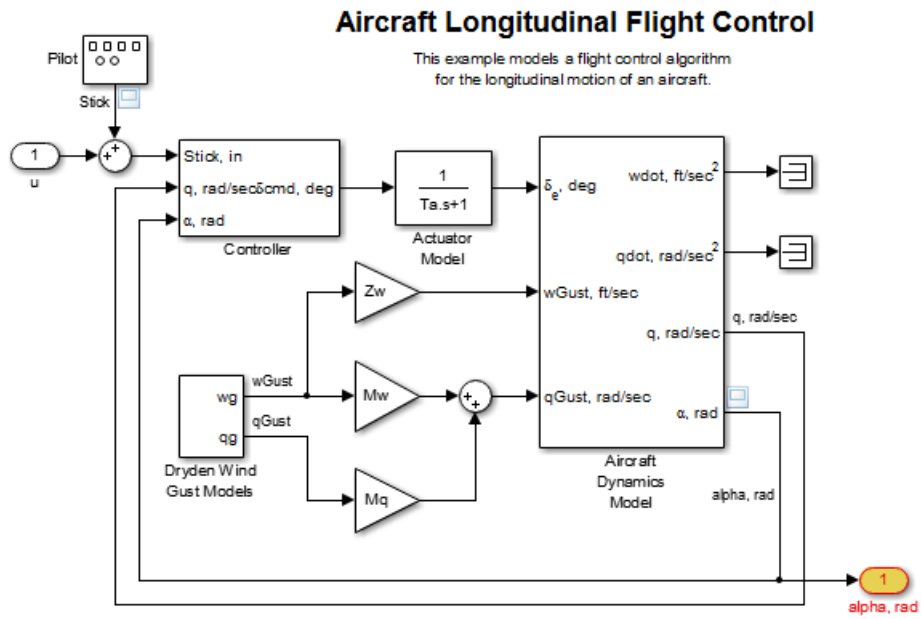
1. Open the Model Configuration Parameters dialog box. In the **Solver** pane, set solver parameters as follows:

   - 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:

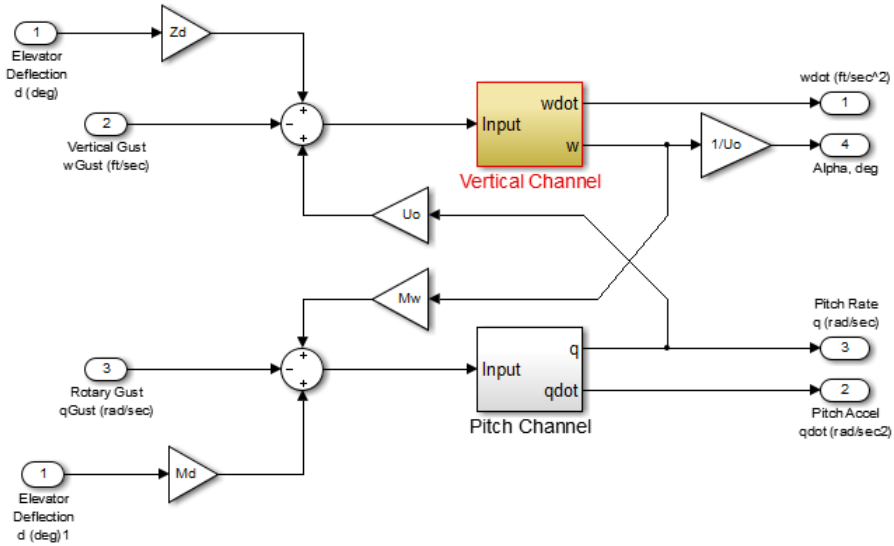
   ```
   hilite_system('<Root>/alpha, rad')
   ```

   The following figure shows block `alpha, rad` highlighted.



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')
```



More About

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 2-282.
- `'BackgroundColor'`: string specifying a background "Color" on page 2-282.

Color

The supported color strings for foreground and background colors are:

```
black
white
gray
red
orange
yellow
green
darkGreen
blue
lightBlue
cyan
magenta
```

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 ÿ (`char(255)`).

See Also

rtwtrace

Introduced in R2010a

isLoaded

Determine if Simulink Project is loaded

Syntax

```
loaded = isLoaded(proj)
```

Description

`loaded = isLoaded(proj)` returns whether the project referenced by the project object `proj` is loaded.

Examples

Find Out if Project Is Loaded

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Find out if the project is still loaded.

```
loaded = isLoaded(proj)
```

```
loaded =

     1
```

Input Arguments

`proj` — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

Output Arguments

`loaded` — Loaded status

1 | 0

Project loaded status, returned as true (1) if the project is loaded.

Data Types: `logical`

See Also

Functions

`reload` | `simulinkproject`

Introduced in R2013a

legacy_code

Use Legacy Code Tool

Syntax

```
legacy_code('help')
specs = legacy_code('initialize')
legacy_code('sfcn_cmex_generate', specs)
legacy_code('compile', specs, compilerOptions)
legacy_code('generate_for_sim', specs, modelname)
legacy_code('slblock_generate', specs, modelname)
legacy_code('sfcn_tlc_generate', specs)
legacy_code('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 “Build S-Function With 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 “Build S-Function With 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`).
- `convert2DMatrixToRowMajor`— A logical value specifying the automatic conversion of a matrix between a 2–D column-major format and a row-major

format. The 2-D column-major format is used by MATLAB, Simulink, and the generated code. The row-major format is used by C. By default, the value is `false` (0).

Note: This option does not support a 2–D matrix of complex data.

- `supportCoverage`— A logical value specifying whether the generated S-function must be compatible with Model Coverage. By default, the value is `false` (0).
- `supportCoverageAndDesignVerifier`— A logical value specifying whether the generated S-function must be compatible with Model Coverage and Simulink Design Verifier™. By default, the value is `false` (0).
- `outputsConditionallyWritten`— A logical value specifying whether the legacy code conditionally writes the output ports. If `true`, the generated S-function specifies that the memory associated with each output port cannot be overwritten and is global (`SS_NOT_REUSABLE_AND_GLOBAL`). If `false`, the memory associated with each output port is reusable and is local (`SS_REUSABLE_AND_LOCAL`). By default, the value is `false` (0). For more information, see `ssSetOutputPortOptimOpts`.

`modelname`

The name of a Simulink model into which Legacy Code Tool is to insert the masked S-function block generated when you specify `legacy_code` with the action string `'slblock_generate'`. If you omit this argument, the block appears in an empty model editor window.

More About

- "Integrate C Functions Using Legacy Code Tool"
- "Build S-Function With Legacy Code Tool"

Introduced in R2006b

libinfo

Get information about library blocks referenced by model

Syntax

```
libdata = libinfo('system')
libdata = libinfo('system', constraint1, value1, ...)
```

Description

`libdata = libinfo('system')` returns information about library blocks referenced by *`system`* and all the systems underneath it.

`libdata = libinfo('system', constraint1, value1, ...)` restricts the search as indicated by the search constraint(s) *`c1`*, *`v1`*, ...

Input Arguments

`system`

The system to search recursively for library blocks.

`constraint1, value1, ...`

One or more pairs, each consisting of a search constraint followed by a constraint value. You can specify any of the search constraints that you can use with `find_system`.

Output Arguments

`libdata`

An array of structures that describes each library block referenced by *`system`*. Each structure has the following fields:

`Block`	Path of the link to the library block

`Library`	Name of the library containing the referenced block
`ReferenceBlock`	Path of the library block
`LinkStatus`	Value of the `LinkStatus` parameter for the link to the library block

More About

- "Custom Libraries and Linked Blocks"

See Also

`find_system`

Introduced before R2006a

linmod

Extract continuous-time linear state-space model around operating point

Syntax

```
argout = linmod('sys');
argout = linmod('sys',x,u);
argout = linmod('sys', x, u, para);
argout = linmod('sys', x, u, 'v5');
argout = linmod('sys', x, u, para, 'v5');
argout = linmod('sys', x, u, para, xpert, upert, 'v5');
```

Arguments

sys	Name of the Simulink system from which the linear model is extracted.
x and u	State (x) and the input (u) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify x. To extract the x structure from the model, use the following command: `x = Simulink.BlockDiagram.getInitialState('sys');` You can then change the operating point values within this structure by editing `x.signals.values`. If the state contains different data types (for example, `'double'` and `'uint8'`), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is `'double'`.
Ts	Sample time of the discrete-time linearized model
'v5'	An optional argument that invokes the perturbation algorithm created prior to MATLAB 5.3. Invoking this optional argument is equivalent to calling `linmodv5`.
para	A three-element vector of optional arguments:

- `para(1)` — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the `'v5'` flag. The default value is 1e-05.
- `para(2)` — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (`t`) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.
- `para(3)` — Set `para(3)=1` to remove extra states associated with blocks that have no path from input to output. The default value is 0.

`xpert` and `upert` — The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

```
xpert = para(1) + 1e-3*para(1)*abs(x)
upert = para(1) + 1e-3*para(1)*abs(u)
```

When a model has model references using the `Model` block, you must use the Simulink structure format to specify `xpert`. To extract the `xpert` structure, use the following command:

```
xpert = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the perturbation values within this structure by editing `xpert.signals.values`.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling `linmodv5` or specifying the `'v5'` input argument to `linmod`.

argout

`linmod`, `dlinmod`, and `linmod2` return state-space representations if you specify the output (left-hand) side of the equation as follows:

- `[A,B,C,D] = linmod('sys', x, u)` obtains the linearized model of `sys` around an operating point with the specified state variables `x` and the input `u`. If you omit `x` and `u`, the default values are zero.

`linmod` and `dlinmod` both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using `linmod` as an example:

- `[num, den] = linmod('sys', x, u)` returns the linearized model in transfer function form.
- `sys_struc = linmod('sys', x, u)` returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

`linmod` compute a linear state-space model by linearizing each block in a model individually.

`linmod` obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

The default algorithm uses preprogrammed analytic block Jacobians for most blocks which should result in more accurate linearization than numerical perturbation of block inputs and states. A list of blocks that have preprogrammed analytic Jacobians is available in the Simulink Control Design documentation along with a discussion of the block-by-block analytic algorithm for linearization.

The default algorithm also allows for special treatment of problematic blocks such as the `Transport Delay` and the `Quantizer`. See the mask dialog of these blocks for more information and options.

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a 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

`dlinmod` | `linmod2` | `linmodv5`

Introduced in R2007a

linmod2

Extract continuous-time linear state-space model around operating point

Syntax

```
argout = linmod2('sys', x, u);
argout = linmod2('sys', x, u, para);
```

Arguments

sys	Name of the Simulink system from which the linear model is extracted.
x, *u*	State (*x*) and the input (*u*) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify `x`. To extract the `x` structure from the model, use the following command: `x = Simulink.BlockDiagram.getInitialState('sys');` You can then change the operating point values within this structure by editing `x.signals.values`. If the state contains different data types (for example, `'double'` and `'uint8'`), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is `'double'`.
para	A three-element vector of optional arguments: • `para(1)` — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the `'v5'` flag. The default value is 1e-05. • `para(2)` — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (`t`) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.

- `para(3)` — Set `para(3)=1` to remove extra states associated with blocks that have no path from input to output. The default value is 0.

argout

`linmod`, `dlinmod`, and `linmod2` return state-space representations if you specify the output (left-hand) side of the equation as follows:

- `[A,B,C,D] = linmod('sys', x, u)` obtains the linearized model of *sys* around an operating point with the specified state variables *x* and the input *u*. If you omit *x* and *u*, the default values are zero.

`linmod` and `dlinmod` both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using `linmod` as an example:

- `[num, den] = linmod('sys', x, u)` returns the linearized model in transfer function form.
- `sys_struc = linmod('sys', x, u)` returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

`linmod2` computes a linear state-space model by perturbing the model inputs and model states, and uses an advanced algorithm to reduce truncation error.

`linmod2` obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a 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 more information, see “Linearizing Models”.

See Also

`linmod` | `dlinmod` | `linmodv5`

Introduced in R2007a

linmodv5

Extract continuous-time linear state-space model around operating point

Syntax

```
argout = linmodv5('sys');
argout = linmodv5('sys',x,u);
argout = linmodv5('sys', x, u, para);
argout = linmodv5('sys', x, u, para, xpert, upert);
```

Arguments

sys — Name of the Simulink system from which the linear model is extracted.

x, *u* — State (*x*) and the input (*u*) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify `x`. To extract the `x` structure from the model, use the following command:

```
x = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the operating point values within this structure by editing `x.signals.values`.

If the state contains different data types (for example, `'double'` and `'uint8'`), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is `'double'`.

para — A three-element vector of optional arguments:

- `para(1)` — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the `'v5'` flag. The default value is 1e-05.
- `para(2)` — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that

gives the time (`t`) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.

- `para(3)` — Set `para(3)=1` to remove extra states associated with blocks that have no path from input to output. The default value is 0.

`xpert`, *`upert`*

The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

```
xpert = para(1) + 1e-3*para(1)*abs(x)
upert = para(1) + 1e-3*para(1)*abs(u)
```

When a model has model references using the `Model` block, you must use the Simulink structure format to specify `xpert`. To extract the `xpert` structure, use the following command:

```
xpert = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the perturbation values within this structure by editing `xpert.signals.values`.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling `linmodv5` or specifying the `'v5'` input argument to `linmod`.

argout

`linmod`, `dlinmod`, and `linmod2` return state-space representations if you specify the output (left-hand) side of the equation as follows:

- `[A,B,C,D] = linmod('sys', x, u)` obtains the linearized model of `sys` around an operating point with the specified state variables `x` and the input `u`. If you omit `x` and `u`, the default values are zero.

`linmod` and `dlinmod` both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using `linmod` as an example:

- `[num, den] = linmod('sys', x, u)` returns the linearized model in transfer function form.
- `sys_struc = linmod('sys', x, u)` returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

`linmodv5` computes a linear state space model using the full model perturbation algorithm created prior to MATLAB 5.3.

`linmodv5` obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a 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`

Introduced in R2011b

load_system

Invisibly load Simulink model

Syntax

```
load_system('sys')
```

Description

`load_system('sys')` loads `sys`, where *`sys`* is the name of a Simulink model, into memory without making its model window visible.

You cannot use `load_system` to load MATLAB file models last saved in Simulink Version 1.3 (for example: `load_system mymodel.m`). If you have a MATLAB file model, you must upgrade to Simulink model file format as follows:

1. Execute the model as a function:

   ```
   mymodel
   ```

2. Save the model as a Simulink model file:

   ```
   save_system mymodel
   ```

Examples

The command

```
load_system('vdp')
```

loads the `vdp` sample model into memory.

See Also

`close_system` | `open_system`

Introduced before R2006a

model

Execute particular phase of simulation of model

Syntax

```
[sys,x0,str,ts] = model([],[],[],'sizes');
[sys,x0,str,ts] = model([],[],[],'compile');
outputs = model(t,x,u,'outputs');
derivs = model(t,x,u,'derivs');
dstates = model(t,x,u,'update');
model([],[],[],'term');
```

Description

The `model` command executes a specific phase of the simulation of a Simulink model whose name is `model`. The command's last argument (`flag`) specifies the phase of the simulation to be executed. See “Simulation Phases in Dynamic Systems” for a description of the steps that Simulink software uses to simulate a model.

This command ignores the effects of state transitions and conditional execution. Therefore, it is not suitable for models which have such logic. Use this command for models which can be represented as simple dynamic systems. Such systems should meet these requirements.

- All states in the model must be built-in non-bus data types. For a discussion on built-in data types, see “About Data Types in Simulink”.
- If you are using vector format to specify the state, this command can access only non-complex states of `double` data type.
- There is minimal amount of state logic (Stateflow, conditionally executed subsystems etc.)
- The models are not mixed-domain models. That is, most blocks in the model are built-in Simulink blocks and do not include user-written S-functions or blocks from other Sim* products.

For models which do not comply with these requirements, using this command can cause Simulink to produce results which can only be interpreted by further analyzing and simplifying the model.

Note: The state variable `x` can be represented in structure as well as vector formats. The variable follows the limitations of the format in which it is specified.

This command is also not intended to be used to run a model step-by-step, for example, to debug a model. Use the Simulink debugger if you need to examine intermediate results to debug a model.

Arguments

`sys`	Vector of model size data: • `sys(1)` = number of continuous states • `sys(2)` = number of discrete states • `sys(3)` = number of outputs • `sys(4)` = number of inputs • `sys(5)` = reserved • `sys(6)` = direct-feedthrough flag (1 = yes, 0 = no) • `sys(7)` = number of sample times (= number of rows in `ts`)
`x0`	Vector containing the initial conditions of the system's states
`str`	Vector of names of the blocks associated with the model's states. The state names and initial conditions appear in the same order in `str` and `x0`, respectively.
`ts`	An *m*-by-2 matrix containing the sample time (period, offset) information
outputs	Outputs of the model at time step `t`.
derivs	Derivatives of the continuous states of the model at time `t`.
dstates	Discrete states of the model at time `t`.
`t`	Time step, specified as real double in scalar format.
`x`	State vector, specified as real double in structure or vector format.

`u`	Inputs, specified as real double in vector format.
`flag`	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

The following command executes the compilation phase of the `vdp` model that comes with Simulink software.

```
vdp([], [], [], 'compile')
```

The following command terminates the simulation initiated in the previous example.

```
vdp([], [], [], 'term')
```

Note 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`

Introduced in R2007a

modeladvisor

Open Model Advisor

Syntax

```
modeladvisor(model)
```

Description

`modeladvisor(model)` opens the Model Advisor for the model or subsystem specified by `model`. If the specified model or subsystem is not open, this command opens it.

Examples

Open Model Advisor for model

Open the Model Advisor for `vdp` example model:

```
modeladvisor('vdp')
```

Open Model Advisor for subsystem

Open the Model Advisor for the Aircraft Dynamics Model subsystem of the `f14` example model:

```
modeladvisor('f14/Aircraft Dynamics Model')
```

Open Model Advisor for currently selected model

Open the Model Advisor on the currently selected model:

```
modeladvisor(bdroot)
```

Open Model Advisor for currently selected subsystem

Open the Model Advisor on the currently selected subsystem:

```
modeladvisor(gcs)
```

Input Arguments

`model` — Model or subsystem name

string

Model or subsystem name or handle, specified as a string.

Data Types: `char`

See Also

"Run Model Checks"

Introduced before R2006a

new_system

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 6-101 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`

Introduced before R2006a

num2fixpt

Convert number to nearest value representable by specified fixed-point data type

Syntax

```
outValue = num2fixpt(OrigValue, FixPtDataType, FixPtScaling,
    RndMeth, DoSatur)
```

Description

`num2fixpt(OrigValue, FixPtDataType, FixPtScaling, RndMeth, DoSatur)` returns the result of converting `OrigValue` to the nearest value representable by the fixed-point data type `FixPtDataType`. Both `OrigValue` and `outValue` are of data type `double`. As illustrated in the example that follows, you can use `num2fixpt` to investigate quantization error that might result from converting a number to a fixed-point data type. The arguments of `num2fixpt` include:

`OrigValue`	Value to be converted to a fixed-point representation. Must be specified using a `double` data type.
`FixPtDataType`	The fixed-point data type used to convert `OrigValue`.
`FixPtScaling`	Scaling of the output in either Slope or [Slope Bias] format. If `FixPtDataType` does not specify a generalized fixed-point data type using the `sfix` or `ufix` command, `FixPtScaling` is ignored.
`RndMeth`	Rounding technique used if the fixed-point data type lacks the precision to represent `OrigValue`. If `FixPtDataType` specifies a floating-point data type using the `float` command, `RndMeth` is ignored. Valid values are `Zero`, `Nearest`, `Ceiling`, or `Floor` (the default).
`DoSatur`	Indicates whether the output should be saturated to the minimum or maximum representable value upon underflow or overflow. If `FixPtDataType` specifies a floating-point data type using the `float` command, `DoSatur` is ignored. Valid values are `on` or `off` (the default).

Examples

Suppose you wish to investigate the quantization effect associated with representing the real-world value 9.875 as a signed, 8-bit fixed-point number. The command

```
num2fixpt(9.875, sfix(8), 2^-1)

ans =

   9.50000000000000
```

reveals that a slope of `2^-1` results in a quantization error of 0.375. The command

```
num2fixpt(9.875, sfix(8), 2^-2)

ans =

   9.75000000000000
```

demonstrates that a slope of `2^-2` reduces the quantization error to 0.125. But a slope of `2^-3`, as used in the command

```
num2fixpt(9.875, sfix(8), 2^-3)

ans =

   9.87500000000000
```

eliminates the quantization error entirely.

See Also

`fixptbestexp` | `fixptbestprec`

Introduced before R2006a

open_system

Open Simulink model, library, subsystem, or block dialog box

Syntax

```
open_system(obj)

open_system(sys,'loadonly')

open_system(sbsys,'window')
open_system(sbsys,'tab')

open_system(blk,'mask')
open_system(blk,'force')
open_system(blk,'parameter')
open_system(blk,'OpenFcn')
```

Description

`open_system(obj)` opens the specified model, library, subsystem, or block. This is equivalent to double-clicking the model or library in the Current Folder Browser, or the subsystem or block in the Simulink Editor.

A model or library opens in a new window. For a subsystem or block within a model, the behavior depends on the type of block and its properties.

- Any OpenFcn callback parameter is evaluated.
- If there is no OpenFcn callback, and a mask is defined, the mask parameter dialog box opens.
- Without an OpenFcn callback or a mask parameter, Simulink opens the object.
  - A referenced model opens in a new window.
  - A subsystem opens in a new tab in the same window.
  - For blocks, the parameters dialog box for the block opens.

To open a specific subsystem or block, you must load the model or library containing it. Otherwise Simulink returns an error.

You can override the default behavior by supplying a second input argument.

`open_system(sys,'loadonly')` loads the specified model or library without opening the Simulink Editor. This is equivalent to using `load_system`.

`open_system(sbsys,'window')` opens the subsystem `sbsys` in a new Simulink Editor window. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(sbsys,'tab')` opens the subsystem in a new Simulink Editor tab in the same window. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(blk,'mask')` opens the mask dialog box of the block or subsystem specified by `blk`. Load the model or library containing `blk` before opening it.

`open_system(blk,'force')` looks under the mask of a masked block or subsystem. It opens the dialog box of the block under the mask or opens a masked subsystems in a new Simulink Editor tab. This is equivalent to the **Look Under Mask** menu item. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(blk,'parameter')` opens the block parameter dialog box.

`open_system(blk,'OpenFcn')` runs the block callback `OpenFcn`.

Examples

Open a Model

Open the `f14` model.

```
open_system('f14')
```

Load a Model Without Opening it

Load the `f14` model.

```
open_system('f14','loadonly')
```

Open a Subsystem

Open the Controller subsystem of the `f14` model.

```
load_system('f14')
open_system('f14/Controller')
```

Open a Subsystem in New Tab in Existing Window

Open the f14 model and open the `Controller` subsystem in a new tab.

```
f14
open_system('f14/Controller','tab')
```

Open a Subsystem in a Separate Window

Open a subsystem in its own Simulink Editor window.

```
open_system('f14')
open_system('f14/Controller','window')
```

Open a Referenced Model

Open the model `sldemo_mdlref_counter`, which is referenced by the CounterA model block in `sldemo_mdlref_basic`.

```
open_system('sldemo_mdlref_basic')
open_system('sldemo_mdlref_basic/CounterA')
```

The referenced model opens in its own Simulink Editor window.

Open Block Dialog Box

Open the block parameters dialog box for the first Gain block in the `Controller` subsystem.

```
load_system('f14')
open_system('f14/Controller/Gain')
```

Run Block Open Callback Function

Define an `OpenFcn` callback for a block and execute the block callback.

```
f14
set_param('f14/Pilot','OpenFcn','disp(''Hello World!'')')
open_system('f14/Pilot','OpenFcn')
```

The words `Hello World` appear on the MATLAB Command Prompt.

Open Masked Subsystem

Open the contents of the masked subsystem Vehicle in the model `sf_car`.

```
open_system('sf_car')
open_system('sf_car/Vehicle', 'force')
```

Open Multiple Systems with One Command

Create a cell array of two model names, `f14` and `vdp`. Open both models using `open_system` with the cell array name.

```
models = {'f14','vdp'}
open_system(models)
```

Input Arguments

`obj` — Model, referenced model, library, subsystem, or block path

string

Model, referenced model, library, subsystem, or block path, specified as a string. If the model is not on the MATLAB path, specify the full path to the model file. Specify the block or subsystem using its full name, e.g., `f14/Controller/Gain`, on an opened or loaded model. On UNIX systems, the fully qualified path name of a model can start with a tilde (~), signifying your home directory.

Data Types: `char`

`sys` — Model or library path

string

The full name or path of a model or library, specified as a string.

Data Types: `char`

`sbsys` — Subsystem path

string

The full name or path of a subsystem in an open or loaded model, specified as a string.

Data Types: `char`

`blk` — Block or subsystem path

string

The full name or path of a block or subsystem in an open or loaded model, specified as a string.

Data Types: `char`

See Also

`close_system` | `load_system` | `new_system` | `save_system`

Introduced before R2006a

openDialog

Open configuration parameters dialog

Syntax

```
openDialog(configObj)
```

Arguments

configObj

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`openDialog` opens a configuration parameters dialog box. If *`configObj`* is a configuration set, the dialog box displays the configuration set. If *`configObj`* is a configuration reference, the dialog box displays the referenced configuration set, or generates an error if the reference does not specify a valid configuration set. If the dialog box is already open, its window becomes selected.

Examples

The following example opens a configuration parameters dialog box that shows the current parameters for the current model. The parameter values derive from the active configuration set or configuration reference (configuration object). The code is the same in either case; the only difference is which type of configuration object is currently active.

```
myConfigObj = getActiveConfigSet(gcs);
openDialog(myConfigObj);
```

More About

- "Manage a Configuration Set"

- “Manage a Configuration Reference”

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

Introduced in R2006b

performanceadvisor

Open Performance Advisor

Syntax

```
performanceadvisor(model)
```

Description

`performanceadvisor(model)` opens the Performance Advisor on the model or subsystem specified by model. If the specified model or subsystem is not open, this command opens it.

Input Arguments

`model`

A string specifying the name or handle to the model or subsystem.

Examples

Open Performance Advisor

Open Performance Advisor on the `vdp` example model.

```
performanceadvisor('vdp')
```

Performance Advisor opens the `vdp` model and opens Performance Advisor on the model.

- “Improve Simulation Performance Using Performance Advisor ”
- “Perform a Quick Scan Diagnosis”
- “Improve vdp Model Performance”

Alternatives

In the Simulink Editor, select **Analysis > Performance Tools > Performance Advisor**.

More About

- "Performance Advisor Window"

Introduced in R2013a

reload

Reload Simulink Project

Syntax

```
reload(proj)
```

Description

`reload(proj)` reloads the project. Use `reload` when you want to run the project startup shortcuts.

Examples

Reload Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

When you want to run the startup shortcuts again, reload the project.

```
reload(proj)
```

Input Arguments

`proj` — Project

project object

Project, specified as a project object already created with `simulinkproject` to manipulate a Simulink Project at the command line.

See Also

Functions

`isLoaded` | `simulinkproject`

Introduced in R2013a

removeCategory

Remove Simulink Project category of labels

Syntax

```
removeCategory(proj,categoryName)
```

Description

`removeCategory(proj,categoryName)` removes a category of labels, `categoryName`, from the Simulink Project specified by `proj`.

Examples

Remove Category

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Create a new category of labels.

```
createCategory(proj,'Engineers','char');
```

Remove the new category of labels.

```
removeCategory(proj,'Engineers');
```

A message appears warning you that you cannot undo the operation. Click **Continue**. You can configure warnings in the Preferences in the Simulink Project Tool.

Input Arguments

`proj` — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

`categoryName` — Name of category

string

Name of the category to remove, which exists in the project, specified as a string.

See Also

Functions

`createCategory` | `findCategory` | `simulinkproject`

Introduced in R2013a

removeFile

Remove file from Simulink Project

Syntax

```
removeFile(proj,file)
```

Description

`removeFile(proj,file)` removes a file from the project `proj`.

Examples

Remove File from Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Remove a file.

```
removeFile(proj,'models/AnalogControl.mdl')
```

Add the file back to the project.

```
addFile(proj,'models/AnalogControl.mdl')
```

Input Arguments

`proj` — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

`file` — Path of file

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 in the project.

Example: `'models/myModelName.slx'`

See Also

Functions

`addFile` | `findFile` | `simulinkproject`

Introduced in R2013a

removeLabel

Remove label from Simulink Project

Syntax

```
removeLabel(category,labelName)
removeLabel(file,categoryName,labelName)
removeLabel(file,labelDefinition)
```

Description

`removeLabel(category,labelName)` removes the label from the specified category of labels in the currently loaded project.

`removeLabel(file,categoryName,labelName)` removes the specified label in the category `categoryName` from the file. Use this syntax to specify category and label by name.

`removeLabel(file,labelDefinition)` removes the specified label `labelDefinition` from the file. Before you can remove the label, you need to get the label from the `file.Label` property or by using `findLabel`.

Examples

Remove a Label

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Examine the first existing category.

```
cat = proj.Categories(1)
```

```
cat =
```

```
  Category with properties:

                Name: 'Classification'
            DataType: 'none'
    LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat,'Future');
```

Remove the new label.

```
removeLabel(cat,'Future');
```

Input Arguments

`category` — Category of labels

category object

Category of labels, specified as a category object. Get a category object from the `proj.Categories` property or by using `findCategory`.

`labelName` — Name of label

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`.

See Also

Functions

`addLabel` | `createLabel` | `findCategory` | `findLabel` | `simulinkproject`

Introduced in R2013a

replace_block

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 6-101.

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/Unlocked','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`

Introduced before R2006a

save_system

Save Simulink system

Syntax

```
save_system
save_system(sys)
save_system(sys, newsysname)
save_system(sys, newsysname.slx)
save_system(sys, newsysname, Name,Value)
save_system(sys, 'exported_file_name.xml', 'ExportToXML', true)
filename = save_system(sys)
```

Description

`save_system` saves the current top-level system. If the system has not previously been saved, `save_system` creates a new file in the current folder.

`save_system(sys)` saves the top-level system that you specify in `sys` to a file using the current system name. `sys` must be a system name with no file extension. The system must be loaded. `sys` can be a 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)`

'AllowPrompt'

Logical value that indicates whether to display any output prompt or message in a dialog box or only messages at the command line. For example, prompts to make files writable, or messages about exported versions. If you want to allow prompts, then set to true. Also accepts on or off.
true
false (default)

Default: false

'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 to builtin libraries, use the Upgrade Advisor to scan the model for out-of-date blocks and upgrade the Simulink blocks to their current versions.

`'BreakUserLinks'`

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.

You can also use `Simulink.exportToVersion`.

`SaveAsVersion` is a legacy option for this argument that is also supported.

These version names are not case sensitive:

```
'R14'
'R14SP1'
'R14SP2'
'R14SP3'
'R2006A'
'R2006B'
'R2007A'
'R2007B'
'R2008A'
'R2008B'
'R2009A'
'R2009B'
'R2010A'
'R2010B'
'R2011A'
'R2011B'
'R2012A'
'R2012A_MDL'
'R2012A_SLX'
'R2012B'
'R2012B_MDL'
'R2012B_SLX'
'R2013A'
'R2013A_MDL'
'R2013A_SLX'
'R2013B'
'R2013B_MDL'
```

```
'R2013B_SLX'
'R2014A'
'R2014A_MDL'
'R2014A_SLX'
'R2014B'
'R2014B_MDL'
'R2014B_SLX'
'R2015A'
'R2015A_MDL'
'R2015A_SLX'
```

If you use the Export to Previous Version dialog box instead of `save_system`, then the **Save as type** list supports 7 years of previous releases.

`'OverwriteIfChangedOnDisk'`

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 Source for Data in Model Workspace”. 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')
```

More About

- "Save a Model"

See Also

`close_system` | `new_system` | `open_system` | `Simulink.exportToVersion`

Introduced before R2006a

set_param

Set system and block parameter values

Syntax

```
set_param(Object,ParameterName,Value,...ParameterNameN,ValueN)
```

Description

`set_param(Object,ParameterName,Value,...ParameterNameN,ValueN)` sets the parameter to the specified value on the specified model or block object.

When you set multiple parameters on the same model or block, use a single `set_param` command with multiple pairs of `ParameterName`, `Value` arguments, rather than multiple `set_param` commands. This technique is efficient because using a single call requires evaluating parameters only once. If any parameter names or values are invalid, then the function doesn't set any parameters.

Tips:

- If you make multiple calls to `set_param` for the same block, then specifying the block using a numeric handle is more efficient than using the full block path. Use `getSimulinkBlockHandle` to get a block handle.
- If you use `matlab -nodisplay` to start a session, you cannot use `set_param` to run your simulation. The `-nodisplay` mode does not support simulation using `set_param`. Use the `sim` command instead.
- After you set parameters in the MATLAB workspace, to see the changes in a model, update the diagram.

  ```
  set_param(model,'SimulationCommand','Update')
  ```

For parameter names, see:

- "Model Parameters" on page 6-2
- "Block-Specific Parameters" on page 6-101
- "Common Block Properties" on page 6-86

Examples

Set Model Configuration Parameters for a Model

Open `vdp` and set the `Solver` and `StopTime` parameters.

```
vdp
set_param('vdp','Solver','ode15s','StopTime','3000')
```

Set Model Configuration Parameters for Current Model

Open a model and set the `Solver` and `StopTime` parameters. Use `bdroot` to get the current top-level model.

```
vdp
set_param(bdroot,'Solver','ode15s','StopTime','3000')
```

Set a Gain Block Parameter Value

Open `vdp` and set a `Gain` parameter value in the `Mu` block.

```
vdp
set_param('vdp/Mu','Gain','10')
```

Set Position of Block

Open `vdp` and set the position of the `Fcn` block.

```
vdp
set_param('vdp/Fcn','Position',[50 100 110 120])
```

Set Position of Block Using a Handle

Set the position of the `Fcn` block in the `vdp` model.

Use `getSimulinkBlockHandle` to load the `vdp` model if necessary (by specifying `true`), and get a handle to the `Fcn` block. If you make multiple calls to `set_param` for the same block, then using the block handle is more efficient than specifying the full block path as a string.

```
fcnblockhandle = getSimulinkBlockHandle('vdp/Fcn',true);
```

You can use the block handle in subsequent calls to `get_param` or `set_param`. If you examine the handle, you can see that it contains a double. Do not try to use the number

of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Instead, assign the handle to a variable and use that variable name to specify a block.

Use the block handle with `set_param` to set the position.

```
set_param(fcnblockhandle,'Position',[50 100 110 120])
```

- "Associating User Data with Blocks"
- "Use MATLAB Commands to Change Workspace Data"
- "Control Simulation Using the set_param Command"
- "Simulate a Model Interactively"

Input Arguments

`Object` — Name or handle of a model or block

string | handle

Handle or name of a model or block, specified as a numeric handle or a string. A numeric handle must be a scalar. You can also set parameters of lines and ports, but you must use numeric handles to specify them.

Tip If you make multiple calls to `set_param` for the same block, then specifying a block using a numeric handle is more efficient than using the full block path with `set_param`. Use `getSimulinkBlockHandle` to get a block handle. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Assign the handle to a variable and use that variable name to specify a block.

Example: `'vdp/Fcn'`

`ParameterName` — Model or block parameter name

string

Model or block parameter name, specified as the comma-separated pair consisting of the parameter name, specified as a string, and the value, specified in the format determined by the parameter type. Case is ignored for parameter names. Value strings are case sensitive. Values are often strings, but they can also be numeric, arrays, and other types.

Many block parameter values are specified as strings, but two exceptions are these parameters: `Position`, specified as a vector, and `UserData`, which can be any data type.

Example: `'Solver','ode15s','StopTime','3000'`

Example: `'SimulationCommand', 'start'`

Example: `'Position',[50 100 110 120]`

Data Types: `char`

More About

- "Model Parameters" on page 6-2
- "Block-Specific Parameters" on page 6-101
- "Common Block Properties" on page 6-86

See Also

`bdroot` | `gcb` | `gcs` | `get_param` | `getSimulinkBlockHandle`

Introduced before R2006a

setActiveConfigSet

Specify model's active configuration set or configuration reference

Syntax

```
setActiveConfigSet(model, configObjName)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObjName`

The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`setActiveConfigSet` specifies the active configuration set or configuration reference (configuration object) of `model` to be the configuration object specified by `configObjName`. If no such configuration object is attached to the model, an error occurs. The previously active configuration object becomes inactive.

Examples

The following example makes `DevConfig` the active configuration object of the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
setActiveConfigSet(gcs, 'DevConfig');
```

More About

- "Manage a Configuration Set"

- “Manage a Configuration Reference”

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog`

Introduced before R2006a

sfix

Create `Simulink.NumericType` object describing signed fixed-point data type

Syntax

```
a = sfix(WordLength)
```

Description

`sfix(WordLength)` returns a `Simulink.NumericType` object that describes a signed fixed-point number with the specified word length and unspecified scaling.

Note: `sfix` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `sfix(WordLength)` with `fixdt(1,WordLength)`.

Examples

Define a 16-bit signed fixed-point data type.

```
a = sfix(16)
```

```
a =

  NumericType with properties:

    DataTypeMode: 'Fixed-point: unspecified scaling'
      Signedness: 'Signed'
      WordLength: 16
         IsAlias: 0
       DataScope: 'Auto'
      HeaderFile: ''
     Description: ''
```

See Also

`fixdt` | `Simulink.NumericType` | `float` | `sfrac` | `sint` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

sfrac

Create `Simulink.NumericType` object describing signed fractional data type

Syntax

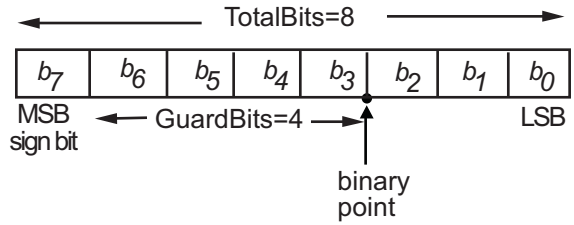
```
a = sfrac(WordLength)
a = sfrac(WordLength, GuardBits)
```

Description

`sfrac(`*`WordLength`*`)` returns a `Simulink.NumericType` object that describes the data type of a signed fractional data type with a word size given by *`WordLength`*.

`sfrac(`*`WordLength`*`, `*`GuardBits`*`)` returns a `Simulink.NumericType` object that describes the data type of a signed fractional number. The total word size is given by *`WordLength`* with *`GuardBits`* bits located to the left of the binary point.

The most significant (leftmost) bit is the sign bit. The default binary point for this data type is assumed to lie immediately to the right of the sign bit. If guard bits are specified, they lie to the left of the binary point and to right of the sign bit. For example, the structure for an 8-bit signed fractional data type with 4 guard bits is:



Note: `sfrac` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `sfrac(WordLength,GuardBits)` with `fixdt(1,WordLength,(WordLength-1-GuardBits))` and `sfrac(WordLength)` with `fixdt(1,WordLength,(WordLength-1))`.

Examples

Define an 8-bit signed fractional data type with 4 guard bits. Note that the range of this data type is $-2^4 = -16$ to $(1 - 2^{(1-8)}).2^4 = 15.875$.

```
a = sfrac(8,4)
```

```
a =

  NumericType with properties:

      DataTypeMode: 'Fixed-point: binary point scaling'
        Signedness: 'Signed'
        WordLength: 8
    FractionLength: 3
           IsAlias: 0
         DataScope: 'Auto'
        HeaderFile: ''
       Description: ''
```

See Also

`fixdt` | `Simulink.NumericType` | `float` | `sfix` | `sint` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

signalbuilder

Create and access Signal Builder blocks

Syntax

```
[time, data] = signalbuilder(block)
[time, data, signames] = signalbuilder(block)
[time, data, signames, groupnames] = signalbuilder(block)
block = signalbuilder([], 'create', time, data, signames,
groupnames)
block = signalbuilder(path, 'create', time, data, signames,
groupnames)
block = signalbuilder(path,'create', time, data, signames,
groupnames, vis)
block = signalbuilder(path,'create', time, data, signames,
groupnames, vis, pos)
block = signalbuilder(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.

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.

Configuration	Time/Data Format
>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

Field	Description	Example Value
`timeRange`	Two-element vector specifying the time range to print (must not exceed the block's time range)	`[3 6]`
`visibleSignals`	Vector specifying index of signals to print	`[1 2]`
`yLimits`	Cell array specifying limits for each signal's *y*-axis	`{[-1 1], [0 1]}`
`extent`	Two-element vector of the form: `[width, height]` specifying the dimensions (in pixels) of the area in which to print the signals	`[500 300]`
`showTitle`	Logical value specifying whether to print a title; `true` (1) prints the title	`false`

Set up the structure with one or more of these fields before you print. For example, if you want to print just group 2 using a configuration structure, `configstruct`, set up the structure as follows. You do not need to specify any other fields.

```
configstruct.groupIndex=2
```

The optional argument *`printArgs`* allows you to configure print options (see `print` in the MATLAB Function Reference).

`figh = signalbuilder(`*`block`*`, 'print', `*`config`*`, 'figure')` prints the currently active signal group or the signal group that *`config`* specifies to a new hidden figure handle, `figh`.

Interpolating Missing Data Values

When specifying a periodic signal such as a Sine Wave, the `signalbuilder` function uses linear Lagrangian interpolation to compute data values for time steps that occur between time steps for which the `signalbuilder` function supplies data. When specifying periodic signals, specify them as a time vector that is defined as multiples of sample time, for example:

```
t = 0.2*[0:49]';
```

Examples

Example 1

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 5], {[2 2];[0 2]});
```

Get signal builder data from this block.

```
[time, data, signames, groupnames] = signalbuilder('untitled/Signal Builder')
```

```
time =

    [1x2 double]
    [1x2 double]

data =

    [1x2 double]
    [1x2 double]

signames =

    'Signal 1'    'Signal 2'

groupnames =

    'Group 1'
```

The Signal Builder block contains two signals in one group. Alter the second signal in the group:

```
signalbuilder(block, 'set', 2, 1, [0 5], [2 0])
```

To make this same change using the signal name and group name:

```
signalbuilder(block, 'set', 'Signal 2', 'Group 1', [0 5], [2 0])
```

Delete the first signal from the group:

```
signalbuilder(block, 'set', 1, 1, [], [])
```

Append the group with a new signal:

```
signalbuilder(block, 'append', [0 2.5 5], [0 2 0], 'Signal 2', 'Group 2');
```

Append another group with a new signal using `appendgroup`:

```
signalbuilder(block, 'appendgroup', [0 2.5 5], [0 2 0], 'Signal 2', 'Group 3');
```

Example 2

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 2], {[0 1],[1 0]});
```

The Signal Builder block has two groups, each of which contains a signal. To delete the second group, also delete its signal:

```
signalbuilder(block, 'set', 1, 2, [], [])
```

Example 3

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 1], ...
         {[0 0],[1 1];[1 0],[0 1];[1 1],[0 0]});
```

The Signal Builder block has two groups, each of which contains three signals.

Example 4

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([],'create',{[0 10],[0 20]},{[6 -6],...
[2 5]});
```

The Signal Builder block has two groups. Each group contains one signal.

Append a new signal group to the existing block.

```
block = signalbuilder(block,'append',[0 30],[10 -10]);
```

Append a new signal, `sig3`, to all groups.

```
signalbuilder(block,'appendsignal',[0 30],[0 10],'sig3');
```

Example 5

Create a Signal Builder block in a new model editor window:

```
time = [0 1];
data = {[0 0],[1 1];[1 0],[0 1];[1 1],[0 0]};
block = signalbuilder([], 'create', time, data);
```

The Signal Builder block has two groups. Each group contains three signals.

Delete the second group. To delete a signal group, also delete all the signals in the group.

```
signalbuilder(block, 'set',[1,2,3],'Group 2',[]);
```

Example 6

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 5], {[2 2];[0 2]});
```

The Signal Builder block has one group that contains two signals.

Hide the signal, `Signal 1`.

```
signalbuilder(block,'hidesignal','Signal 1', 'Group 1')
```

`Signal 1` is no longer visible in the Signal Builder block.

Make `Signal 1` visible again.

```
signalbuilder(block,'showsignal','Signal 1', 'Group 1')
```

Example 7

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 5], {[2 2] [0 2]});
```

The Signal Builder block has two groups, each with one signal.

Create a structure, `configstruct`, to customize the Signal Builder block that you want to print.

```
configstruct.groupIndex = 2;
configstruct.timeRange = [0 2];
configstruct.visibleSignals = 1;
configstruct.yLimits = {[0 1]};
configstruct.extent = [500 300];
configstruct.showTitle = true;
```

This sequence fills all the fields of the `configstruct` structure.

Print a view of the Signal Builder block to the default printer. The `configstruct` structure defines the view to print.

```
signalbuilder(block,'print',configstruct)
```

Print with print options, for example `-dps`.

```
signalbuilder(block,'print',configstruct,'-dps')
```

Print a view of the Signal Builder block as defined by the `configstruct` structure to a new hidden figure handle, `figH`.

```
figH = signalbuilder(block,'print',configstruct,'figure')
figure(figH)
```

Introduced in R2007a

sim

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 to simulate the block diagram, *`model`*, using parameter name-value pairs `ParameterName1`, `Value1` and `ParameterName2`, `Value2`.

`simOut = sim('model', ParameterStruct);` causes Simulink to simulate the block diagram, *`model`*, using the parameter values specified in the structure *`ParameterStruct`*.

`simOut = sim('model', ConfigSet);` causes Simulink to simulate the block diagram, *`model`*, using the configuration settings specified in the model configuration set, *`ConfigSet`*.

When you do not specify an output argument in the `sim` command, Simulink stores the simulation output in the variable *ans*.

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
`ConcurrencyResolvingTo-FileSuffix`	(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'`.
`LoggingFileName | {'out.mat'}`	Use when you enable `LoggingToFile` name-value pair for logging to persistent storage. Specify the destination MAT-file for data logging. Do not use a file name from one locale in a different locale.
`LoggingToFile 'on' | {'off'} |`	Store logging data that uses `Dataset` format to persistent storage (MAT-file). Using a `Simulink.SimlationData.DatasetRef` object to access signal logging and states logging data loads data into the model workspace incrementally. Accessing data for other kinds of logging loads all of the data at once. Use this feature when logging large amounts of data that can cause memory issues. For details, see "Log Data Using Persistent Storage".
`RapidAccelerator-ParameterSets`	(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".

Parameter	Description
`RapidAcceleratorUpTo-DateCheck {'on'} | 'off'`	(Rapid Accelerator mode only) Enables/disables up-to-date check. If you set this value to `'off'`, Simulink does not perform an up-to-date check. It skips the start/stop callbacks in blocks. If you call the `sim` command from `parfor`, set this value to `'off'`. 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. Setting `SrcWorkspace` to `current` within a `parfor` loop causes a transparency violation. See "Transparency Violation" for more details. Default is the base workspace.
`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 a warning and stops the simulation.
`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 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.

When simulating a model with infinite stop time, to stop the simulation, you must press **Ctrl+C**.

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

Rapid Accelerator Simulations Using PARFOR | "Backwards Compatible Syntax" | `parfor` | "sim in parfor with Rapid Accelerator Mode" | `sldebug` | "Log Data Using Persistent Storage" | "Configuration Parameters Dialog Box Overview"

Introduced before R2006a

simplot

Redirects to the Simulation Data Inspector

Compatibility

`simplot` will be removed in a future release. Use the Simulation Data Inspector instead.

Syntax

```
simplot
```

Description

`simplot` redirects to the Simulation Data Inspector and returns empty handles. This function is no longer supported and has been replaced by the Simulation Data Inspector. Use the **Simulation Data Inspector** button in the Simulink Editor to capture simulation output in the Simulation Data Inspector. Programmatically, use the function `Simulink.sdi.view` instead.

See Also

`Simulink.sdi.view`

simulink

Open Simulink Start Page

Syntax

```
simulink
```

Description

`simulink` opens the Simulink Start Page. Choose a model or project template, or browse the examples. When you create a new model, the Simulink Library Browser opens.

The behavior of the `simulink` function changed in R2016a. Formerly it opened the Simulink Library Browser and loaded the Simulink block library. If you wish to preserve that behavior, use `slLibraryBrowser` instead to open the Library Browser, or `load_system simulink` to load the Simulink block library.

If you want to start Simulink without opening the Library Browser or Start Page, use the faster `start_simulink` instead.

See Also

`slLibraryBrowser` | `start_simulink`

Related Examples

- “Create Models and Open Existing Models”
- “Modeling”

Introduced before R2006a

simulinkproject

Open Simulink Project and get project object

Syntax

```
simulinkproject
simulinkproject(projectPath)

proj = simulinkproject
proj = simulinkproject(projectPath)
```

Description

`simulinkproject` opens Simulink Project or brings focus to the tool if it is already open. After you open the tool, you can create new projects or access recent projects using the **Simulink Project** tab.

`simulinkproject(projectPath)` opens the Simulink project specified by any file or folder under the project root in `projectPath` and gives focus to Simulink Project.

`proj = simulinkproject` returns a project object `proj` you can use to manipulate the project at the command line. You need to get a project object before you can use any of the other project functions.

If you want to avoid giving focus to Simulink Project in your startup script, specify an output argument.

`proj = simulinkproject(projectPath)` opens the Simulink project specified by `projectPath` and returns a project object.

To avoid your startup script opening windows that take focus away from the MATLAB Desktop, use `start_simulink` instead of the `simulink` function, and use `simulinkproject` with an output argument instead of `uiopen`.

Examples

Open Simulink Project Tool

Open the Simulink Project Tool.

```
simulinkproject
```

Open a Simulink Project

Specify either the `.prj` file path or the folder that contains your `.SimulinkProject` folder and `.prj` file. The project opens and brings focus to Simulink Project.

```
simulinkproject('C:/projects/project1/')
```

Open a Simulink Project and Get a Project Object

Open a specified project and get a project object to manipulate the project at the command line. To avoid your startup script opening windows that take focus away from the MATLAB Desktop, use `start_simulink` instead of the `simulink` function, and use `simulinkproject` with an output argument instead of `uiopen`. If you use `uiopen(myproject.prj)` this calls `simulinkproject` with no output argument and gives focus to Simulink Project.

```
start_simulink
proj = simulinkproject('C:/projects/project1/myproject.prj')
```

Get Airframe Example Project

Open the Airframe project and create a project object to manipulate and explore the project at the command line.

```
sldemo_slproject_airframe
proj = simulinkproject
```

```
proj =

  ProjectManager with properties:

            Name: 'Simulink Project Airframe Example'
      Categories: [1x1 slproject.Category]
       Shortcuts: [1x8 slproject.Shortcut]
     ProjectPath: [1x7 slproject.PathFolder]
ProjectReferences: [1x0 slproject.ProjectReference]
```

```
           Files: [1x30 slproject.ProjectFile]
      RootFolder: 'C:\Work\Simulink\Projects\slexamples\airframe'
```

Find Project Commands

Find out what you can do with your project.

```
methods(proj)
```

```
Methods for class slproject.ProjectManager:

addFile                         findCategory
addFolderIncludingChildFiles    findFile
close                           isLoaded
createCategory                  listModifiedFiles
export                          refreshSourceControl


reload
removeCategory
removeFile
```

Examine Project Properties Programmatically

After you get a project object using the `simulinkproject` function, you can examine project properties.

Examine the project files.

```
files = proj.Files
```

```
files =

  1x30 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Use indexing to access files in this list. The following command gets file number 13. Each file has properties describing its path, attached labels, and source control information.

```
proj.Files(13)
```

```
ans =
```

```
  ProjectFile with properties:

      Path: 'C:\Work\slexamples\airframe\models\AnalogControl.mdl'
            Labels: [1x1 slproject.Label]
          Revision: '2'
SourceControlStatus: Unmodified
```

Examine the labels of the 13th file.

```
proj.Files(13).Labels
```

```
ans =

  Label with properties:

File: 'C:\Work\slexamples\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:\Work\slexamples\\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
  Revision: '2'
SourceControlStatus: Unmodified
```

Find out what you can do with the file.

```
methods(myfile)
```

```
Methods for class slproject.ProjectFile:

addLabel
findLabel
removeLabel
```

- “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”

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, or any subfolder or file under your project root, 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
Name	Project name
Categories	Categories of project labels
Shortcuts	Shortcut files in project
ProjectPath	Folders that the project puts on the MATLAB path
ProjectReferences	Folders that contain referenced projects
Files	Paths and names of project files
RootFolder	Full path to project root folder

More About

Tips

Alternatively, you can use `slproject.loadProject` to load a project, and `slproject.getCurrentProjects` to get a project object. Use `simulinkproject` to open projects and explore projects interactively. Use `slproject.getCurrentProjects` and `slproject.loadProject` for project automation scripts.

- "What Are Simulink Projects?"

See Also

Functions

`addFile` | `addFolderIncludingChildFiles` | `addLabel` | `createCategory` | `findFile` | `findLabel` | `listModifiedFiles` | `refreshSourceControl` | `removeFile` | `slproject.getCurrentProjects` | `slproject.loadProject` | `start_simulink`

Introduced in R2012a

Simulink.architecture.add

Add tasks or triggers to selected architecture of model

Syntax

```
Simulink.architecture.add(Type,Object)
```

Description

`Simulink.architecture.add(Type,Object)` adds the new task or trigger `Object` of the specified `Type` to a model.

Examples

Add periodic trigger

Add a periodic trigger, MyTrigger1, to the software node CPU of the selected architecture of the `sldemo_concurrent_execution` model.

```
sldemo_concurrent_execution;
Simulink.architecture.add('PeriodicTrigger','sldemo_concurrent_execution/CPU/MyTrigger1');
```

Input Arguments

`Type` — Object type

'PeriodicTrigger' | 'AperiodicTrigger' | 'Task'

Object type that identifies the kind of trigger or task to add, , specified as a `'PeriodicTrigger'`, `'AperiodicTrigger'`, or `'Task'`.

- `'PeriodicTrigger'`

  Adds a periodic trigger to the architecture. Set the properties of the trigger with the `Simulink.architecture.set_param` function.

- `'AperiodicTrigger'`

Adds an aperiodic trigger to the architecture. Set the properties of the trigger with the `Simulink.architecture.set_param` function.

- `'Task'`

  Adds a task to the architecture. Set the properties of the task with the `Simulink.architecture.set_param` function.

`Object` — Trigger or task object identifier

string

Trigger or task object identifier to add to architecture, specified as a string.

Example: `'sldemo_concurrent_execution/CPU/MyTrigger1'`

Data Types: `char`

See Also

`Simulink.architecture.delete` | `Simulink.architecture.find_system` | `Simulink.architecture.get_param` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.register` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.config

Create or convert configuration for concurrent execution

Syntax

```
Simulink.architecture.config(model,'Convert')
Simulink.architecture.config(model,'Add')
Simulink.architecture.config(model,'OpenDialog')
```

Description

`Simulink.architecture.config(model,'Convert')` converts the active configuration set in the specified model to one for concurrent execution.

`Simulink.architecture.config(model,'Add')` adds and activates a new configuration set for concurrent execution.

`Simulink.architecture.config(model,'OpenDialog')` opens the Concurrent Execution dialog box for a model configuration.

Examples

Convert existing configuration set

Convert existing configuration set for concurrent execution in the model vdp.

```
vdp;
Simulink.architecture.config('vdp','Convert');
```

Add new configuration set

Add a new configuration set (copied from the existing configuration set) for concurrent execution in the model vdp.

```
vdp;
```

```
Simulink.architecture.config('vdp','Add');
```

Open Concurrent Execution dialog box

Open the Concurrent Execution dialog box in the model `sldemo_concurrent_execution`.

```
sldemo_concurrent_execution;
Simulink.architecture.config('sldemo_concurrent_execution','OpenDialog');
```

Input Arguments

`model` — Model name

string

Model name whose configuration set you want to convert or add to, specified as a string.

Example:

Data Types: `char`

See Also

`Simulink.architecture.add` | `Simulink.architecture.profile` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.delete

Delete triggers and tasks from selected architecture of model

Syntax

```
Simulink.architecture.delete(Object)
```

Description

`Simulink.architecture.delete(Object)` deletes the specified object trigger or task.

Examples

Delete task Plant

Delete the task Plant from the Periodic trigger of the CPU software node of the selected architecture of the model `sldemo_concurrent_execution`.

```
sldemo_concurrent_execution
Simulink.architecture.delete('sldemo_concurrent_execution/CPU/Periodic/Plant')
```

Input Arguments

`Object` — Object to delete, specified as a string

string

Object to be deleted. Possible objects are:

- Periodic trigger

> **Note:** You cannot delete the last periodic trigger. The software node must contain at least one periodic trigger.

- Aperiodic trigger
- Task

Example: `[bdroot '/CPU/Periodic/ControllerB']`

Data Types: `char`

See Also

`Simulink.architecture.add` | `Simulink.architecture.find_system` | `Simulink.architecture.get_param` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.register`

Introduced in R2014a

Simulink.architecture.find_system

Find objects under architecture object

Syntax

```
object = Simulink.architecture.find_system(RootObject)

object = Simulink.architecture.find_system(RootObject,ParamName,
ParamValue)
```

Description

`object = Simulink.architecture.find_system(RootObject)` looks for all objects under `RootObject`.

`object = Simulink.architecture.find_system(RootObject,ParamName, ParamValue)` returns the object in `RootObject` whose parameter `ParamName` has the value `ParamValue`. Parameter name and value strings are case-sensitive.

Examples

Look for all objects

To find all the objects in `sldemo_concurrent_execution`:

```
sldemo_concurrent_execution
t = Simulink.architecture.find_system('sldemo_concurrent_execution')

t =

    'sldemo_concurrent_execution'
    'sldemo_concurrent_execution/CPU'
    'sldemo_concurrent_execution/CPU/Periodic'
    'sldemo_concurrent_execution/CPU/Periodic/ControllerA'
    'sldemo_concurrent_execution/CPU/Periodic/ControllerB'
    'sldemo_concurrent_execution/CPU/Periodic/Plant'
```

```
    'sldemo_concurrent_execution/CPU/Interrupt'
```

Look for all tasks

To find all the tasks in `sldemo_concurrent_execution`:

```
sldemo_concurrent_execution
t = Simulink.architecture.find_system('sldemo_concurrent_execution','Type','Task')
```

```
t =

    'sldemo_concurrent_execution/CPU/Periodic/ControllerA'
    'sldemo_concurrent_execution/CPU/Periodic/ControllerB'
    'sldemo_concurrent_execution/CPU/Periodic/Plant'
```

Input Arguments

`RootObject` — Object to search

string

Object to search for parameter value, specified as a string giving the object full path name. Possible objects are:

- Model
- Software node
- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

Example: `'sldemo_concurrent_execution'`

`ParamName` — Name of parameter to find

string | scalar | vector

Name of the parameter to find, specified as a string. Possible string values are:

- `'Name'`
- `'Type'`
- `'ClockFrequency'`

- `'Color'`
- `'Period'`
- `'EventHandlerType'`
- `'SignalNumber'`
- `'EventName'`

Example: `'EventName'`

`ParamValue` — Parameter value to find

string | scalar | vector

Parameter value to find, specified as a string, a scalar, or a vector.

Example: `'ERTDefaultEvent'`

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.register` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.get_param

Get configuration parameters of architecture objects

Syntax

```
ParamValue = Simulink.architecture.get_param(Object,ParamName)
```

Description

`ParamValue = Simulink.architecture.get_param(Object,ParamName)` returns the value of the specified parameter for the object, `Object`. `ParamName` is case-sensitive.

Examples

Get period

Get the period of task Plant of trigger Periodic of software node CPU of the selected architecture for the model `sldemo_concurrent_execution`.

```
sldemo_concurrent_execution;
p = Simulink.architecture.get_param('sldemo_concurrent_execution/CPU/Periodic/Plant','Period')
```

```
p =

0.1
```

Input Arguments

`Object` — Object whose parameter value to return

string

Object whose parameter value to return, specified as a string giving the object full path name. Possible objects are:

- Software node

- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

`ParamName` — Parameter whose value to return

string

Name of a parameter for which `Simulink.architecture.get_param` returns a value.

The following are the possible `ParamName` strings:

For a model:

- `'ArchitectureName'`
- `'Type'`

For a software node:

- `'Name'`
- `'Type'`

For a hardware node

- `'Name'`
- `'ClockFrequency'`
- `'Color'`
- `'Type'`

For a periodic trigger:

- `'Name'`
- `'Period'`
- `'Color'`
- `'Type'`

For an aperiodic trigger:

- `'Name'`

- 'Color'
- 'EventHandlerType'
- 'SignalNumber'
- 'EventName'
- 'Type'

For a task:

- 'Name'
- 'Period'
- 'Color'
- 'Type'

See Also

Simulink.architecture.add | Simulink.architecture.delete | Simulink.architecture.find_system | Simulink.architecture.importAndSelect | Simulink.architecture.profile | Simulink.architecture.register | Simulink.architecture.set_param

Introduced in R2014a

Simulink.architecture.importAndSelect

Import and select target architecture for concurrent execution environment for model

Syntax

```
Simulink.architecture.importAndSelect(model,Architecture)
```

```
Simulink.architecture.importAndSelect(model,
CustomArchitectureDescriptionFile)
```

Description

`Simulink.architecture.importAndSelect(model,Architecture)` imports and selects the built-in target architecture for the concurrent execution environment for the model.

`Simulink.architecture.importAndSelect(model, CustomArchitectureDescriptionFile)` imports and selects the architecture from an XML-based architecture description file.

Importing and selecting target architectures requires that the associated support packages or hardware is installed on your computer.

Examples

Import and select a different architecture

Import and select the sample architecture to the model `sldemo_concurrent_execution`.

```
sldemo_concurrent_execution
Simulink.architecture.importAndSelect('sldemo_concurrent_execution','Sample Architectu
```

Import and select a custom architecture

Import and select the custom architecture defined in the XML file `custom_arch.xml`. This example requires you to create a `custom_arch.xml` first.

```
sldemo_concurrent_execution
Simulink.architecture.importAndSelect('sldemo_concurrent_execution','custom_arch.xml')
```

Input Arguments

`model` — Model

string

Model to import architecture to, specified as a string.

Data Types: `char`

`Architecture` — Target architecture name

string

Target architecture name to import into the concurrent execution environment for the model, specified as a string. Possible target names are:

Property	Description
`'Multicore'`	Single CPU with multiple cores
`'Sample Architecture'`	Example architecture consisting of single CPU with multiple cores and two FPGAs. You can use this architecture to model for concurrent execution.
`'Simulink Real-Time'`	Simulink Real-Time™ target
`'Xilinx Zynq ZC702 evaluation kit'`	Xilinx® Zynq® ZC702 evaluation kit target
`'Xilinx Zynq ZC706 evaluation kit'`	Xilinx Zynq ZC706 evaluation kit target
`'Xilinx Zynq Zedboard'`	Xilinx Zynq ZedBoard™ target

Data Types: `char`

`CustomArchitectureDescriptionFile` — Custom target architecture file

XML file

Custom target architecture file name, in XML format, that describes a custom target for the concurrent execution environment for the model, specified as a string giving the XML file name.

Example: `custom_arch.xml`

More About

- "Define a Custom Architecture File"

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` | `Simulink.architecture.find_system` | `Simulink.architecture.profile` | `Simulink.architecture.register` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.profile

Generate profile report for model configured for concurrent execution

Syntax

```
Simulink.architecture.profile(model)
Simulink.architecture.profile(model,numSamples)
```

Description

`Simulink.architecture.profile(model)` generates a profile report for a model configured for concurrent execution. Subsequent calls to the command for the same model name overwrite the existing profile report.

`Simulink.architecture.profile(model,numSamples)` specifies the number of samples to generate a profile report.

Examples

Generate profile report

Generate profile report for the model `sldemo_concurrent_execution`.

```
Simulink.architecture.profile('sldemo_concurrent_execution');
```

The command creates the file `sldemo_concurrent_execution_ProfileReport.html` in the current folder and opens it.

Generate profile report for 120 time steps

Generate profile report for the model `sldemo_concurrent_execution` with data for 120 time steps.

```
Simulink.architecture.profile('sldemo_concurrent_execution',120);
```

The command creates the file
`sldemo_concurrent_execution_ProfileReport.html` in the current folder.

Input Arguments

`model` — Model to profile

string

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 value determines the number of steps to collect data for in the profiled model.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

- "Profile and Evaluate on a Desktop"

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` | `Simulink.architecture.find_system` | `Simulink.architecture.get_param` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.register` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.register

Add custom target architecture to concurrent execution target architecture selector

Syntax

```
Simulink.architecture.register(CustomArchFile)
```

Description

`Simulink.architecture.register(CustomArchFile)` adds an XML-format custom target architecture file `CustomArchFile` to the concurrent execution target architecture selector. To access this selector, click the **Select** button on the Concurrent Execution pane of the Concurrent Execution dialog box.

Examples

Add custom target architecture

Add custom target architecture defined in the XML file `custom_arch.xml` to the concurrent execution target architecture selector. This example requires you to create a `custom_arch.xml` first.

```
sldemo_concurrent_execution;
Simulink.architecture.register('custom_arch.xml')
```

Input Arguments

`CustomArchFile` — Custom target architecture file

XML file

Custom target architecture file that describes a custom target for concurrent execution, specified as an XML file.

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` | `Simulink.architecture.find_system` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.set_param

Set architecture object properties

Syntax

```
Simulink.architecture.set_param(Object,ParamName,ParamValue)
```

Description

`Simulink.architecture.set_param(Object,ParamName,ParamValue)` sets the specified parameter of `Object` to the specified value. Parameter name and value strings are case sensitive.

Examples

Set software node name

Set the software node name from `CPU` to `MyCPUNewName`.

```
sldemo_concurrent_execution
Simulink.architecture.set_param([bdroot  '/CPU'],'Name','MyCPUNewName');
```

Change Periodic

Set Periodic trigger period to .02.

```
sldemo_concurrent_execution
Simulink.architecture.set_param([bdroot '/MyCPUNewName/Periodic'],'Period','.02')
```

Input Arguments

`Object` — Object whose parameter value to set

string

Object whose parameter value to set, specified as a string giving the object full path name. Possible objects are:

- Software node
- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

`ParamName` — Name of the parameter to set

string

Name of parameter whose value to set, specified as a string.

These are the possible parameters whose values you can set for each of the object types:

For software node:

- `'Name'` — Name of the software node (string).

For hardware node:

- `'Name'` — Name of the hardware node (string).
- `'ClockFrequency'` — Frequency of the hardware node clock (string).
- `'Color'` — Color of the trigger icon, specified as an RGB triplet (vector).

For a periodic trigger:

- `'Name'` — Name of the trigger (string).
- `'Period'` — Period of the trigger (string).
- `'Color'` — Color of the trigger icon, specified as an RGB triplet (vector).

For an aperiodic trigger:

- `'Name'` — Name of the trigger (string).
- `'Color'` — Color of the trigger icon, specified as an RGB triplet (vector).
- `'EventHandlerType'` — Trigger source for the interrupt-driven task (string). Possible values:
  - `'Event (Windows)'`
  - `'Posix Signal (Linux/VxWorks 6.x)'`

- `'SignalNumber'` — Signal number for the trigger (string). You can set this value only if `EventHandlerType` is set to `Event (Windows)`.
- `'EventName'` — Event name for the trigger (string). You can set this value only if `EventHandlerType` is set to `Posix Signal (Linux/VxWorks 6.x)`.

For task:

- `'Name'` — Name of the task (string).
- `'Period'` — Period of the task (string).
- `'Color'` — Color of the task icon, specified as an RGB triplet (vector).

Data Types: `char`

`ParamValue` — Value to set the parameter to

string | vector

Value to set the parameter to, specified as a string, scalar, or vector. The possible values depend on the parameter.

Example: `'MyCPUNewName'`

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` | `Simulink.architecture.find_system` | `Simulink.architecture.get_param` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.register`

Introduced in R2014a

Simulink.Block.getSampleTimes

Return sample time information for a block

Syntax

```
ts = Simulink.Block.getSampleTimes(block)
```

Input Arguments

block

Full name or handle of a Simulink block

Output Arguments

ts

The command returns `ts` which is a `1xn` array of Simulink.SampleTime objects associated with the model passed to Simulink.Block.getSampleTimes. Here `n` is the number of sample times associated with the block. The format of the returns is:

```
1xn Simulink.SampleTime
Package: Simulink
value: [1x2 double]
Description: [char string]
ColorRGBValue: [1x3 double]
Annotation: [char string]
OwnerBlock: [char string]
ComponentSampleTimes: [1x2 struct]
Methods
```

- `value` — A two-element array of doubles that contains the sample time period and offset
- `Description` — A character 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`

Introduced in R2009a

Simulink.BlockDiagram.addBusToVector

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*.

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`

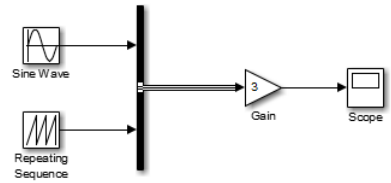
If *`reportOnly`* is `false`, and *`model`* contains any buses used as vectors, a cell array containing the path to each Bus to Vector block that was added to the model. Otherwise, *`BusToVectorBlocks`* is empty ([]).

Tip

You can eliminate warnings and errors about virtual buses used as muxes by using `Simulink.BlockDiagram.addBustoVector` to insert a Bus to Vector block into any virtual bus signal that is used as a mux. For additional information, see “Prevent Bus and Mux Mixtures”.

Examples

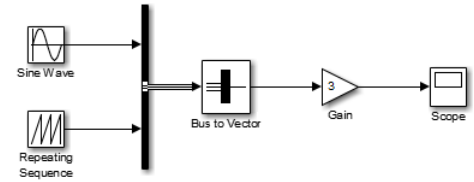
The following model simulates correctly, but the input to the Gain block is a bus, while the output is a vector. Thus the Gain block uses a block as a vector.



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:



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.

More About

- "Mux Signals"
- "Composite Signals"
- "Prevent Bus and Mux Mixtures"

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`

Introduced in R2007a

Simulink.BlockDiagram.buildRapidAcceleratorTarget

Build Rapid Accelerator target for model and return run-time parameter set

Syntax

```
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget(mdl)
```

Description

`rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget(mdl)` builds a Rapid Accelerator target for model, `mdl`, and returns run-time parameter set, `rtp`.

Input Arguments

`mdl`

Name or handle of a Simulink model

Output Arguments

`rtp`

Run-time parameter set that contains two elements:

Element	Description	
`modelChecksum`	1x4 vector that encodes the structure of the model.	
`parameters`	A structure of the tunable parameters in the model. This structure contains the following fields.	
	Field	Description
	`dataTypeName`	The data type name, for example, `double`.
	`dataTypeId`	Internal data type identifier for use by Simulink Coder.

Element	Description		
	`complex`	Complex type or real type specification. Value is `0` if real, `1` if complex.	
	`dtTransIdx`	Internal data type identifier for use by Simulink Coder.	
	`values`	All values associated with this entry in the `parameters` substructure.	
	`map`	Mapping structure information that correlates the values to the model tunable parameters. This structure contains the following fields.	
		Field	Description
		`Identifier`	Tunable parameter name.
		`ValueIndices`	Start and end indices into the values field, `[startIdx, endIdx]`.
		`Dimensions`	Dimension of this tunable parameter (matrices are generally stored in column-major format).

Examples

Build Rapid Accelerator Target for Model

In the MATLAB Command Window, type:

```
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget('f14')

### Building the rapid accelerator target for model: f14
### Successfully built the rapid accelerator target for model: f14

rtp =

    modelChecksum: [2.6812e+09 2.7198e+09 589261472 4.0180e+09]
       parameters: [1x1 struct]
```

More About

- “How Acceleration Modes Work”
- “Choosing a Simulation Mode”

- “Design Your Model for Effective Acceleration”

Introduced in R2012b

Simulink.BlockDiagram.copyContentsToSubSystem

Copy contents of block diagram to empty subsystem

Syntax

```
Simulink.BlockDiagram.copyContentsToSubSystem(bdiag, subsys)
```

Description

`Simulink.BlockDiagram.copyContentsToSubSystem(`*`bdiag`*`, `*`subsys`*`)` copies the contents of the block diagram *`bdiag`* to the subsystem *`subsys`*. The block diagram and subsystem must have already been loaded. The subsystem cannot be part of the block diagram.

The function affects only blocks, lines, and annotations; it does not affect nongraphical information such as configuration sets. You can use this function to convert a referenced model derived from an atomic subsystem into an atomic subsystem that is equivalent to the original subsystem.

This function cannot be used if the destination subsystem contains any blocks or signals. Other types of information can exist in the destination subsystem and are not affected by the function. Use `Simulink.SubSystem.deleteContents` if necessary to empty the subsystem before using `Simulink.BlockDiagram.copyContentsToSubSystem`.

Input Arguments

`bdiag`

Block diagram name or handle

`subsys`

Subsystem name or handle

Examples

Copy the contents of `vdp` to an empty subsystem named `vdp_subsystem` that is in the model named `new_model_with_vdp`:

```
open_system('vdp');
new_system('new_model_with_vdp')
open_system('new_model_with_vdp');
add_block('built-in/Subsystem', 'new_model_with_vdp/vdp_subsystem')
Simulink.BlockDiagram.copyContentsToSubSystem...
('vdp', 'new_model_with_vdp/vdp_subsystem')
```

More About

- "Systems and Subsystems"
- "Create a Subsystem"

See Also

`Simulink.BlockDiagram.deleteContents` |
`Simulink.SubSystem.convertToModelReference` |
`Simulink.SubSystem.copyContentsToBlockDiagram` |
`Simulink.SubSystem.deleteContents`

Introduced in R2007a

Simulink.BlockDiagram.createSubSystem

Create subsystem containing specified set of blocks

Syntax

```
Simulink.BlockDiagram.createSubSystem(blocks)
Simulink.BlockDiagram.createSubSystem()
```

Description

`Simulink.BlockDiagram.createSubSystem(blocks)` creates a new subsystem and moves the specified blocks into the subsystem. All of the specified blocks must originally reside at the top level of the model or in the same existing subsystem within the model.

If any of the blocks have unconnected input ports, the command creates input port blocks for each unconnected input port in the subsystem and connects the input port block to the unconnected input port. The command similarly creates and connects output port blocks for unconnected output ports on the specified blocks. If any of the specified blocks is an input port, the command creates an input port block in the parent system and connects it to the corresponding input port of the newly created subsystem. The command similarly creates and connects output port blocks for each of the specified blocks that is an output port block.

`Simulink.BlockDiagram.createSubSystem()` creates a new subsystem in the currently selected model and moves the currently selected blocks within the current model to the new subsystem.

Input Arguments

`blocks`

An array of block handles

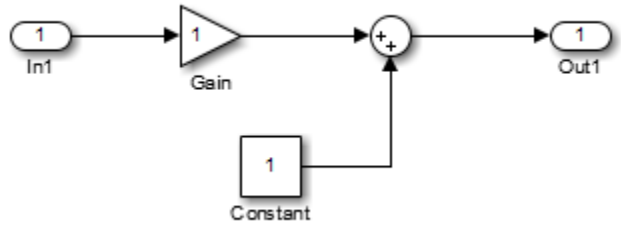
Default: [ ]

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
```

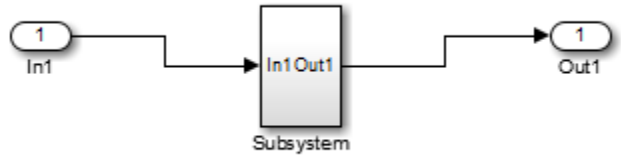
For example, suppose you create the following model and save it as `initial_model.slx`.



Executing

```
convert2subsys('initial_model');
```

converts this model to create a subsystem:



More About

- "Systems and Subsystems"
- "Create a Subsystem"

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.BlockDiagram.deleteContents` | `Simulink.SubSystem.convertToModelReference` | `Simulink.SubSystem.copyContentsToBlockDiagram`

Introduced in R2009a

Simulink.BlockDiagram.deleteContents

Delete contents of block diagram

Syntax

```
Simulink.BlockDiagram.deleteContents(bdiag)
```

Description

`Simulink.BlockDiagram.deleteContents(bdiag)` deletes the contents of the block diagram *`bdiag`*. The function affects only blocks, lines, and annotations. The block diagram must have already been loaded.

Input Arguments

`bdiag`

Block diagram name or handle

Examples

Delete the graphical content of an open block diagram named `f14`, including all subsystems:

```
Simulink.BlockDiagram.deleteContents('f14');
```

More About

- "Modeling"
- "Create a Subsystem"

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.SubSystem.convertToModelReference`

```
Simulink.SubSystem.copyContentsToBlockDiagram |
Simulink.SubSystem.deleteContents
```

Introduced in R2007a

Simulink.BlockDiagram.expandSubsystem

Expand subsystem contents to containing model level

Syntax

```
Simulink.BlockDiagram.expandSubsystem(block)
```

Description

`Simulink.BlockDiagram.expandSubsystem(block)` expands the contents of the subsystem for the specified Subsystem block. Subsystem expansion involves moving the contents of a virtual subsystem into the system that contains that subsystem.

You can expand virtual subsystems that are not masked, linked, or commented. For details, see "Subsystems That You Can Expand".

Input Arguments

`block`

A string that specifies one of the following:

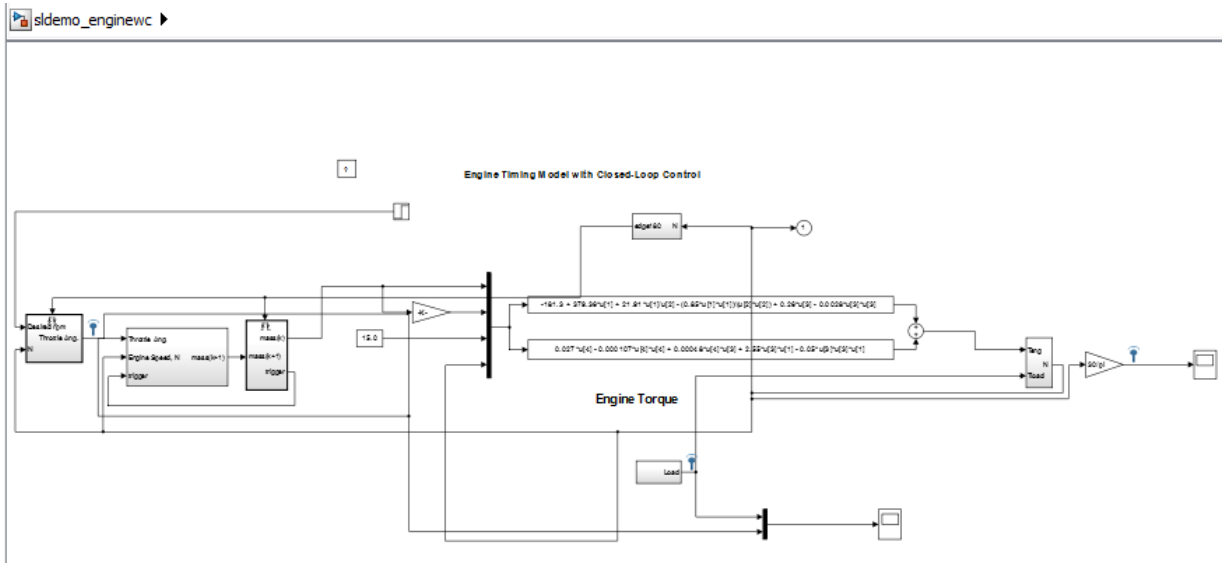
- The path to a subsystem block in a loaded model.
- The block handle of a subsystem block in a loaded model.
- `gcb` (the currently selected block)

Examples

The following function expands the Combustion subsystem.

```
open_system('sldemo_enginewc')
Simulink.BlockDiagram.expandSubsystem('sldemo_enginewc/Combustion')
```

The blocks and signals that were in the Combustion subsystem become part of the top-level model that contained the Combustion subsystem, replacing that Subsystem block.



More About

- "Expand Subsystem Contents"
- "Systems and Subsystems"

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.BlockDiagram.createSubSystem` | `Simulink.BlockDiagram.deleteContents` | `Simulink.SubSystem.convertToModelReference` | `Simulink.SubSystem.copyContentsToBlockDiagram`

Introduced in R2014a

Simulink.BlockDiagram.getChecksum

Return checksum of model

Syntax

```
[checksum,details] = Simulink.BlockDiagram.getChecksum('model')
```

Description

`[checksum,details] = Simulink.BlockDiagram.getChecksum('model')` returns the checksum of the specified model. Simulink software computes the checksum based on attributes of the model and the blocks the model contains.

One use of this command is to determine why the Accelerator mode in Simulink software regenerates code. For an example, see `slAccelDemoWhyRebuild`.

Note: `Simulink.BlockDiagram.getChecksum` compiles the specified model, if the model is not already in a compiled state.

This command accepts the argument *`model`*, which is the full name or handle of the model for which you are returning checksum data.

This command returns the following output:

- *`checksum`* — Array of four 32-bit integers that represents the model's 128-bit checksum.
- *`details`* — Structure of the form

  ```
  ContentsChecksum: [1x1 struct]
  InterfaceChecksum: [1x1 struct]
  ContentsChecksumItems: [nx1 struct]
  InterfaceChecksumItems: [mx1 struct]
  ```

  - `ContentsChecksum` — Structure of the following form that represents a checksum that provides information about all blocks in the model.

    ```
    Value: [4x1 uint32]
    ```

```
MarkedUnique: [bool]
```

- `Value` — Array of four 32-bit integers that represents the model's 128-bit checksum.
- `MarkedUnique` — True if any blocks in the model have a property that prevents code reuse.

- `InterfaceChecksum` — Structure of the following form that represents a checksum that provides information about the model.

```
Value: [4x1 uint32]
MarkedUnique: [bool]
```

  - `Value` — Array of four 32-bit integers that represents the model's 128-bit checksum.
  - `MarkedUnique` — Always true. Present for consistency with `ContentsChecksum` structure.

- `ContentsChecksumItems` and `InterfaceChecksumItems` — Structure arrays of the following form that contain information that Simulink software uses to compute the checksum for `ContentsChecksum` and `InterfaceChecksum`, respectively:

```
Handle: [char array]
Identifier: [char array]
Value: [type]
```

  - `Handle` — Object for which Simulink software added an item to the checksum. For a block, the handle is a full block path. For a block port, the handle is the full block path and a 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)
```

`Simulink.BlockDiagram.getChecksum` returns a checksum that depends on why and how you compiled the model. This function also compiles the model if it is not in a compiled state. The model compiles for:

- Simulation— if the simulation mode is Accelerator or you have not installed Simulink Coder
- Code generation— in all other cases

To compile the model before calling `Simulink.BlockDiagram.getChecksum`, use this command:

```
modelName([],[],[],'compile')
```

Note: The checksum that `Simulink.BlockDiagram.getChecksum` returns can vary from the checksum returned if you first compile the model at the command line (using the `model` command) before running `Simulink.BlockDiagram.getChecksum`.

Tip

The structural checksum reflects changes to the model that can affect the simulation results, including:

- Changing the solver type, for example from `Variable-step` to `Fixed-step`
- Adding or deleting blocks or connections between blocks
- Changing the values of nontunable block parameters, for example, the **Seed** parameter of the `Random Number` block
- Changing the number of inputs or outputs of blocks, even if the connectivity is vectorized
- Changing the number of states or the initial states in the model
- Selecting a different function in the `Trigonometric Function` block
- Changing signs used in a `Sum` block
- Adding a Target Language Compiler (TLC) file to inline an S-function

Examples of model changes that do not affect the structural checksum include:

- Changing the position of a block
- Changing the position of a line
- Resizing a block
- Adding, removing, or changing a model annotation

See Also

`Simulink.SubSystem.getChecksum` | `Simulink.getFileChecksum`

Introduced in R2006b

Simulink.BlockDiagram.getInitialState

Return initial state 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 block that owns 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`

Introduced in R2006b

Simulink.BlockDiagram.getSampleTimes

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`

Introduced in R2009a

Simulink.BlockDiagram.loadActiveConfigSet

Package: Simulink.BlockDiagram

Load, associate, and activate configuration set with model

Syntax

```
Simulink.BlockDiagram.loadActiveConfigSet(model, filename)
```

Description

`Simulink.BlockDiagram.loadActiveConfigSet(model, filename)` loads a configuration set, associates it with a model, and makes it the active configuration set. `model` is the name or handle of a model. `filename` is the name of the file (`.m` or `.mat`) that creates or contains a configuration set object to load. If you do not provide a file extension, it defaults to `.m`. If the file name is the same as a model name on the MATLAB path, the software cannot determine which file contains the configuration set object and displays an error message.

Examples

Save the configuration set from the `sldemo_counters` model to `my_config_set.m`.

```
% Open the sldemo_counters model
sldemo_counters
% Save the active configuration set to my_config_set.m
Simulink.BlockDiagram.saveActiveConfigSet('sldemo_counters', 'my_config_set.m')
```

Load the configuration set from `my_config_set.m`, associate it with the `vdp` model, and make it the active configuration set.

```
% Open the vdp model
vdp
% Load the configuration set from my_config_set.m, making it the active
% configuration set for vdp.
Simulink.BlockDiagram.loadActiveConfigSet('vdp', 'my_config_set.m')
```

More About

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`.
- "Load a Saved Configuration Set"

See Also

`Simulink.BlockDiagram.saveActiveConfigSet` | `Simulink.ConfigSet` | `attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

Introduced in R2010b

Simulink.BlockDiagram.propagateConfigSet

Propagate top model configuration reference to referenced models

Syntax

```
[isPropagated, convertedModels] =
Simulink.BlockDiagram.propagateConfigSet(model)
[isPropagated, convertedModels] =
Simulink.BlockDiagram.propagateConfigSet(model, 'include',
refModels)
[isPropagated, convertedModels] =
Simulink.BlockDiagram.propagateConfigSet(model, 'exclude',
refModels)
handle = Simulink.BlockDiagram.propagateConfigSet(model, 'gui')
```

Description

`[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model)` propagates the configuration reference for `model` to all referenced models. Execute the function from a writable folder.

`[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'include', refModels)` propagates the configuration reference for `model` to the models in the `refModels` list. Execute the function from a writable folder.

`[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'exclude', refModels)` propagates the configuration reference for `model` to all referenced models in the hierarchy except for the models in the `refModels` list. Execute the function from a writable folder.

`handle = Simulink.BlockDiagram.propagateConfigSet(model, 'gui')` opens the Configuration Reference Propagation dialog box.

Examples

Propagate a Configuration Reference to All Referenced Models

```
[isPropagated,convertedModels] = ...
Simulink.BlockDiagram.propagateConfigSet('sldemo_mdlref_depgraph')
```

Propagate a Configuration Reference to Listed Referenced Models

```
[isPropagated,convertedModels] = ...
Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','include',...
{'sldemo_mdlref_heater','sldemo_mdlref_house'})
```

Propagate a Configuration Reference to Referenced Models with Exclusions

```
[isPropagated,convertedModels] = ...
Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','exclude',...
{'sldemo_mdlref_heater','sldemo_mdlref_house'})
```

Open the Configuration Reference Propagation Dialog Box for a Model

```
Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','gui')
```

- “Share a Configuration Across Referenced Models”
- “Manage a Configuration Reference”

Input Arguments

`model` — Top model

string

Top model with configuration reference to propagate, specified as a string.

Example: ‘mdl’

`refModels` — Referenced models

cell array of strings

List of referenced models to be included or excluded in propagation, specified as a cell array of strings.

Example: {'mdl1','mdl2','mdl3'}

Output Arguments

`isPropagated` — Success of propagation

false (default) | true

Indication of whether configuration reference propagation is successful, specified as a Boolean.

`convertedModels` — Converted models

cell array of strings

List of converted model names, specified as a cell array of strings.

`handle` — Handle to dialog box

handle

Handle to the Configuration Reference Propagation dialog box. Returned when you specify the '`gui`' argument to the function.

See Also

`Simulink.BlockDiagram.restoreConfigSet`

Introduced in R2012b

Simulink.BlockDiagram.restoreConfigSet

Restore model configuration for converted models

Syntax

```
[isRestored, restoredModels] =
Simulink.BlockDiagram.restoreConfigSet(model)
```

Description

`[isRestored, restoredModels] = Simulink.BlockDiagram.restoreConfigSet(model)` restores the model configuration for all converted models after propagating a configuration reference from a top model to the referenced models. Execute the function from a writable folder.

Examples

Restore the Model Configuration for Converted Models

```
[isRestored,restoredModels] = ...
Simulink.BlockDiagram.propagateConfigSet('sldemo_mdlref_depgraph');
```

- “Share a Configuration Across Referenced Models”
- “Manage a Configuration Reference”

Input Arguments

`model` — Top model

string

Name of top model, specified as a string.

Example: ‘mdl’

Output Arguments

`isRestored` — Success of restoration

false (default) | true

Indication of whether configuration reference propagation is successful, specified as a Boolean.

`restoredModels` — Restored models

cell array of strings

List of restored model names, specified as a cell array of strings.

See Also

`Simulink.BlockDiagram.propagateConfigSet`

Introduced in R2012b

Simulink.BlockDiagram.saveActiveConfigSet

Package: Simulink.BlockDiagram

Save active configuration set of model

Syntax

```
Simulink.BlockDiagram.saveActiveConfigSet(model, filename)
```

Description

`Simulink.BlockDiagram.saveActiveConfigSet(model, filename)` saves the active configuration set of a model to a `.m` or `.mat` file. `model` is the name or handle of the model. `filename` is the name of the file to save the model configuration set. If you specify a `.m` extension, the file contains a function that creates a configuration set object. If you specify a `.mat` extension, the file contains a configuration set object. If you do not provide a file extension, the active configuration set is saved to a file with a `.m` extension. Do not specify `filename` to be the same as a model name; otherwise the software cannot determine which file contains the configuration set object when loading the file.

Examples

Save the configuration set from the `sldemo_counters` model to `my_config_set.m`.

```
% Open the sldemo_counters model
sldemo_counters
% Save the active configuration set to my_config_set.m
Simulink.BlockDiagram.saveActiveConfigSet('sldemo_counters', 'my_config_set.m')
```

More About

- "Save a Configuration Set"

See Also

`Simulink.BlockDiagram.loadActiveConfigSet` | `Simulink.ConfigSet` | `attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

Introduced in R2010b

Simulink.Bus.cellToObject

Convert cell array containing bus information to bus objects

Syntax

```
Simulink.Bus.cellToObject(busCells)
```

Description

`Simulink.Bus.cellToObject(busCells)` creates a set of bus objects in the MATLAB base workspace from a cell array of bus information. 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, Units,
Description}
```

More About

- "Composite Signals"

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`

Introduced before R2006a

Simulink.Bus.createMATLABStruct

Create MATLAB structures using same hierarchy and attributes as bus signals

Syntax

```
structFromBus = Simulink.Bus.createMATLABStruct(busSource)
structFromBus = Simulink.Bus.createMATLABStruct(busSource,
partialValues)
structFromBus = Simulink.Bus.createMATLABStruct(busSource,
partialValues,dims)

structsForBuses = Simulink.Bus.createMATLABStruct(portHandles)
structsForBuses = Simulink.Bus.createMATLABStruct(portHandles,
partialStructures)
structsForBuses = Simulink.Bus.createMATLABStruct(busObjectNames)
```

Description

`structFromBus = Simulink.Bus.createMATLABStruct(busSource)` creates a MATLAB structure that has the same hierarchy and attributes (such as type and dimension) as the bus specified in `busSource`. The resulting structure uses the ground values of the bus signal.

`structFromBus = Simulink.Bus.createMATLABStruct(busSource, partialValues)` creates a structure that uses specified values of `partialValues` instead of the corresponding ground values of the bus signal.

`structFromBus = Simulink.Bus.createMATLABStruct(busSource, partialValues,dims)` creates a structure that has the specified dimensions. To create a structure for an array of buses, use `dims`.

`structsForBuses = Simulink.Bus.createMATLABStruct(portHandles)` creates a cell array of structures for bus signal ports, specified with port handles. The resulting cell array of structures uses ground values. Use this syntax to create initialization structures for multiple bus ports. This syntax improves performance compared to using separate `Simulink.Bus.createMATLABStruct` calls to create the structures.

`structsForBuses = Simulink.Bus.createMATLABStruct(portHandles, partialStructures)` creates a cell array of structures that uses the specified values of `partialStructures` instead of the ground values.

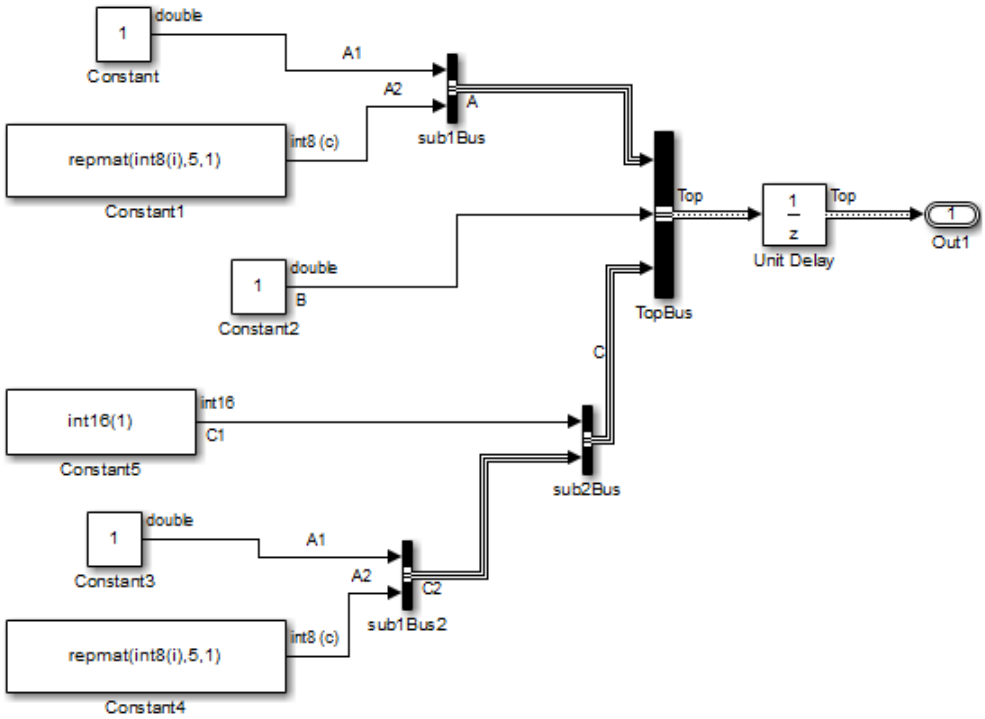
`structsForBuses = Simulink.Bus.createMATLABStruct(busObjectNames)` creates a cell array of structures based on the specified bus objects.

Examples

MATLAB Structure from Bus Object

Open a Simulink model and simulate it.

```
run([docroot '/toolbox/simulink/ug/examples/signals/busic_example.mdl']);
sim('busic_example')
```



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.

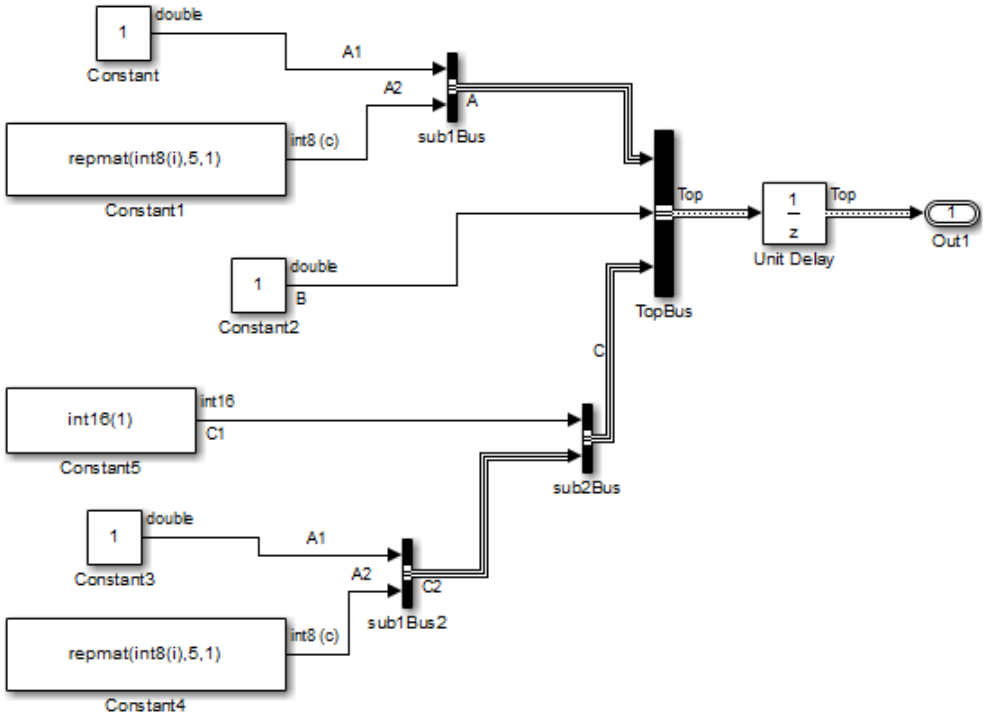
You can use `mStruct` as the initial condition structure for the Unit Delay block.

MATLAB Structure from Bus Port and Partial Structure

Create a MATLAB structure based on a port that connects to a bus signal. Use a partial structure to specify values for a subset of bus elements of the bus signal that connects to the port.

Open a Simulink model.

```
run([docroot '/toolbox/simulink/ug/examples/signals/busic_example.mdl']);
sim('busic_example')
```



Find the port handle for the Bus Creator block port that produces the `Top` bus signal. The `Outport` handle is the handle that you need.

```
ph = get_param('busic_example/TopBus','PortHandles')
```

```
ph =

      Inport: [143.0013 144.0013 145.0013]
     Outport: 34.0013
      Enable: []
     Trigger: []
       State: []
       LConn: []
       RConn: []
    Ifaction: []
```

Create a partial structure, which is a MATLAB structure that specifies values for a subset of bus elements for the bus signal created by the TopBus block.

```
PartialstructForK = struct('A',struct('A1',4),'B',3)

PartialstructForK =

    A: [1x1 struct]
    B: 3
```

Bus elements represented by structure fields `Top.B` and `Top.A` are at the same nesting level in the bus. You can use this partial structure to override the ground values for the `B` and `A` bus signal elements.

When you create a structure from a bus object or from a bus port, you can use a partial structure as an optional argument.

Create a MATLAB structure by using the port handle (`ph`) for the `TopBus` block. Override the ground values for the `A.A1` and `B` bus elements.

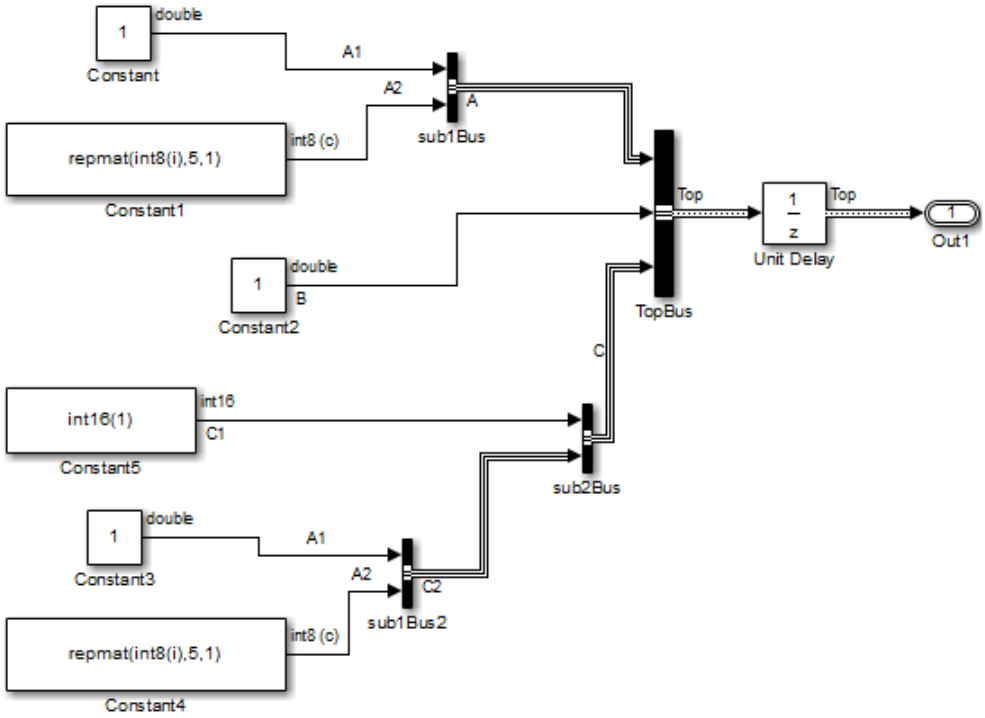
```
outPort = ph.Outport;
mStruct = Simulink.Bus.createMATLABStruct(outPort,PartialstructForK)

mStruct =

    A: [1x1 struct]
    B: 3
    C: [1x1 struct]
```

Initialize Signal Elements That Use a Data Type Other than `double`

Create a MATLAB structure for a bus whose signal elements use a data type other than `double`. Use a partial structure to specify initialization values for a subset of the elements. When you create the partial structure, match the data types of the fields with the data types of the corresponding elements.

Open a Simulink model.

```
run([docroot '/toolbox/simulink/ug/examples/signals/busic_example.mdl']);
sim('busic_example')
```



The `C1` signal element that the `Constant5` block produces uses the data type `int16`.

Find the port handle for the Bus Creator block port that produces the `Top` bus signal.

```
ph = get_param('busic_example/TopBus','PortHandles');
```

Create a partial structure that specifies values for a subset of the elements in the bus signal created by the TopBus block. To set the value of the `C.C1` field, use a typed expression. Match the data type in the expression with the data type of the signal element in the model (`int16`).

```
PartialstructForK = struct('B',3,'C',struct('C1',int16(5)));
```

Create a full structure by using the port handle (`ph`) for the `TopBus` block. Override the ground values for the `C.C1` and `B` elements.

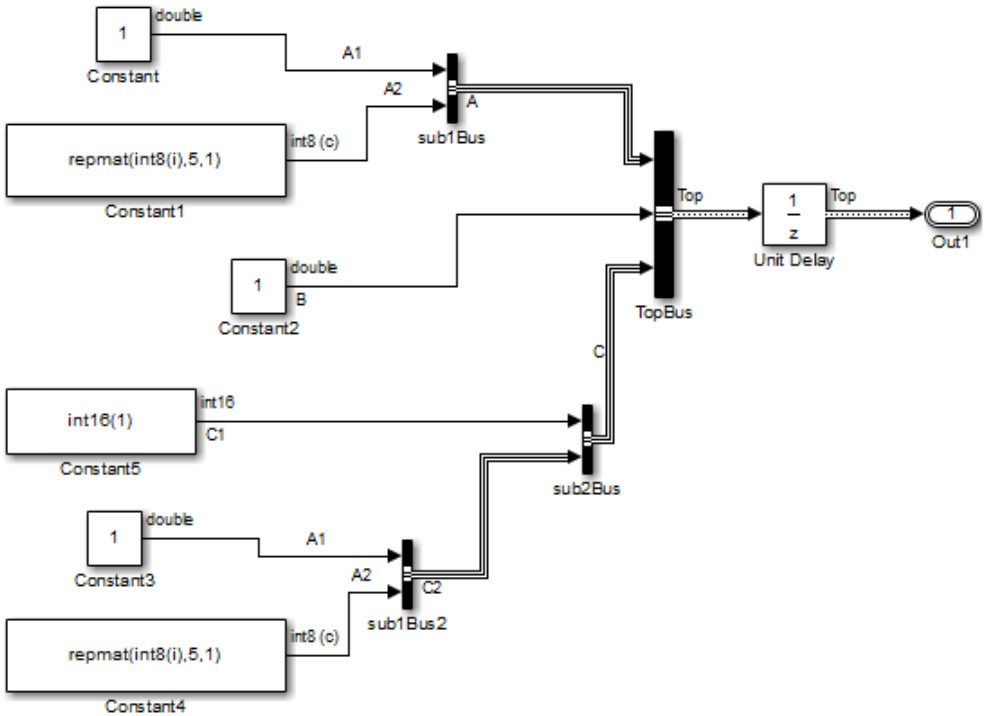
```
outPort = ph.Outport;
mStruct = Simulink.Bus.createMATLABStruct(outPort,PartialstructForK);
```

The field `C.C1` in the output structure continues to use the data type `int16`.

MATLAB Structure with Specified Dimensions

Open a Simulink model and simulate it.

```
run([docroot '/toolbox/simulink/ug/examples/signals/busic_example.mdl']);
sim('busic_example')
```



Create a partial structure, which is a MATLAB structure that specifies values for a subset of bus elements for the bus signal created by the TopBus block.

```
PartialStructForK = struct('A',struct('A1',4),'B',3)
```

```
PartialStructForK =

    A: [1x1 struct]
    B: 3
```

Create a MATLAB structure using the bus object `Top` (which the `busic_example` model loads), a partial structure, and dimensions for the resulting structure.

```
structFromBus = Simulink.Bus.createMATLABStruct...
     ('Top',PartialStructForK,[2 3])
```

```
structFromBus =

2x3 struct array with fields:

    A
    B
    C
```

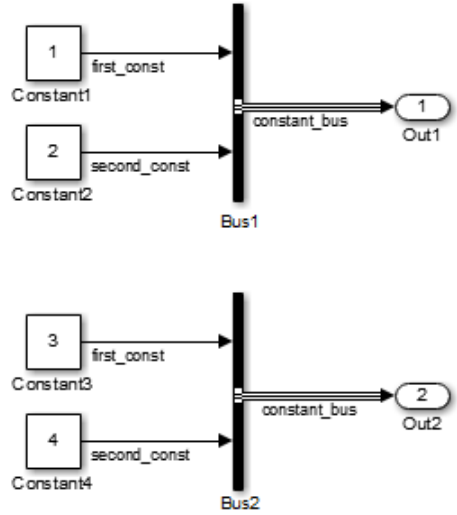
Close the system.

```
close_system('busic_example')
```

Cell Array of MATLAB Structures

Open a Simulink model and simulate it.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_two_outports_create_struct')))
sim('ex_two_outports_create_struct')
```



Find the port handles for the Bus Creator blocks Bus1 and Bus2.

```
ph_1 = get_param...
     ('ex_two_outports_create_struct/Bus Creator','PortHandles')
ph_2 = get_param...
     ('ex_two_outports_create_struct/Bus Creator1','PortHandles')
```

Create a MATLAB structure using an array of port handles.

```
mStruct = Simulink.Bus.createMATLABStruct...
     ([ph_1.Outport ph_2.Outport])
```

```
mStruct =

    [1x1 struct]
    [1x1 struct]
```

Close the system.

```
close_system('ex_two_outports_create_struct')
```

Input Arguments

`busSource` — Source representing a bus signal

a `Simulink.Bus` object | port handle

Source representing a bus signal to use for creating a MATLAB structure, specified as the name of a bus object or port handle.

- If you use the `dims` argument, then for `busSource`, use a bus object.
- For an array of buses signal, you cannot use a port handle.
- If you use a bus object name, then the bus object must be in the MATLAB base workspace. The data type for the bus object name is `char`.
- If you use a port handle, then the model must compile successfully before you use the `createMATLABStruct` method. The data type for the port handle is a `double`.

Example:

```
structFromBus = Simulink.Bus.createMATLABStruct('myBusObject');
structForPortHandle = Simulink.Bus.createMATLABStruct(port_handle_1);
```

`partialValues` — Values for a subset of leaf nodes of the resulting structure

partial structure | `[]`

Values for a subset of leaf nodes of the resulting structure, specified as a partial structure or empty array. Each field that you specify in a partial structure must match the data attributes of the corresponding bus element exactly. For details, see "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.

Each dimension element must be an integer that is greater than or equal to 1. If you specify `partialValues`, then each dimension element in `dims` must be greater than or equal to its corresponding dimension element in the partial structure.

To use ground values, use an empty matrix (`[]`) for `partialValues` .

Data Types: `double`

`portHandles` — Handles of bus signal ports

array

Handles of bus signal ports, specified as an array. If you use the `partialStructures` argument, then the number of port handles that you specify in `portHandles` must be the same as the number of partial structures.

Data Types: `double`

`partialStructures` — Partial structures

cell array

Partial structures specified as a cell array. The number of port handles that you specify in `portHandles` must be the same as the number of partial structures.

Data Types: `cell`

`busObjectNames` — Bus object names

cell array

Bus object names, specified as a cell array.

Data Types: `cell`

Output Arguments

`structFromBus` — Bus signal hierarchy and attributes

MATLAB structure

Bus signal hierarchy and attributes, returned as a MATLAB structure.

The dimensions of `structFromBus` depend on the input arguments:

- If you specify only `busSource`, then the dimension is 1.
- If you also specify `partialValues`, then the dimensions match the dimensions of `partialValues`.
- If you specify the `dims` argument, then the dimensions match the dimensions of `dims`.

`structsForBuses` — Structures having the same hierarchy and attributes as bus signals

cell array

Structures having the same hierarchy and attributes as bus signals, returned as a cell array of structures of data with same hierarchy and attributes as a bus signals that you specify with an array of port handles. The cell array of structures uses ground values of the bus signals.

The dimensions of `StructsForBuses` depend on the input arguments:

- If you specify only `portHandles`, then the dimension is 1.
- If you also specify `partialStructures`, then the dimensions match the dimensions of `partialStructures`.

Tips

- If you use the `Simulink.Bus.createMATLABStruct` function repeatedly for the same model (for example, in a loop in a script), you can improve performance by avoiding multiple model compilations. For improved speed, put the model in compile before using the function multiple times. For example, to put the `vdp` model in compile, use this command:

  ```
  [sys,x0,str,ts] = vdp([],[],[],'compile')
  ```

After you create the MATLAB structure, terminate the compile. For example:

```
vdp([],[],[],'term')
```

- You can use the Bus Editor to invoke the `Simulink.Bus.createMATLABStruct` function. Use one of these approaches:

  - Select the **File** > **Create a MATLAB structure** menu item.
  - Select the bus object for which you want to create a full MATLAB structure. Then, in the toolbar, click the **Create a MATLAB structure** button ().

  You can then edit the MATLAB structure in the MATLAB Editor and evaluate the code to create or update the values in this structure.

- You can use the `Simulink.Bus.createMATLABStruct` function to specify the initial value of the output of a referenced model. For details, see the "Referenced Model: Setting Initial Value for Bus Output" section of the Detailed Workflow for Managing Data with Model Reference example.

See Also

"Specify Initial Conditions for Bus Signals" | "Composite Signals" | `Bus to Vector` | `Bus Assignment` | `Bus Creator` | `Simulink.Bus` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createObject` | `Simulink.Bus.objectToCell` | `Simulink.Bus.save` | `Simulink.BusElement` | `Simulink.SimulationData.createStructOfTimeseries` |

Introduced in R2010a

Simulink.Bus.createObject

Create bus objects from blocks or MATLAB structures

Syntax

```
busInfo = Simulink.Bus.createObject(modelName, blks)
busInfo = Simulink.Bus.createObject(modelName,blks,fileName)
busInfo = Simulink.Bus.createObject(modelName,blks,fileName,format)
busInfo =
Simulink.Bus.createObject(structTimeseries,fileName,format)
busInfo = Simulink.Bus.createObject(structNumeric,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,Units,
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(*structTimeseries*,*fileName*,*format*) creates bus objects in the MATLAB workspace from a MATLAB structure of timeseries objects and optionally saves the bus objects in the specified file.

busInfo = Simulink.Bus.createObject(*structNumeric*,*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. The file name must be unique. If you omit this argument, the function save the created bus objects in a cell array, not in a file.

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'

`structTimeseries`

MATLAB timeseries structure variable used to create bus objects

`structNumeric`

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

Examples

Use Bus Creator Blocks to Create a Bus Object

Create a bus object from the Bus Creator block called Bus Creator2.

```
open_system('busdemo')
bus2Info = Simulink.Bus.createObject...
('busdemo', 'busdemo/Bus Creator2')
close_system('busdemo')
```

Create a bus object from two Bus Creator blocks, using block handles to specify the blocks. In the Simulink Editor, select the Bus Creator2 block and then the Bus Creator block and assign their block handles to variables. Use those variables in a vector specify the blocks to use for creating the bus object. This example also shows how to specify a file for saving the output (`busdemo_busobject`).

```
clear;
open_system('busdemo')
```

```
% Select the Bus Creator2 block
bc2 = gcbh;
% Select the Bus Creator block
bc1 = gcbh;
bus3Info = Simulink.Bus.createObject...
('busdemo', [bc2 bc1], 'busdemo_busobject')
 close_system('busdemo')
```

Use a Structure of Timeseries Objects to Create a Bus Object

Create a bus object from a MATLAB structure of timeseries objects that results from logged data for the COUNTERBUS bus signal.

```
model = 'sldemo_mdlref_bus';
open_system(model);
sim(model);
topOut
```

```
topOut =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'topOut'
    Total Elements: 3

  Elements:
    1: 'COUNTERBUS'
    2: 'OUTERDATA'
    3: 'INCREMENTBUS'

  -Use get or getElement to access elements by index or name.
  -Use addElement or setElement to add or modify elements.

  Methods, Superclasses
```

```
bus4Info = Simulink.Bus.createObject(topOut.get('COUNTERBUS').Values);
close_system(model);
```

Create a bus object from a MATLAB structure, independent of a model.

```
X = struct('a',1,'b',2)
```

```
bus3Info = Simulink.Bus.createObject(X)
```

More About

- "Composite Signals"

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

Introduced before R2006a

Simulink.Bus.objectToCell

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, Units,
Description}
```

More About

- "Composite Signals"

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

Introduced in R2007a

Simulink.Bus.save

Save bus objects in MATLAB file

Syntax

```
Simulink.Bus.save(fileName)
Simulink.Bus.save(fileName, format)
Simulink.Bus.save(fileName, format, busNames)
```

Description

`Simulink.Bus.save(`*`fileName`*`)` saves all bus objects (instances of `Simulink.Bus` class 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, Units,
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: {}

More About

- "Composite Signals"

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`

Introduced before R2006a

Simulink.createFromTemplate

Create model or project from template

Syntax

```
Simulink.createFromTemplate(templatename)
h = Simulink.createFromTemplate(templatename)
h = Simulink.createFromTemplate(templatename,Name,Value)
```

Description

`Simulink.createFromTemplate(templatename)` creates a model or a project from the template file specified by `templatename`.

`h = Simulink.createFromTemplate(templatename)` creates a model or a project from the template file and returns `h`, either a numeric model handle or a `simulinkproject` object.

`h = Simulink.createFromTemplate(templatename,Name,Value)` specifies additional options as one or more Name, Value pair arguments.

Examples

Create a Model From a Template

```
Simulink.createFromTemplate('simple_simulation.sltx')
```

Create a Project From a Template and Get the Handle

Create a project from a template, specify the name and root folder, and return the handle to the new project (a `simulinkproject` object) for manipulating it programmatically.

```
proj = Simulink.createFromTemplate('code_generation_example.sltx',Name,'myProject',Fold
```

- “Create Models and Open Existing Models”
- “Create a Template from a Model”

- “Using Templates to Create Standard Project Settings”

Input Arguments

`templatename` — Template file name
string

Template file name, specified as a string. If the template is not on the MATLAB path, specify the fully-qualified path to the template file and `*.sltx` extension.

Example:

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

`'Folder'` — Project root folder
string

Project root folder, if creating a new project, specified as a string.

Data Types: `char`

`'Name'` — New model or project name
string

New model or project name, specified as a string.

Data Types: `char`

Output Arguments

`h` — Handle
numeric handle | `simulinkproject`

Handle to the new model project, returned either as a numeric model handle or a `simulinkproject` object.

See Also

`Simulink.exportToTemplate` | `Simulink.findTemplates`

Introduced in R2016a

Simulink.data.assigninGlobal

Modify variable values in context of Simulink model

Syntax

```
Simulink.data.assigninGlobal(modelName,varName,varValue)
```

Description

`Simulink.data.assigninGlobal(modelName,varName,varValue)` assigns the value `varValue` to the variable or data dictionary entry `varName` in the context of the Simulink model `modelName`. `assigninGlobal` creates the variable or data dictionary entry if it does not already exist. The function operates in the Design Data section of the data dictionary that is linked to the target model or in the MATLAB base workspace if the target model is not linked to any data dictionary.

If the target model is linked to a data dictionary that references other dictionaries, `assigninGlobal` searches for `varName` in the entire dictionary hierarchy. If `assigninGlobal` does not find a matching entry, the function creates an entry in the dictionary that is linked to the target model.

Examples

Modify Variable in Model With or Without Data Dictionary

Create a variable `myNewVariable` with value `237` in the context of the Simulink model `vdp.slx`, which is not linked to any data dictionary. `myNewVariable` appears as a variable in the MATLAB base workspace.

```
Simulink.data.assigninGlobal('vdp','myNewVariable',237)
```

Create a variable `myNewEntry` with value `true` in the context of the Simulink model `sldemo_fuelsys_dd_controller.slx`, which is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd`. The entry `myNewEntry` appears in the Design Data section of the dictionary.

```
Simulink.data.assigninGlobal('sldemo_fuelsys_dd_controller',...
'myNewEntry',true)
```

Confirm the addition of `myNewEntry` to the data dictionary `sldemo_fuelsys_dd_controller.sldd` by viewing the dictionary in Model Explorer.

```
myDictionaryObj = Simulink.data.dictionary.open(...
'sldemo_fuelsys_dd_controller.sldd');
show(myDictionaryObj)
```

- "Store Data in Dictionary Programmatically"

Input Arguments

`modelName` — Name of target Simulink model

string

Name of target Simulink model, specified as a string.

Example: `'myTestModel'`

Data Types: `char`

`varName` — Name of target variable or data dictionary entry

string

Name of target variable or data dictionary entry, specified as a string.

Example: `'myTargetVariable'`

Data Types: `char`

`varValue` — Value to assign to variable or data dictionary entry

MATLAB expression

Value to assign to variable or data dictionary entry, specified as a MATLAB expression that returns any valid data type or data dictionary content.

Example: `27.5`

Example: `myBaseWorkspaceVariable`

Example: `Simulink.Parameter`

More About

Tips

- `assigninGlobal` helps you transition Simulink models to using data dictionaries. You can use the function to assign values to model variables before and after linking a model to a data dictionary.

- "What Is a Data Dictionary?"
- "Considerations before Migrating to Data Dictionary"

See Also

`Simulink.data.dictionary.open` | `Simulink.data.evalinGlobal` | `Simulink.data.existsInGlobal`

Introduced in R2015a

Simulink.data.dictionary.cleanupWorkerCache

Restore defaults after parallel simulation with data dictionary

Syntax

```
Simulink.data.dictionary.cleanupWorkerCache
```

Description

`Simulink.data.dictionary.cleanupWorkerCache` restores default settings after you have finished parallel simulation of a model that is linked to a data dictionary. Use this function in a `spmd` block, after you finish parallel simulation using `parfor` blocks, to restore default settings that were altered by the `Simulink.data.dictionary.setupWorkerCache` function.

During parallel simulation of a model that is linked to a data dictionary, you can allow each worker to access and modify the data in the dictionary independently of other workers. The function `Simulink.data.dictionary.setupWorkerCache` grants each worker a unique dictionary cache to allow independent access to the data, and the function `Simulink.data.dictionary.cleanupWorkerCache` restores cache settings to their default values.

You must have a Parallel Computing Toolbox™ license to perform parallel simulation using a `parfor` block.

Examples

Sweep Data Dictionary Parameter Using Parallel Simulation

To use parallel simulation to sweep a model parameter that is defined in a data dictionary, use this code as a template. Change the names and values of the model, data dictionary, and swept parameter to match your application.

You cannot use this code for parallel Rapid Accelerator Mode or Accelerator Mode simulation. For an example of parallel simulation using Rapid Accelerator Mode, see “Parallel Simulations Using Parfor: Parameter Sweep in Rapid Accelerator Mode”.

You must have a Parallel Computing Toolbox license to perform parallel simulation.

```
% For convenience, define names of model and data dictionary
model = 'myParamSweepMdl';
dd = 'myParamSweepDD.sldd';

% Define parameter sweeping values
ParamValues = [20 35 49 78 106 123 148 192 205 225];

% Grant each worker in the parallel pool an independent data dictionary
% so they can use the data without interference
spmd
    Simulink.data.dictionary.setupWorkerCache
end

% Determine the number of times to simulate
numberOfSims = length(ParamValues);

% Prepare a nondistributed array to contain simulation output
simOut = cell(1,numberOfSims);

parfor index = 1:numberOfSims
    % Create objects to interact with dictionary data
    % You must create these objects for every iteration of the parfor-loop
    dictObj = Simulink.data.dictionary.open(dd);
    sectObj = getSection(dictObj,'Design Data');
    entryObj = getEntry(sectObj,'SpeedVect');
    % Suppose SpeedVect is a Simulink.Parameter stored in the data dictionary

    % Modify the value of the Simulink.Parameter stored in the data dictionary
    temp = getValue(entryObj);
    temp.Value = ParamValues(index);
    setValue(entryObj,temp);

    % Simulate and store simulation output in the nondistributed array
    simOut{index} = sim(model);

    % Each worker must discard all changes to the data dictionary and
    % close the dictionary when finished with an interation of the parfor-loop
    discardChanges(dictObj);
    close(dictObj);
end

% Restore default settings that were changed by the function
```

```
% Simulink.data.dictionary.setupWorkerCache
% Prior to calling cleanupWorkerCache, close the model
bdclose(model)
spmd
    Simulink.data.dictionary.cleanupWorkerCache
end
```

Note: If data dictionaries are open, you cannot use the command `Simulink.data.dictionary.cleanupWorkerCache`. To identify open data dictionaries, use `Simulink.data.dictionary.getOpenDictionaryPaths`.

More About

- "What Is a Data Dictionary?"
- "Running Code on Parallel Pools"

See Also

`parfor` | `Simulink.data.dictionary.closeAll` | `Simulink.data.dictionary.getOpenDictionaryPaths` | `Simulink.data.dictionary.setupWorkerCache` | `spmd`

Introduced in R2015a

Simulink.data.dictionary.closeAll

Close all connections to all open data dictionaries

Syntax

```
Simulink.data.dictionary.closeAll
Simulink.data.dictionary.closeAll(dictFileName)
Simulink.data.dictionary.closeAll( ___ ,unsavedAction)
```

Description

`Simulink.data.dictionary.closeAll` attempts to close all connections to all data dictionaries that are open. For example, if you create objects, such as `Simulink.data.Dictionary`, that refer to a dictionary, that dictionary is open.

Some commands and functions, such as `Simulink.data.dictionary.cleanupWorkerCache`, cannot operate when dictionaries are open. It is a best practice to close each connection individually by using functions and methods such as the close method of a `Simulink.data.Dictionary` object. To find dictionaries that are open, use `Simulink.data.dictionary.getOpenDictionaryPaths`. However, you can use this function to close all connections to all dictionaries.

You can also use this function to close dictionaries in a shutdown script that is part of a Simulink Project.

`Simulink.data.dictionary.closeAll(dictFileName)` closes all connections to the dictionary named `dictFileName`. If you open multiple dictionaries that use this file name (for example, if the dictionaries have different file paths), the function closes all connections to all of the dictionaries.

You cannot specify `dictFileName` as a full file path such as `'C:\temp\myDict.sldd'`.

`Simulink.data.dictionary.closeAll( ___ ,unsavedAction)` closes all connections to the target dictionaries by discarding or saving unsaved changes. You can choose whether to save or discard all changes to all of the target dictionaries.

Examples

Close All Connections to All Open Dictionaries

Discard any unsaved changes. All of the entries in the dictionaries revert to the last saved state.

```
Simulink.data.dictionary.closeAll('-discard')
```

Close All Connections to Single Data Dictionary

Open multiple connections to a data dictionary, make a change, and close all of the connections by discarding the unsaved change.

At the command prompt, open a data dictionary by creating a `Simulink.data.Dictionary` object that refers to the dictionary.

```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd.sldd');
```

Display the dictionary in the Model Explorer

```
show(dictObj)
```

You now have two connections to this dictionary: The `Simulink.data.Dictionary` object and the Model Explorer.

Make a change to the dictionary by adding an entry.

```
dDataSectObj = getSection(dictObj,'Design Data');
addEntry(dDataSectObj,'myEntry',5.2);
```

The `Simulink.data.dictionary.Section` object `dDataSectObj` is a third connection to the dictionary.

Close the connections to the dictionary. Discard the unsaved change.

```
Simulink.data.dictionary.closeAll('sldemo_fuelsys_dd.sldd','-discard')
```

The dictionary no longer appears as a node in the **Model Hierarchy** pane of the Model Explorer. The `Simulink.data.Dictionary` object `dictObj` is disconnected from the dictionary. You cannot interact with the dictionary by using the `Simulink.data.dictionary.Section` object `dDataSectObj`.

Clear the objects that referred to the dictionary.

```
clear dictObj dDataSectObj
```

- "Store Data in Dictionary Programmatically"

Input Arguments

`dictFileName` — File name of target data dictionary or dictionaries

string

File name of target data dictionary or dictionaries, specified as a string. Use the file extension `sldd`.

Example: `'myDict.sldd'`

Data Types: `char`

`unsavedAction` — Action for unsaved changes

`'-discard' | '-save'`

Action for unsaved changes, specified as `'-discard'` (to discard changes) or `'-save'` (to save changes).

More About

Tips

A data dictionary is open if any of these conditions are true:

- The dictionary appears as a node in the **Model Hierarchy** pane of the Model Explorer. To close this connection to the dictionary, right-click the node in Model Explorer and select **Close**. Alternatively, use the hide method of a `Simulink.data.Dictionary` object.
- You created an object of any of these classes that refer to the dictionary:
  - `Simulink.data.Dictionary`
  - `Simulink.data.dictionary.Section`
  - `Simulink.data.dictionary.Entry`

To close these connections to the dictionary, use the close method of the `Simulink.data.Dictionary` object or clear the object. Clear the `Simulink.data.dictionary.Section` and `Simulink.data.dictionary.Entry` objects.

- A model that is linked to the dictionary is open. To close this connection to the dictionary, close the model.

See Also

Simulink.data.Dictionary | `Simulink.data.dictionary.cleanupWorkerCache` | `Simulink.data.dictionary.getOpenDictionaryPaths` | `Simulink.data.dictionary.setupWorkerCache`

Introduced in R2016a

Simulink.data.dictionary.create

Create new data dictionary and create `Simulink.data.Dictionary` object

Syntax

```
dictionaryObj = Simulink.data.dictionary.create(dictionaryFile)
```

Description

`dictionaryObj = Simulink.data.dictionary.create(dictionaryFile)` creates a data dictionary file in your current working folder or in a file path you can specify in `dictionaryFile`. The function returns a Simulink.data.Dictionary object representing the new data dictionary.

Examples

Create New Data Dictionary and Data Dictionary Object

Create a data dictionary `myNewDictionary.sldd` in your current working folder and a `Simulink.data.Dictionary` object representing the new data dictionary. Assign the object to the variable `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd')
```

```
myDictionaryObj =

  data dictionary with properties:

         DataSources: {0x1 cell}
    HasUnsavedChanges: 0
          NumberOfEntries: 0
```

- “Store Data in Dictionary Programmatically”

Input Arguments

`dictionaryFile` — Name of new data dictionary

string

Name of new data dictionary, specified as a string containing the file name and, optionally, path of the dictionary to create. If you do not specify a path, `Simulink.data.dictionary.create` creates the new data dictionary file in your working MATLAB folder. `Simulink.data.dictionary.create` also supports file paths specified relative to your working folder.

Example: `'myDictionary.sldd'`

Example: `'C:\Users\jsmith\myDictionary.sldd'`

Example: `'..\myOtherDictionary.sldd'`

Data Types: `char`

Output Arguments

`dictionaryObj` — Newly created data dictionary

`Simulink.data.Dictionary` object

Newly created data dictionary, returned as a `Simulink.data.Dictionary` object.

Alternatives

You can use the Simulink Editor to create a data dictionary and link it to a model. See "Migrate Single Model to Use Dictionary" for more information.

More About

- "What Is a Data Dictionary?"

See Also

Simulink.data.Dictionary | `Simulink.data.dictionary.open`

Introduced in R2015a

Simulink.data.dictionary.getOpenDictionaryPaths

Return file names and paths of open data dictionaries

Syntax

```
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths(
dictFileName)
```

Description

`openDDs = Simulink.data.dictionary.getOpenDictionaryPaths` returns the file names and paths of all data dictionaries that are open. For example, a data dictionary is open if you create objects, such as `Simulink.data.Dictionary`, that refer to the dictionary. If you open two or more dictionaries that have the same file name but different file paths, this function returns multiple file paths.

Before executing commands and functions that cannot operate when dictionaries are open, use this function to identify open dictionaries so that you can close them. For example, when you run parallel simulations as described in "Sweep Data Dictionary Parameter Using Parallel Simulation", this function helps you identify open dictionaries before executing the command `Simulink.data.dictionary.cleanupWorkerCache`.

`openDDs = Simulink.data.dictionary.getOpenDictionaryPaths(dictFileName)` returns the file paths of data dictionaries that have the file name `dictFileName`. If you open two or more dictionaries that have the same file name but different file paths, you can use this syntax to return all of the file paths.

Examples

Identify and Close All Open Data Dictionaries

Open, identify, and close a data dictionary. After you close the connections to the dictionary, you can use commands and functions, such as

`Simulink.data.dictionary.cleanupWorkerCache`, that cannot operate when dictionaries are open.

At the command prompt, open a data dictionary by creating a `Simulink.data.Dictionary` object that refers to the dictionary.

```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd.sldd');
```

Display the dictionary in the Model Explorer

```
show(dictObj)
```

Identify all of the dictionaries that are open.

```
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths;
```

The file path of the dictionary that you opened, `sldemo_fuelsys_dd.sldd`, appears in the cell array of strings `openDDs`.

Close the connection from the Model Explorer to the dictionary.

```
hide(dictObj)
```

The dictionary no longer appears as a node in the **Model Hierarchy** pane of the Model Explorer.

Close the connection from the `Simulink.data.Dictionary` object to the dictionary.

```
close(dictObj)
clear dictObj
```

- "Store Data in Dictionary Programmatically"

Input Arguments

`dictFileName` — File name of target data dictionary or dictionaries

string

File name of target data dictionary or dictionaries, specified as a string. Use the file extension `sldd`.

Example: `'myDict.sldd'`

Data Types: `char`

Output Arguments

`openDDs` — File names and paths of open data dictionaries

cell array of strings

File names and paths of open data dictionaries, returned as a cell array of strings.

More About

Tips

A data dictionary is open if any of these conditions are true:

- The dictionary appears as a node in the **Model Hierarchy** pane of the Model Explorer. To close this connection to the dictionary, right-click the node in Model Explorer and select **Close**. Alternatively, use the hide method of a `Simulink.data.Dictionary` object.
- You created an object of any of these classes that refer to the dictionary:
  - `Simulink.data.Dictionary`
  - `Simulink.data.dictionary.Section`
  - `Simulink.data.dictionary.Entry`

  To close these connections to the dictionary, use the close method of the `Simulink.data.Dictionary` object or clear the object. Clear the `Simulink.data.dictionary.Section` and `Simulink.data.dictionary.Entry` objects.
- A model that is linked to the dictionary is open. To close this connection to the dictionary, close the model.

See Also

Simulink.data.Dictionary | `Simulink.data.dictionary.cleanupWorkerCache` | `Simulink.data.dictionary.closeAll` | `Simulink.data.dictionary.setupWorkerCache`

Introduced in R2016a

Simulink.data.dictionary.open

Open data dictionary for editing

Syntax

```
dictionaryObj = Simulink.data.dictionary.open(dictionaryFile)
```

Description

`dictionaryObj = Simulink.data.dictionary.open(dictionaryFile)` opens the specified data dictionary and returns a Simulink.data.Dictionary object representing an existing data dictionary identified by its file name and, optionally, file path with `dictionaryFile`.

Make sure any dictionaries referenced by the target dictionary are on the MATLAB path.

Examples

Open Existing Data Dictionary

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')
```

```
myDictionaryObj =

  Dictionary with properties:

         DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 0
          NumberOfEntries: 4
```

- “Store Data in Dictionary Programmatically”

Input Arguments

`dictionaryFile` — Target data dictionary

string

Target data dictionary, specified as a string containing the file name and, optionally, path of the dictionary. If you do not specify a path, `Simulink.data.dictionary.open` searches the MATLAB path for the specified file. `Simulink.data.dictionary.open` also supports paths specified relative to the MATLAB working folder.

Example: `'myDictionary_ex_API.sldd'`

Example: `'C:\Users\jsmith\myDictionary_ex_API.sldd'`

Example: `'..\myOtherDictionary.sldd'`

Data Types: `char`

See Also

show | Simulink.data.Dictionary | `Simulink.data.dictionary.create`

Introduced in R2015a

Simulink.data.dictionary.setupWorkerCache

Enable parallel simulation with data dictionary

Syntax

```
Simulink.data.dictionary.setupWorkerCache
```

Description

`Simulink.data.dictionary.setupWorkerCache` prepares the workers in a parallel pool for simulating a model that is linked to a data dictionary. Use this function in a `spmd` block, prior to starting a `parfor` block, to provide the workers in a parallel pool a way to safely interact with a single data dictionary.

During parallel simulation of a model that is linked to a data dictionary, you can allow each worker to access and modify the data in the dictionary independently of other workers. `Simulink.data.dictionary.setupWorkerCache` temporarily provides each worker in the pool with its own data dictionary cache, allowing the workers to use the data in the dictionary without permanently changing it.

You must have a Parallel Computing Toolbox license to perform parallel simulation using a `parfor` block.

Examples

Sweep Data Dictionary Parameter Using Parallel Simulation

To use parallel simulation to sweep a model parameter that is defined in a data dictionary, use this code as a template. Change the names and values of the model, data dictionary, and swept parameter to match your application.

You cannot use this code for parallel Rapid Accelerator Mode or Accelerator Mode simulation. For an example of parallel simulation using Rapid Accelerator Mode, see “Parallel Simulations Using Parfor: Parameter Sweep in Rapid Accelerator Mode”.

You must have a Parallel Computing Toolbox license to perform parallel simulation.

```
% For convenience, define names of model and data dictionary
model = 'myParamSweepMdl';
dd = 'myParamSweepDD.sldd';

% Define parameter sweeping values
ParamValues = [20 35 49 78 106 123 148 192 205 225];

% Grant each worker in the parallel pool an independent data dictionary
% so they can use the data without interference
spmd
    Simulink.data.dictionary.setupWorkerCache
end

% Determine the number of times to simulate
numberOfSims = length(ParamValues);

% Prepare a nondistributed array to contain simulation output
simOut = cell(1,numberOfSims);

parfor index = 1:numberOfSims
    % Create objects to interact with dictionary data
    % You must create these objects for every iteration of the parfor-loop
    dictObj = Simulink.data.dictionary.open(dd);
    sectObj = getSection(dictObj,'Design Data');
    entryObj = getEntry(sectObj,'SpeedVect');
    % Suppose SpeedVect is a Simulink.Parameter stored in the data dictionary

    % Modify the value of the Simulink.Parameter stored in the data dictionary
    temp = getValue(entryObj);
    temp.Value = ParamValues(index);
    setValue(entryObj,temp);

    % Simulate and store simulation output in the nondistributed array
    simOut{index} = sim(model);

    % Each worker must discard all changes to the data dictionary and
    % close the dictionary when finished with an interation of the parfor-loop
    discardChanges(dictObj);
    close(dictObj);
end

% Restore default settings that were changed by the function
```

```
% Simulink.data.dictionary.setupWorkerCache
% Prior to calling cleanupWorkerCache, close the model
bdclose(model)
spmd
    Simulink.data.dictionary.cleanupWorkerCache
end
```

Note: If data dictionaries are open, you cannot use the command `Simulink.data.dictionary.cleanupWorkerCache`. To identify open data dictionaries, use `Simulink.data.dictionary.getOpenDictionaryPaths`.

More About

- "What Is a Data Dictionary?"
- "Running Code on Parallel Pools"

See Also

`parfor` | `Simulink.data.dictionary.cleanupWorkerCache` | `spmd`

Introduced in R2015a

Simulink.data.evalinGlobal

Evaluate MATLAB expression in context of Simulink model

Syntax

```
returnValue = Simulink.data.evalinGlobal(modelName,expression)
```

Description

`returnValue = Simulink.data.evalinGlobal(modelName,expression)` evaluates the MATLAB expression `expression` in the context of the Simulink model `modelName` and returns the values returned by `expression`. `evalinGlobal` evaluates `expression` in the Design Data section of the data dictionary that is linked to the target model or in the MATLAB base workspace if the target model is not linked to any data dictionary.

Examples

Evaluate MATLAB Expression in Model With or Without Data Dictionary

Evaluate the MATLAB expression `myNewVariable = 237;` in the context of the model `vdp`, which is not linked to any data dictionary. `myNewVariable` appears as a variable in the MATLAB base workspace.

```
Simulink.data.evalinGlobal('vdp','myNewVariable = 237;')
```

Evaluate the MATLAB expression `myNewEntry = true;` in the context of the model `sldemo_fuelsys_dd_controller`, which is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd`. `myNewEntry` appears as an entry in the Design Data section of the dictionary.

```
Simulink.data.evalinGlobal('sldemo_fuelsys_dd_controller',...
'myNewEntry = true;')
```

Confirm the creation of the entry `myNewEntry` in the data dictionary `sldemo_fuelsys_dd_controller.sldd` by viewing the dictionary in Model Explorer.

```
myDictionaryObj = Simulink.data.dictionary.open(...
'sldemo_fuelsys_dd_controller.sldd');
show(myDictionaryObj)
```

- “Store Data in Dictionary Programmatically”

Input Arguments

`modelName` — Name of target Simulink model

string

Name of target Simulink model, specified as a string.

Example: `'myTestModel'`

Data Types: `char`

`expression` — MATLAB expression to evaluate

string

MATLAB expression to evaluate, specified as a string.

Example: `'a = 5.3'`

Example: `'whos'`

Example: `'CurrentSpeed.Value = 290.73'`

Data Types: `char`

Output Arguments

`returnValue` — Value returned by specified expression

valid entry or variable value

Value returned by the specified MATLAB expression.

More About

Tips

- `evalinGlobal` helps you transition Simulink models to the use of data dictionaries. You can use the function to manipulate model variables before and after linking a model to a data dictionary.

See Also

evalin | `Simulink.data.assigninGlobal` | `Simulink.data.existsInGlobal`

Introduced in R2015a

Simulink.data.existsInGlobal

Check existence of variable in context of Simulink model

Syntax

```
varExists = Simulink.data.existsInGlobal(modelName,varName)
```

Description

`varExists = Simulink.data.existsInGlobal(modelName,varName)` returns an indication of the existence of a variable or data dictionary entry `varName` in the context of the Simulink model `modelName`. `Simulink.data.existsInGlobal` searches the Design Data section of the data dictionary that is linked to the target model or the MATLAB base workspace if the target model is not linked to any data dictionary.

Examples

Determine Existence of Variable in Model With or Without Data Dictionary

Determine the existence of a variable `PressVect` in the context of the Simulink model `vdp.slx`, which is not linked to any data dictionary.

```
Simulink.data.existsInGlobal('vdp','PressVect')
```

```
ans =

     0
```

Because `vdp.slx` is not linked to any data dictionary, `existsInGlobal` searches only in the MATLAB base workspace for `PressVect`.

Determine the existence of a variable `PressVect` in the context of the Simulink model `sldemo_fuelsys_dd_controller.slx`, which is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd`.

```
Simulink.data.existsInGlobal('sldemo_fuelsys_dd_controller','PressVect')
```

```
ans =
```

```
1
```

Because `sldemo_fuelsys_dd_controller.slx` is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd` , `existsInGlobal` searches for `PressVect` only in the Design Data section of the dictionary.

- "Store Data in Dictionary Programmatically"

Input Arguments

`modelName` — Name of target Simulink model

string

Name of target Simulink model, specified as a string.

Example: `'myTestModel'`

Data Types: `char`

`varName` — Name of target variable or data dictionary entry

string

Name of target variable or data dictionary entry, specified as a string.

Example: `'myTargetVariable'`

Data Types: `char`

Output Arguments

`varExists` — Indication of existence of target variable or data dictionary entry

`1 | 0`

Indication of existence of target variable or data dictionary entry, returned as `1` to indicate existence or `0` to indicate absence.

Alternatives

You can use Model Explorer to search a data dictionary or any workspace for entries or variables.

More About

Tips

- `existsInGlobal` helps you transition Simulink models to the use of data dictionaries. You can use the function to find model variables before and after linking a model to a data dictionary.

See Also

exist | `Simulink.data.assigninGlobal` | `Simulink.data.evalinGlobal`

Introduced in R2015a

Simulink.data.getEnumTypeInfo

Get information about enumerated data type

Syntax

```
information = Simulink.data.getEnumTypeInfo(enumTypeName,
infoRequest)
```

Description

`information = Simulink.data.getEnumTypeInfo(enumTypeName, infoRequest)` returns information about an enumerated data type `enumTypeName`.

Use this function only to return information about an enumerated data type. To customize an enumerated data type, for example, by specifying a default enumeration member or by controlling the scope of the type definition in generated code, see "Customize Simulink Enumeration".

Examples

Return Default Value of Enumerated Data Type

Get the default enumeration member of an enumerated data type `LEDcolor`. Suppose `LEDcolor` defines two enumeration members, `GREEN` and `RED`, and uses `GREEN` as the default member.

```
Simulink.data.getEnumTypeInfo('LEDcolor','DefaultValue')
```

```
ans =

    GREEN
```

Get Scope of Enumerated Data Type Definition in Generated Code

For an enumerated data type `LEDcolor`, find out if generated code exports or imports the definition of the type to or from a header file.

```
Simulink.data.getEnumTypeInfo('LEDcolor','DataScope')
Simulink.data.getEnumTypeInfo('LEDcolor','HeaderFile')

ans =

Auto


ans =

     ''
```

Because `DataScope` is `'Auto'` and `HeaderFile` is empty, generated code defines the enumerated data type `LEDcolor` in the header file *`model`*`_types.h` where *`model`* is the name of the model used to generate code.

- "Customize Simulink Enumeration"

Input Arguments

`enumTypeName` — Name of target enumerated data type

string

Name of the target enumerated data type, specified as a string.

Example: `'myFirstEnumType'`

Data Types: `char`

`infoRequest` — Information to return

valid string option

Information to return, specified as one of the string options in the table.

Specified value	Information returned	Example return value
`'DefaultValue'`	The default enumeration member, returned as an instance of the enumerated data type.	`enumMember1`
`'Description'`	The custom description of this data type, returned as a string. Returns an empty string if a description was not specified for the type.	`'My first enum type.'`

Specified value	Information returned	Example return value
`'HeaderFile'`	The name of the custom header file that defines the data type in generated code, returned as a string. Returns an empty string if a header file was not specified for the type.	`'myEnumType.h'`
`'DataScope'`	Indication whether generated code imports or exports the definition of the data type. A return value of `'Auto'` indicates generated code defines the type in the header file *`model`*`_types.h` or imports the definition from the header file identified by `HeaderFile`. A return value of `'Exported'` or `'Imported'` indicates generated code exports or imports the definition to or from the header file identified by `HeaderFile`.	`'Exported'`
`'StorageType'`	The integer data type used by generated code to store the numeric values of the enumeration members, returned as a string. Returns `'int'` if you did not specify a storage type for the enumerated type, in which case generated code uses the native integer type of the hardware target.	`'int32'`
`'AddClassNameToEnumNames'`	Indication whether generated code prefixes the names of enumeration members with the name of the data type. Returned as `true` or `false`.	`true`

More About

- "Simulink Enumerations"

See Also

`Simulink.defineIntEnumType`

Introduced in R2014b

Simulink.defineIntEnumType

Define enumerated data type

Syntax

```
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues)
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues,
'Description', ClassDesc)
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues,
'DefaultValue', DefValue)
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues,
'DataScope', ScopeSelection)
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues,
'HeaderFile', FileName)
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues,
'AddClassNameToEnumNames', Flag)
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues,
'StorageType', DataType)
```

Description

`Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues)` defines an enumeration named *`ClassName`* with enumeration values specified with *`CellOfEnums`* and underlying numeric values specified by *`IntValues`*.

`Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues, 'Description', ClassDesc)` defines the enumeration with a description (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, 'DataScope', ScopeSelection)` specifies whether the data type definition should be imported from, or exported to, a header file during code generation.

`Simulink.defineIntEnumType(`*`ClassName`*`, `*`CellOfEnums`*`, `*`IntValues`*`, 'HeaderFile', `*`FileName`*`)` specifies the name of a header file containing the enumeration class definition for use in code generated from a model.

`Simulink.defineIntEnumType(`*`ClassName`*`, `*`CellOfEnums`*`, `*`IntValues`*`, 'AddClassNameToEnumNames', `*`Flag`*`)` specifies whether the code generator applies the class name as a prefix to the enumeration values that you specify for *`CellOfEnums`*. For *`Flag`*, specify `true` or `false`. For example, if you specify `true`, the code generator would use `BasicColors.Red` instead of `Red` to represent an enumerated value.

`Simulink.defineIntEnumType(`*`ClassName`*`, `*`CellOfEnums`*`, `*`IntValues`*`, 'StorageType', `*`DataType`*`)` specifies the data type used to store the enumerations' underlying integer values in code generated from a model.

Input Arguments

`ClassName`

The name of the enumerated data type.

`CellOfEnums`

A cell array of 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.

By default, the generated `#include` directive uses the preprocessor delimiter " instead of < and >. To generate the directive `#include <myTypes.h>`, specify `FileName` as `'<myTypes.h>'`.

`'DataScope', 'Auto' | 'Exported' | 'Imported'`

Specifies whether the data type definition should be imported from, or exported to, a header file during code generation.

Value	Action
`Auto` (default)	If no value is specified for `Headerfile`, export the type definition to *`model`*`_types.h`, where *`model`* is the model name. If a value is specified for `Headerfile`, import the data type definition from the specified header file.
`Exported`	Export the data type definition to a header file. If no value is specified for `Headerfile`, the header file name defaults to *`type`*`.h`, where *`type`* is the data type name.
`Imported`	Import the data type definition from a header file. If no value is specified for `Headerfile`, the header file name defaults to *`type`*`.h`, where *`type`* is the data type name.

`'AddClassNameToEnumNames', Flag`

A logical flag that specifies whether code generator applies the class name as a prefix to the enumerations.

`'StorageType', DataType`

Specifies a string that identifies the data type used to store the enumerations' underlying integer values in generated code. The following data types are supported: `'int8'`, `'int16'`, `'int32'`, `'uint8'`, or `'uint16'`.

Examples

Assume an external data dictionary includes the following enumeration:

```
BasicColors.Red(0), BasicColors.Yellow(1), BasicColors.Blue(2)
```

Import the enumeration class definition into the MATLAB workspace while specifying `int16` as the underlying integer data type for generated code:

```
Simulink.defineIntEnumType('BasicColors', ...
 {'Red', 'Yellow', 'Blue'}, ...
 [0;1;2], ...
 'Description', 'Basic colors', ...
 'DefaultValue', 'Blue', ...
 'HeaderFile', 'mybasiccolors.h', ...
 'DataScope', 'Exported', ...
 'AddClassNameToEnumNames', true, ...
 'StorageType', 'int16');
```

More About

- "Import Enumerations Defined Externally to MATLAB"
- "Define Simulink Enumerations"

See Also

`enumeration`

Introduced in R2010b

Simulink.exportToTemplate

Create template from model or project

Syntax

```
templatefile = Simulink.exportToTemplate(obj,templatename)
templatefile = Simulink.exportToTemplate(obj,templatename,
Name,Value)
```

Description

`templatefile = Simulink.exportToTemplate(obj,templatename)` creates a template file (*`templatename`*`.sltx`) from a model or project specified by `obj`.

If you have project templates created in R2014a or earlier (`.zip` files), use `Simulink.exportToTemplate` to upgrade them to `.sltx` files, then you can use them in the start page.

`templatefile = Simulink.exportToTemplate(obj,templatename, Name,Value)` specifies additional template options as one or more Name, Value pair arguments.

Examples

Create a Template From a Model

Open the `vdp` model and create a template from it.

```
vdp
myvdptemplate = Simulink.exportToTemplate(bdroot,'vdptemplate')
```

Create a Template From a Model and Specify Description

Open the `vdp` model and create a template from it, specifying a description.

```
vdp
```

```
myvdptemplate = Simulink.exportToTemplate(bdroot,'vdptemplate','Description','Use this
```

- “Create Models and Open Existing Models”
- “Create a Template from a Model”
- “Using Templates to Create Standard Project Settings”

Input Arguments

`obj` — Model, library, or project

string | numeric handle | `slproject.ProjectManager`

Model, library, or project, specified by name or numeric handle, or a `slproject.ProjectManager` object returned by the `simulinkproject` function.

Data Types: `double` | `char`

`templatename` — Template file name

string

Template file name, specified as a string that can optionally include the fully-qualified path to a template file and `*.sltx`) extension.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Title,'My Project Template'`

`'Group'` — Group of template

string

Group of template, specified as a string. On the Start Page, templates are shown under group headings.

Example: `'Simscape'`

Data Types: `char`

`'Author'` — Author of template
string

Author of template, specified as a string.

Data Types: `char`

`'Description'` — Description of template
string

Description of template, specified as a string.

Data Types: `char`

`'ThumbnailFile'` — Thumbnail image file name
string

Thumbnail image file name, specified as a string.

Data Types: `char`

`'Title'` — Title of model or project template
string

Title of template, specified as a string. On the Start Page, the templates titles are shown on the tiles. The title can be different from the file name, and you can use any characters in the title. The default value is the name of the model or project.

Example: `'My Project Template'`

Data Types: `char`

Output Arguments

`templatefile` — Template file
string

Template file, returned as *`templatename`*`.sltx` file.

See Also

`Simulink.createFromTemplate` | `Simulink.findTemplates`

Introduced in R2016a

Simulink.exportToVersion

Export model or library for use in previous version of Simulink

Syntax

```
exported_file = Simulink.exportToVersion(modelname,target_filename,
version)
exported_file = Simulink.exportToVersion(modelname,target_filename,
version,Name,Value)
```

Description

`exported_file = Simulink.exportToVersion(modelname,target_filename, version)` exports the model or library `modelname` to a file named `target_filename` in a format that the specified previous Simulink `version` can load.

If the system contains functionality not supported by the specified Simulink software version, the command removes the functionality and replaces any unsupported blocks with empty masked subsystem blocks colored yellow. As a result, the converted system may generate different results.

The `save_system` `ExportToVersion` option is a legacy option for this functionality that is also supported.

`exported_file = Simulink.exportToVersion(modelname,target_filename, version,Name,Value)` specifies additional options as one or more Name, Value pair arguments.

Examples

Export a Model to a Previous Version

Get the current top-level system and export it.

```
Simulink.exportToVersion(bdroot,'mymodel.slx','R2014b');
```

Export a Model to a Previous Version and Break Links

Get the current top-level system and export it, replacing links to library blocks with copies of the library blocks in the saved file.

```
Simulink.exportToVersion(bdroot,'mymodel.slx','R2014b','BreakUserLinks',true);
```

- "Save a Model"

Input Arguments

`modelname` — Model to export

string

Model to export, specified as a string, without any file extension. The model must be loaded and unmodified. The target file must not be the same as the model file.

Data Types: `char`

`target_filename` — Exported file name

string

Exported file name, specified as a string. The target file must not be the same as the model file.

Example: `'mymodel.slx'`

Data Types: `char`

`version` — MATLAB release name

`'R14' | 'R14SP1' | 'R14SP2' | 'R14SP3' | 'R2006A' | 'R2006B' | 'R2007A' | 'R2007B' | 'R2008A' | 'R2008B' | 'R2009A' | 'R2009B' | 'R2010A' | 'R2010B' | 'R2011A' | 'R2011B' | 'R2012A' | 'R2012A_MDL' | 'R2012A_SLX' | 'R2012B' | 'R2012B_MDL' | 'R2012B_SLX' | 'R2013A' | 'R2013A_MDL' | 'R2013A_SLX' | 'R2013B' | 'R2013B_MDL' | 'R2013B_SLX' | 'R2014A' | 'R2014A_MDL' | 'R2014A_SLX' | 'R2014B' | 'R2014B_MDL' | 'R2014B_SLX' | 'R2015A' | 'R2015A_MDL' | 'R2015A_SLX' | 'R2015B' | 'R2015B_MDL' | 'R2015B_SLX'`

MATLAB release name, specified as a string, which specifies a previous Simulink version. `Simulink.exportToVersion` exports the system to a format that the specified

previous Simulink version can load. You cannot export to your current version. These version names are not case sensitive.

To export to Release 2012a and later, you can specify model file format as SLX or MDL. If you do not specify a format, you export your default model file format.

If you use the Export to Previous Version dialog box instead of `Simulink.exportToVersion`, then the **Save as type** list supports 7 years of previous releases.

Example: `'R2015B'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

`'AllowPrompt'` — Allow prompt or message dialog box

`false` (default) | `true` | `'on'` | `'off'`

Allow prompt or message dialog box, specified by a logical value that indicates whether to display any output prompt or message in a dialog box or only messages at the command line. For example, prompts to make files writable, or messages about exported versions. If you want to allow prompts, then set to `true`. or `on`.

`'BreakUserLinks'` — Break user-defined links

`false` (default) | `true` | `'on'` | `'off'`

Break user-defined links, specified by a logical value that indicates whether the function replaces links to user-defined library blocks with copies of the library blocks in the saved file.

`'BreakAllLinks'` — Break all library links

`false` (default) | `true` | `'on'` | `'off'`

Break all library links, specified by a 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.

Note The `'BreakAllLinks'` option can result in compatibility issues when upgrading to newer versions of Simulink software. For example:

- Any masks on top of library links to Simulink S-functions will not upgrade to the new version of the S-function.
- Any library links to masked subsystems in a Simulink library will not upgrade to the new subsystem behavior.
- Any broken links prevent the automatic library forwarding mechanism from upgrading the link.
  If you have saved a model with broken links to builtin libraries, use the Upgrade Advisor to scan the model for out-of-date blocks and upgrade the Simulink blocks to their current versions.

Output Arguments

`exported_file` — Exported file

string

Exported file, returned in the format that the specified previous Simulink version can load.

See Also

`save_system`

Introduced in R2016a

Simulink.findTemplates

Find model or project templates with specified properties

Syntax

```
filename = Simulink.findTemplates(templatename)
filename = Simulink.findTemplates(templatename,Name,Value)
[filename,info] = Simulink.findTemplates(templatename)
```

Description

`filename = Simulink.findTemplates(templatename)` returns the names and `TemplateInfo` objects for all matching templates that include `templatename`.

`filename = Simulink.findTemplates(templatename,Name,Value)` also specifies additional template properties as one or more Name, Value pair arguments.

`[filename,info] = Simulink.findTemplates(templatename)` returns the names and `TemplateInfo` objects for all matching templates.

Examples

Find a Particular Template

Get the full path to the default model template.

```
filename = Simulink.findTemplates('factory_default_model');
```

Find All Templates With Specified Folders or Authors

Get all templates inside folders called `work`.

```
filename = Simulink.findTemplates('work/')
```

Get all templates for which the `Author` property includes the string `Smith`.

```
filename = Simulink.findTemplates('*','Author','Smith')
```

Find All DSP Templates and Get TemplateInfo Objects

Get the paths to all DSP model templates, and `sltemplate.TemplateInfo` objects for each of them.

```
[filename,info] = Simulink.findTemplates('dsp*','Type','Model');
```

- "Create Models and Open Existing Models"
- "Create a Template from a Model"
- "Using Templates to Create Standard Project Settings"

Input Arguments

`templatename` — Template name

string

Template name, specified as a string containing a portion of a file name, which can contain the wildcard asterisk character "*".

Example:

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

You can specify regular expressions for any of the Value strings, e.g., including the wildcard asterisk character "*".

Example: `'Author','*son'`

`'Type'` — Model, library, or project

`'Model' | 'Library' | 'Project'`

Model, library, or project template type, specified as a string for model, library, or project.

Example: `'Simscape'`

Data Types: `char`

`'Title'` — Title of template

string

Title of template, specified as a string.

Example: `'Simscape'`

Data Types: `char`

`'Group'` — Group of template

string

Group of template, specified as a string. On the Start Page, templates are shown under group headings.

Example: `'Simscape'`

Data Types: `char`

`'Author'` — Author of template

string

Author of template, specified as a string.

Data Types: `char`

`'Description'` — Description of template

string

Description of template, specified as a string.

Data Types: `char`

Output Arguments

`filename` — Template name

string | cell array of strings

Template names of matching templates, returned as strings.

`info` — Template information

template info objects | array of template info objects

Template information of matching templates, returned as `sltemplate.TemplateInfo` objects.

See Also

`Simulink.createFromTemplate` | `Simulink.exportToTemplate`

Introduced in R2016a

Simulink.findVars

Find variables in models and blocks

Syntax

```
[variables] = Simulink.findVars(context)
[variables] = Simulink.findVars(context,variablefilter)
[variables] = Simulink.findVars( ___ ,Name,Value)
```

Description

`[variables] = Simulink.findVars(context)` finds and returns variables that are used in the blocks and models specified by `context`, including subsystems and referenced models. The function returns an empty vector if `context` does not use any variables.

`[variables] = Simulink.findVars(context,variablefilter)` finds only the variables or enumerated types that are specified by `variablefilter`.

`[variables] = Simulink.findVars( ___ ,Name,Value)` finds variables with additional options specified by one or more `Name,Value` pair arguments. For example, you can search for enumerated data types that are used in `context`, in addition to variables.

Examples

Variables in Use in a Model

Find variables used by `MyModel`.

```
variables = Simulink.findVars('MyModel');
```

Specific Variable in Use in a Model

Find all uses of the base workspace variable `k` by `MyModel`. Use the cached results to avoid compiling `MyModel`.

```
variables = Simulink.findVars('MyModel','Name','k',
'SearchMethod','cached','SourceType','base workspace');
```

Regular Expression Matching

Find all uses of a variable whose name matches the regular expression `^trans`.

```
variables = Simulink.findVars('MyModel','Regexp','on',
'Name','^trans');
```

Variables Common to Two Models

Given two models, find the variables used by the first model, the second, and both

```
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
commonVars = intersect(model1vars,model2Vars);
```

Variables Not Used in a Model

Find the variables that are defined in the model workspace of `MyModel` but that are not used by the model.

```
unusedVars = Simulink.findVars('MyModel','FindUsedVars',false,
'SourceType','model workspace');
```

Specific Variable Not Used in a Model

Determine if the base workspace variable `k` is not used by `MyModel`.

```
varObj = Simulink.VariableUsage('k','base workspace');
unusedVar = Simulink.findVars('MyModel',varObj,
'FindUsedVars',false);
```

Variables Used by a Block

Find the variables that are used by the block `Gain1` in `MyModel`.

```
variables = Simulink.findVars('MyModel',
'Users','MyModel/Gain1');
```

Variables Used in a Model Reference Hierarchy

Find the variables that are used in a model reference hierarchy. Begin the search with the model `MyNestedModel`, and search the entire hierarchy below `MyNestedModel`.

```
variables = Simulink.findVars('MyNestedModel','SearchReferencedModels','on');
```

Variables and Enumerated Types Used in a Model

Find variables and enumerated types that are used in `MyModel`.

```
varsAndEnumTypes = Simulink.findVars('MyModel','IncludeEnumTypes',true);
```

- "Search Using Model Explorer"

Input Arguments

`context` — Models and blocks to search

string | cell array of strings

Models and blocks to search, specified as a string or a cell array of strings. You can specify `context` in one of the following ways:

- The name of a model. For example, (`'vdp'`) specifies the model `vdp.slx`.
- The name or path of a block or masked block. For example, (`'vdp/Gain1'`) specifies a block named `Gain1` at the root level of the model `vdp.slx`.
- A cell array of model or block names.

Data Types: `char | cell`

`variablefilter` — Specific variables to find

array of `Simulink.VariableUsage` objects

Specific variables to find, specified as an array of `Simulink.VariableUsage` objects. Each `Simulink.VariableUsage` object identifies a variable to find.

Example:

```
vars = [Simulink.VariableUsage('k','base workspace')
   Simulink.VariableUsage('myParam','base workspace')];
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',false`

`'FindUsedVars'` — Find variables that are used or not used

`true` (default) | `false`

Flag to find variables that are explicitly used or not used, specified as the comma-separated pair consisting of `'FindUsedVars'` and `true` or `false`. If you specify `FindUsedVars` as `false`, the function finds variables that are not used in `context` but that are defined in the workspace specified by `SourceType`.

Example: `'FindUsedVars',false`

`'IncludeEnumTypes'` — Find enumerated types that are used

`false` (default) | `true`

Flag to find enumerated data types that are used, specified as the comma-separated pair consisting of `'IncludeEnumTypes'` and `true` or `false`. The function finds enumerated types that are used explicitly in `context` as well as types that define variables that are used in `context`.

If you specify `SourceType` as `'base workspace'`, `'model workspace'`, or `'mask workspace'`, the function does not report enumerated types because those sources cannot define enumerated types.

You cannot find unused enumerated types by specifying `FindUsedVars` as `false`.

Example: `'IncludeEnumTypes',true`

`'RegExp'` — Enable regular expression matching

`'off'` (default) | `'on'`

Flag to enable regular expression matching for input arguments, specified as the comma-separated pair consisting of `'RegExp'` and `'on'`. You can match only input arguments that have string values.

Example: `'RegExp','on'`

`'SearchMethod'` — Compile status

`'compiled'` (default) | `'cached'`

Compile status, specified as the comma-separated pair consisting of `'SearchMethod'` and one of these values:

- `'compiled'` — Return up-to-date results by compiling every model in the search context before search.
- `'cached'` — Return quicker results by using results cached during the previous compile.

Example: `'SearchMethod','compiled'`

`'SearchReferencedModels'` — Enable search in referenced models
`'off'` (default) | `'on'`

Flag to enable search in referenced models, specified as the comma-separated pair consisting of `'SearchReferencedModels'` and `on`.

Example: `'SearchReferencedModels','on'`

`'Name'` — Name of a variable or enumerated type to search for
string

Name of a variable or enumerated data type to search for, specified as the comma-separated pair consisting of `'Name'` and a string.

Example: `'Name','trans'`

Data Types: `char`

`'SourceType'` — Workspace or source defining the variables or enumerated types
string

Workspace or source defining the variables, specified as the comma-separated pair of `'SourceType'` and one of these options:

- `'base workspace'`
- `'model workspace'`
- `'mask workspace'`
- `'data dictionary'`

The function filters results for variables that are defined in the specified source.

Example: `'SourceType','base workspace'`

If you search for enumerated data types by specifying `'IncludeEnumTypes'` as `true`, `'SourceType'` represents the way an enumerated type is defined. You can specify one of these options:

- `'MATLAB file'`
- `'dynamic class'`
- `'data dictionary'`

The function filters results for enumerated types that are defined in the specified source.

Example: `'SourceType','MATLAB file'`

If you do not specify `SourceType`, the function does not filter results by source.

`'Users'` — Name of block to search for variables

string

Name of specific block to search for variables, specified as the comma-separated pair consisting of `'Users'` and a string.

To search a set of specific blocks, enable regular expression matching by specifying `RegExp` as `'on'` and use regular expressions in the string. For example, you can specify `'Users','MyModel/Gain*` to search all blocks in `MyModel` whose names begin with `Gain`.

Example: `'Users','MyModel/Gain1'`

Example: `'Users','MyModel/mySubsystem/Gain2'`

Example: `'Users','MyModel/Gain*`

Limitations

`Simulink.findVars` does not work with these constructs:

- MATLAB code in scripts and initialization and callback functions
- Libraries and blocks in libraries
- Variables in `MATLAB Function` blocks, except for input arguments

  However, `Simulink.findVars` can find enumerated types anywhere they are used in `MATLAB Function` blocks.
- Calls directly to MATLAB from the Stateflow action language
- S-functions that use data type variables registered using `ssRegisterDataType`

  To make the variables searchable, use `ssRegisterTypeFromNamedObject` instead.

- Variables referenced by machine-parented data in Stateflow

`Simulink.findVars` discovers variable usage in inactive subsystem variants only if you select **Analyze all choices during update diagram and generate preprocessor conditionals** in the `Variant Subsystem` block dialog box. If you do not select this check box, the function does not discover variable usage in inactive variants.

More About

- "Model Exploration"
- "Variables"

See Also

Simulink.VariableUsage | `find_system` | intersect

Introduced in R2010a

Simulink.getFileChecksum

Checksum of file

Syntax

```
checksum = Simulink.getFileChecksum(filename)
```

Description

`checksum = Simulink.getFileChecksum(filename)` returns the checksum of the specified file, using the MD5 checksum algorithm. Use the checksum to see if the file has changed compared to a previous checksum. You can use checksums as part of an audit trail.

Use `Simulink.getFileChecksum` to get a checksum for any file. If the file contents do not change from one checksum to the next, the checksum from `Simulink.getFileChecksum` stays the same. Otherwise, the checksum is different with each change to the file contents.

For functional information on a model, use `Simulink.BlockDiagram.getChecksum` instead. `Simulink.BlockDiagram.getChecksum` looks at the functional aspect of the model. If the functional aspect doesn't change, then `Simulink.BlockDiagram.getChecksum` returns the same checksum.

For example, if you moved a block, the file contents are different (measured by `Simulink.getFileChecksum`) but the function of the model is unchanged (measured by `Simulink.BlockDiagram.getChecksum`).

Examples

Get Checksum of a File

Use `fullfile` to specify a full path to a file and get the checksum.

```
filechecksum = Simulink.getFileChecksum(fullfile(matlabroot,'toolbox',...
```

```
'matlab','demos','gatlin.mat'));
```

Input Arguments

`filename` — File name to get checksum for

file of any type

File name to get checksum for, with file extension and optional full path. Use `fullfile` to specify a full path to a file, or use the form `'C:\Work\filename.mat'`.

Example: `'lengthofline.m'`

Data Types: `char`

Output Arguments

`checksum` — Checksum value

string

Checksum value in a 32-character string.

See Also

`Simulink.BlockDiagram.getChecksum` | `Simulink.SubSystem.getChecksum`

Introduced in R2014b

Simulink.ModelDataLogs.convertToDataset

Convert logging data from `Simulink.ModelDataLogs` format to `Simulink.SimulationData.Dataset` format

Syntax

```
convertedDataset =
sourceModelDataLogsObject.convertToDataset(convertedDatasetName)
```

Description

Note: The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

`convertedDataset = sourceModelDataLogsObject.convertToDataset(convertedDatasetName)` converts the *`sourceModelDataLogsObject`* to a `Simulink.SimulationData.Dataset` object. The name of the converted object is based on *`convertedDatasetName`*.

The resulting `Simulink.SimulationData.Dataset` object is a flat list. This list has one element for each `Simulink.Timeseries` or `Simulink.TsArray` object in the `Simulink.ModelDataLogs` object.

Limitations

Source of Simulink.ModelDataLogs Logged Data	Conversion Limitation
Referenced model	Loads all ancestors of the referenced model not previously loaded. If any ancestor model does not appear on the MATLAB path, the conversion fails. If the model has changed, or the model ancestors have changed, after Simulink logged the data, the conversion can fail. For example, adding, deleting, or renaming a block after logging can cause conversion failure.
Variant model or subsystem	The current active variant must be the same one that was active when Simulink logged the data. Otherwise, the conversion fails.
Frame signal	The conversion fails.
Mux block	The conversion produces a different `Simulink.SimulationData.Dataset` object as the dataset than Simulink creates when you simulate the model using the `Dataset` format for the logged data.
Stateflow chart	Not supported.

Input Arguments

`sourceModelDataLogsObject`

A `Simulink.ModelDataLogs` object that you want to convert to a `Simulink.SimulationData.Dataset` object.

`convertedDatasetName`

Name of the dataset that the conversion process creates.

Output Arguments

`convertedDataset`

The `Simulink.SimulationDataset` object that the `Simulink.ModelDataLogs.convertToDataset` function creates.

For details about the converted dataset, see Simulink.SimulationData.Dataset.

Example

In releases before R2016a, you could log signals using `ModelDataLogs` format. If you have a MAT-file with signal logging data that uses the `ModelDataLogs` format, here is how you can convert that data to `Dataset` format. This example assumes that the model that generated the logging data had the **Configuration Parameters** > **Data Import/Export** > **Signal logging** name set to `logsout`.

1. Load the MAT-file.
2. Convert `logsout` to a dataset called `myModel_dataset`. (The elements information will be different for your data.)

```
dataset = logsout.convertToDataset('myModel_Dataset')

dataset =
  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'myModel_Dataset'
    Total Elements: 2

  Elements:
    1: 'x1'
    2: 'x2'

  -Use get or getElement to access elements by index or name.
  -Use addElement or setElement to add or modify elements.

  Methods, Superclasses
```

More About

- “Migrate Scripts That Use ModelDataLogs API”
- “Export Signal Data Using Signal Logging”

See Also

`Simulink.ModelDataLogs` | Simulink.SimulationData.Dataset | `Simulink.SimulationData.updateDatasetFormatLogging`

Introduced in R2011a

getBlockSimState

Class: Simulink.SimState.ModelSimState
Package: Simulink.SimState

Access SimState of individual `Stateflow Chart`, `MATLAB Function`, or S-function block

Syntax

```
blockSimState = getBlockSimState(x, 'blockpath')
```

Description

`blockSimState = getBlockSimState(x, 'blockpath')` returns the SimState of the block specified as *`blockpath`*. *`blockpath`* must be either a `Stateflow Chart`, `MATLAB Function`, or S-function block. For other types of blocks, see the `loggedStates` property of the Simulink.SimState.ModelSimState class.

Input Arguments

`x`

The x argument is a `Simulink.SimState.ModelSimState` object.

`blockpath`

The path to the block for which you are requesting the SimState values.

Output Arguments

`blockSimState`

The SimState of the block specified.

Examples

```
chartState = getBlockSimState(x, 'mymodel/chart')
```

See Also

Simulink.SimState.ModelState.setBlockSimState

setBlockSimState

Class: Simulink.SimState.ModelSimState
Package: Simulink.SimState

Set SimState of individual `Stateflow Chart`, `MATLAB Function`, or S-function block

Syntax

```
setBlockSimState(x,'blockpath', blockSimState)
```

Description

`setBlockSimState(x,'blockpath', blockSimState)` sets the SimState of the block specified as *`blockpath`*. *`blockpath`* must be either a `Stateflow Chart`, `MATLAB Function`, or S-function block. For other types of blocks, see the `loggedStates` property of the Simulink.SimState.ModelSimState class.

Input Arguments

`x`

The argument *`x`* is a `Simulink.SimState.ModelSimState` object.

`blockpath`

The path to the block for which you are setting the SimState values

`blockSimState`

The SimState of the block specified.

Examples

```
newObj = setBlockSimState(obj, 'mymodel/chart', newChartState);
```

See Also

Simulink.SimState.ModelState.getBlockSimState

Simulink.saveVars

Save workspace variables and their values in MATLAB code format

Syntax

Note: `Simulink.saveVars` is not recommended. Use `matlab.io.saveVariablesToScript` instead.

```
Simulink.saveVars(filename)
Simulink.saveVars(filename, VarNames)
Simulink.saveVars(filename, '-regexp', RegExps)
Simulink.saveVars(filename, Specifications, UpdateOption)
Simulink.saveVars(filename, Specifications, Configuration)
Simulink.saveVars(filename, Specifications, MatlabVer)
[r1, r2] = Simulink.saveVars(filename, Specifications)
```

Description

`Simulink.saveVars(filename)` saves all variables in the current workspace for which MATLAB code can be generated to a MATLAB file named *`filename`*`.m`. If MATLAB code cannot be generated for a variable, the variable is saved into a companion MAT-file named *`filename`*`.mat`, and a warning is generated. If either file already exists, it is overwritten. The *`filename`* cannot match the name of any variable in the current workspace, and can optionally include the suffix `.m`. Using `Simulink.saveVars` has no effect on the contents of any workspace.

Executing the MATLAB file restores the variables saved in the file to the current workspace. If a companion MAT-file exists, code in the MATLAB file loads the MAT-file, restoring its variables also. When both a MATLAB file and a MAT-file exist, do not load the MATLAB file unless the MAT file is available, or an error will occur. Do not load a MAT-file directly, or incomplete data restoration will result. No warning occurs if loading a file overwrites any existing variables.

You can edit a MATLAB file that `Simulink.saveVars` creates. You can insert comments between or within the MATLAB code sections for saved variables. However,

if you later use `Simulink.saveVars` to update or append to the file, only comments between MATLAB code sections will be preserved. Internal comments should therefore be used only in files that you do not expect to change any further.

You must not edit the header section in the MATLAB file, which comprises the first five comment lines. Simulink does not check that a manually edited MATLAB file is syntactically correct. MathWorks recommends not editing any MATLAB code in the file. You cannot edit a MAT-file and should never attempt to do so.

`Simulink.saveVars(`*`filename`*`, `*`VarNames`*`)` saves only the variables specified in *`VarNames`*, which is a comma-separated list of variable names. You can use the wildcard character `*` to save all variables that match a pattern. The `*` matches one or more characters, including non-alphanumeric characters.

`Simulink.saveVars(`*`filename`*`, '-regexp', `*`RegExps`*`)` saves only variables whose names match one of the regular expressions in *`RegExps`*, which is a comma-separated list of expressions. See “Regular Expressions” 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.

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', 10, '-v4');
```

Specify a 2-D slice for the output of the `my3Dtable` 3-D array. Specify that the 2-D slice expands along the first and third dimensions:

```
my3DTable = zeros(3, 4, 2, 'single');
Simulink.saveVars('mfile.m', 'my3DTable', '-2dslice', 1, 3);
```

The generated MATLAB code is:

```
my3DTable = zeros(3, 4, 2, 'single');
my3DTable (:,1,:) = single ( ...
  [1 13;
   5 17;
   9 21]);
my3DTable (:,2,:) = single( ...
  [2 14;
   6 18;
   10 22]);
my3DTable (:,3,:) = single( ...
  [3 15;
   7 19;
   11 23]);
my3DTable (:,4,:) = single( ...
  [4 16;
   8 20;
   12 24]);
```

Limitations

The `Simulink.saveVars` function:

- Does not preserve shared references.
- Ignores dynamic properties of objects.
- Saves the following to the MAT-file although they could appear in the MATLAB file:
  - `fi` objects.
  - `Simulink.ConfigSet` objects with custom target components. (Use the `Simulink.ConfigSet` method `saveAs` instead.)
  - `Simulink.Timeseries` and `Simulink.ModelDataLogs` objects.

More About

Tips

- If you do not need to save variables in an easily-understood form, see the `save` function.
- If you need to save only bus objects, use the `Simulink.Bus.save` function.
- If you need to save only a configuration set, use the `Simulink.ConfigSet.saveAs` method.

See Also

`save` | `matlab.io.saveVariablesToScript` | `Simulink.Bus.save` | `Simulink.ConfigSet` | `Simulink.Bus.save`

Introduced in R2010a

Simulink.sdi.addToRun

Add simulation data to existing run

Syntax

```
signalIDs = Simulink.sdi.addToRun(runID,'base',varName)
signalIDs = Simulink.sdi.addToRun(runID,'model',modelNameOrHandle)
signalIDs = Simulink.sdi.addToRun(runID,'vars',var)
signalIDs = Simulink.sdi.addToRun(runID,'namevalue',dataName,
dataValue)
```

Description

`signalIDs = Simulink.sdi.addToRun(runID,'base',varName)` adds data, `varName`, from the base workspace to an existing run, specified by `runID`.

`signalIDs = Simulink.sdi.addToRun(runID,'model',modelNameOrHandle)` adds model simulation data, specified on the **Data Import/Export** pane of the Configuration Parameters dialog box, to an existing run, specified by `runID`. Open the model before you use this syntax.

`signalIDs = Simulink.sdi.addToRun(runID,'vars',var)` adds data stored as variables, `var`, from the calling workspace to an existing run, specified by `runID`.

`signalIDs = Simulink.sdi.addToRun(runID,'namevalue',dataName, dataValue)` adds simulation data `dataValue`, to an existing run, specified by `runID`, and lets you specify a name, `dataName`, for the data in the run.

Examples

Add Simulation Data from Base Workspace

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat', ...
             'StructureWithTime','ReturnWorkspaceOutputs','on');
```

```
% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run')

% Add simulation output from the base workspace
Simulink.sdi.addToRun(runID,'base',{'simOut'});

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add Simulation Data As Specified in a Model

```
% Open the model
sldemo_absbrake;
```

Click **Run** to simulate the model.

```
% Create a Data Inspector run
runID = Simulink.sdi.createRun('My Run');
Simulink.sdi.addToRun(runID,'model','sldemo_absbrake');

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add Simulation Data by Passing Variables Directly to `Simulink.sdi.addToRun`

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat', ...
             'StructureWithTime','ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run');
Simulink.sdi.addToRun(runID,'vars',simOut);

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add Simulation Data and Name the Data

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run');
```

```
% Name simulation output passed to Simulink.sdi.addToRun
Simulink.sdi.addToRun(runID,'namevalue',{'MyData'},{simOut});

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`runID` — Unique run identifier
integer

Unique number identifying a run in the Simulation Data Inspector, specified as an integer.

`varName` — Base workspace data
cell array

The names of variables in the base workspace, specified as a cell array of strings.

Example: `{'simOut'}`

`modelNameOrHandle` — Model name
string

The model name, or a model handle, specified as a string.

Example: `'sldemo_absbrake'`

`var` — Variable data
variable

Data stored as variables. These variables are assumed to be in the calling workspace.

Example: `simOut`

`dataName` — Signal data name
call array

Name of the data in the run, specified as a cell array.

Example: `{'MyData'}`

`dataValue` — Signal data values

cell array

Values of the signal data, specified as a cell array.

Example: `{simOut}`

Output Arguments

`signalIDs` — Unique signal identifier

array

Unique signal identifier, returned as an array of integers where each element is a unique ID for a signal added to the run.

See Also

`Simulink.sdi.createRun` | Simulink.sdi.Run | `Simulink.sdi.view`

Introduced in R2011b

Simulink.sdi.changeLoggedToStreamed

Change signals marked for logging to streaming

Syntax

```
Simulink.sdi.changeLoggedToStreamed(model)
Simulink.sdi.changeLoggedToStreamed(model,Name,Value)
```

Description

`Simulink.sdi.changeLoggedToStreamed(model)` changes signals in `model` marked for logging to streaming.

`Simulink.sdi.changeLoggedToStreamed(model,Name,Value)` uses additional option specified by one or more `Name,Value` pair arguments that are inherited from the method `createFromModel` of the class `Simulink.SimulationData.ModelLoggingInfo`.

Examples

Change Logged Signals to Streamed for Top-Level System

```
% Open the sldemo_absbrake model
sldemo_absbrake;

% Change the logged signal to streamed signals in the model
Simulink.sdi.changeLoggedToStreamed('sldemo_absbrake');
```

Change Logged Signals to Streamed Excluding Referenced Models

```
% Open the model
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_bus_logging')));

% Change the logged signal to streamed signals in the model
Simulink.sdi.changeLoggedToStreamed(gcs,'ReferencedModels','off');
```

The logged signals in the top model are changed to streaming. The signal logged in the referenced model `ex_mdlref_counter_bus` is unchanged.

Save, Remove, and Restore Streamed Signals

You can save the set of streamed signals to a file and then restore them to your model at a later time using the Simulink.HMI.InstrumentedSignals class.

```
% Save signals that are currently streamed in a model
sigs = get_param(bdroot,'InstrumentedSignals');
save my_instrumented_signals.mat sigs

% Remove all streamed signals from the model
set_param(bdroot,'InstrumentedSignals',[]);

% Restore streamed signals to the model
load my_instrumented_signals.mat
set_param(bdroot,'InstrumentedSignals',sigs);
```

- "Stream Data to the Simulation Data Inspector"
- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`model` — Model name

string

Model name or handle, specified as a string.

Example: `'vdp'bdroot`

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: `'Variants','ActiveVariants'`

`'FollowLinks'` — Library links

`'on'` (default) | `'off'`

Include library links, specified as the comma-separated pair consisting of `'FollowLinks'` and one of these values:

- `'on'` — Include logged signals inside libraries.
- `'off'` — Skip all libraries.

Example: `'FollowLinks','on'`

`'LookUnderMasks'` — Masks

`'all'` (default) | `'none'` | `'graphical'` | `'functional'`

Include masks, specified as the comma-separated pair consisting of `'LookUnderMasks'` and one of these values:

- `'all'` — Include logged signals from all masked subsystems.
- `'none'` — Skip all masked subsystems.
- `'graphical'` — Include logged signals from masked subsystems with no workspace and no dialog box.
- `'functional'` — Include logged signals from masked subsystems with no dialog box.

Example: `'LookUnderMasks','all'`

`'Variants'` — Variants

`'ActiveVariants'` (default) | `'AllVariants'`

Include variants, specified as the comma-separated pair consisting of `'Variants'` and one of these values:

- `'ActiveVariants'` — Include logged signals only in active subsystem and model reference variants.
- `'AllVariants'` — Include logged signals all subsystem and model reference variants.

Example: `'Variants','ActiveVariants'`

`'IncludeCommented'` — Commented blocks

`'off'` (default) | `'on'`

Include commented blocks, specified as the comma-separated pair consisting of `'IncludeCommented'` and one of these values:

- `'on'` — Include commented blocks.
- `'off'` — Skip commented blocks.

Example: `'IncludeCommented','off'`

`'ReferencedModels'` — Referenced models
`'on'` (default) | `'off'`

Include referenced models, specified as the comma-separated pair consisting of `'ReferencedModels'` and one of these values:

- `'on'` — Include logged signals from reference models.
- `'off'` — Skip all reference models.

Example: `'ReferencedModels','on'`

See Also

`Simulink.sdi.changeStreamedToLogged` | `Simulink.sdi.markSignalForStreaming`

Introduced in R2015b

Simulink.sdi.changeStreamedToLogged

Change signals marked for streaming to logging

Syntax

```
Simulink.sdi.changeStreamedToLogged(model)
Simulink.sdi.changeStreamedToLogged(model,Name,Value)
```

Description

`Simulink.sdi.changeStreamedToLogged(model)` changes signals in `model` marked for streaming to logging. If you use this function, any connections to Dashboard blocks are broken.

`Simulink.sdi.changeStreamedToLogged(model,Name,Value)` uses additional option specified by one or more `Name,Value` pair arguments.

Examples

Change Streamed Signals to Logged for Top-Level System

```
% Open the sldemo_absbrake model
sldemo_absbrake;

% Change the logged signal to streamed signals in the model
Simulink.sdi.changeLoggedToStreamed('sldemo_absbrake');
```

- "Stream Data to the Simulation Data Inspector"
- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`model` — Model name

string

Model name or handle, specified as a string.

Example: `'vdp'bdroot`

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: `'Variants','ActiveVariants'`

`'FollowLinks'` — Library links

`'on'` (default) | `'off'`

Include library links, specified as the comma-separated pair consisting of `'FollowLinks'` and one of these values:

- `'on'` — Include logged signals inside libraries.
- `'off'` — Skip all libraries.

Example: `'FollowLinks','on'`

`'LookUnderMasks'` — Masks

`'all'` (default) | `'none'` | `'graphical'` | `'functional'`

Include masks, specified as the comma-separated pair consisting of `'LookUnderMasks'` and one of these values:

- `'all'` — Include logged signals from all masked subsystems.
- `'none'` — Skip all masked subsystems.
- `'graphical'` — Include logged signals from masked subsystems with no workspace and no dialog box.
- `'functional'` — Include logged signals from masked subsystems with no dialog box.

Example: `'LookUnderMasks','all'`

`'Variants'` — Variants

`'ActiveVariants'` (default) | `'AllVariants'`

Include variants, specified as the comma-separated pair consisting of `'Variants'` and one of these values:

- `'ActiveVariants'` — Include logged signals only in active subsystem and model reference variants.
- `'AllVariants'` — Include logged signals all subsystem and model reference variants.

Example: `'Variants','ActiveVariants'`

`'IncludeCommented'` — Commented blocks

`'off'` (default) | `'on'`

Include commented blocks, specified as the comma-separated pair consisting of `'IncludeCommented'` and one of these values:

- `'on'` — Include commented blocks.
- `'off'` — Skip commented blocks.

Example: `'IncludeCommented','off'`

`'ReferencedModels'` — Referenced models

`'on'` (default) | `'off'`

Include referenced models, specified as the comma-separated pair consisting of `'ReferencedModels'` and one of these values:

- `'on'` — Include logged signals from reference models.
- `'off'` — Skip all reference models.

Example: `'ReferencedModels','on'`

See Also

`Simulink.sdi.changeLoggedToStreamed` | `Simulink.sdi.markSignalForStreaming`

Introduced in R2015b

Simulink.sdi.clear

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','StructureWithTime');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

Simulink.sdi.clear;

% The number of runs is now zero.
runCount = Simulink.sdi.getRunCount()
```

- “Inspect and Compare Signal Data Programmatically”

Introduced in R2011b

Simulink.sdi.close

Close Simulation Data Inspector

Syntax

```
Simulink.sdi.close
Simulink.sdi.close(filename)
```

Description

`Simulink.sdi.close` closes the Simulation Data Inspector. It returns an error if there is unsaved data.

`Simulink.sdi.close(filename)` closes the Simulation Data Inspector and saves the data in the specified `filename`.

Examples

Close Simulation Data Inspector and Save Data

Log data, simulate a model, view the results, close the Simulation Data Inspector, and save the data.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a Data Inspector run
runID = Simulink.sdi.createRun('My Run');
Simulink.sdi.addToRun(runID, 'base', {'simOut'});

% See the results in Simulation Data Inspector
Simulink.sdi.view;

% Close the Simulation Data Inspector and save the data
Simulink.sdi.close('savedData.mat');
```

The data file, `savedData.mat`, is saved in the current working directory.

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`filename` — Filename to save data

string (default)

Filename to save data, specified as a string. The string must fully specify the target file to save.

Introduced in R2013b

Simulink.sdi.compareRuns

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.

Examples

Compare Simulation With Code Generation Results

```
% Load the model 'slexAircraftExample'
load_system('slexAircraftExample');

% Configure model "slexAircraftExample" for logging
set_param('slexAircraftExample','SolverType','Fixed-Step','SaveOutput','on',...
          'SaveFormat','StructureWithTime','ReturnWorkspaceOutputs','on');

% CD to temporary directory and build
cd(tempdir);
rtwbuild('slexAircraftExample');

% Run the executable
if ispc
   system('slexAircraftExample');
elseif unix
   system('./slexAircraftExample');
```

```
end

% Create a run using the slexAircraftExample.mat placed in the current directory
[run1ID,~,~] = Simulink.sdi.createRun('My Run','file','slexAircraftExample.mat');

% 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
```

Compare Two Runs Using Specified Alignment Algorithms

```
% 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});

% Get and change one of the parameters of the model
mws = get_param('slexAircraftExample','modelworkspace');
wsMq = mws.evalin('Mq');
mws.assignin('Mq',3*wsMq);

% 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
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`runID1` — Unique run identifier

integer

Run ID, a unique number identifying the first run for comparison, specified as an integer.

`runID2` — Unique run identifier

integer

Run ID, a unique number identifying the second run for comparison, specified as an integer.

`alignmentMethods` — Signal alignment methods

array

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 can use the following values.

Value	Align By
`Simulink.sdi.AlignType.BlockPath`	Path to the source block for the signal
`Simulink.sdi.AlignType.DataSource`	Data name (for example, `logsout.Stick.Data`)
`Simulink.sdi.AlignType.SID`	"Locate Diagram Components Using Simulink Identifiers"
`Simulink.sdi.AlignType.SignalName`	Signal name

For example,
`[Simulink.sdi.AlignType.DataSource,Simulink.sdi.AlignType.BlockPath,Simulink.s`
could be an alignment array.

Output Arguments

`diff` — Comparison difference data

object

Instance of the Simulink.sdi.DiffRunResult object that contains the differences between two simulation runs.

See Also

`Simulink.sdi.createRun` | Simulink.sdi.DiffRunResult

Introduced in R2011b

Simulink.sdi.compareSignals

Compare data from two signals

Syntax

```
diff = Simulink.sdi.compareSignals(signalID1,signalID2)
```

Description

`diff = Simulink.sdi.compareSignals(signalID1,signalID2)` compares two signals and returns the results in a `Simulink.sdi.DiffSignalsResult` object.

Examples

Compare Two Signals

Call `Simulink.sdi.createRun` to get signal IDs for a simulation run in the Simulation Data Inspector. The function `Simulink.sdi.compareSignals` returns a `Simulink.sdi.DiffSignalResult` object containing the result data of the comparison. From this object you can determine if the signals are different.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run and get signal IDs
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','namevalue',{'MyData'},{simOut});

sig1 = signalIDs(1);
sig2 = signalIDs(2);

% Compare two signals, which returns results in
% instance of a Simulink.sdi.DiffSignalResult object
diff = Simulink.sdi.compareSignals(sig1,sig2);

% Find if the signal data match
```

```
match = diff.match;

% Get the tolerance used in Simulink.sdi.compareSignals
tolerance = diff.tol;
```

Compare Signals From Two Different Runs

```
% Load the model 'slexAircraftExample'
load_system('slexAircraftExample');

% Configure model "slexAircraftExample" for logging
set_param('slexAircraftExample','SolverType','Fixed-Step','SaveOutput','on',...
          'SaveFormat','StructureWithTime','ReturnWorkspaceOutputs',...
          'on');

% CD to temporary directory and build
cd(tempdir);
rtwbuild('slexAircraftExample');

% Run the executable
if ispc
   system('slexAircraftExample');
elseif unix
   system('./slexAircraftExample');
end

% Create a Data Inspector run using slexAircraftExample.mat created in the current
% directory
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','file','slexAircraftExample.mat');

% Get first signal id to compare
sig1 = signalIDs(1);

% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Data Inspector run and get signal IDs
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','namevalue',...
                                         {'MyData'},{simOut});

% Get second signal id to compare
sig2 = signalIDs(1);
```

```
% compare two signals
result = Simulink.sdi.compareSignals(sig1, sig2);
if result.match
   disp('****The signals match****');
else
   disp('****The signals did not match****');
end

% Plot results in a figure
plot(result.tol,'Color','r');
hold on;
plot(result.diff,'Color','g');
legend('Tolerance','Difference');
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`signalID1` — Unique signal identifier

integer

Signal ID, a unique number identifying the first signal for comparison, specified as an integer.

`signalID2` — Unique signal identifier

integer

Signal ID, a unique number identifying the first signal for comparison, specified as an integer.

Output Arguments

`diff` — Comparison difference data

object

Simulink.sdi.DiffSignalResult object containing the results of the comparison.

See Also

`Simulink.sdi.createRun` | Simulink.sdi.DiffSignalResult

Introduced in R2011b

Simulink.sdi.copyRun

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.

Examples

Copy `Simulink.sdi.Run` Object Representing a Run In the Simulation Data Inspector

```
% Configure model 'slexAircraftExample' for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

[newRunID,runIndex,signalIDs] = Simulink.sdi.copyRun(runID);
```

```
% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`runID` — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as a string.

Output Arguments

`runIDcopy` — Unique run identifier

integer

The unique number identifying the copied run, returned as an integer.

`runIndex` — Simulation run index

integer

Number representing the new index to the list of runs currently in the Simulation Data Inspector, returned as an integer.

`signalIDs` — Unique signal identifiers

array

Vector of numbers, where each element is a unique ID for a signal in this run. The signal IDs are different in the new run.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.view`

Introduced in R2011b

Simulink.sdi.copyRunViewSettings

Copy signal line style from one run to another run

Syntax

```
alignedSignals = Simulink.sdi.copyRunViewSettings(sourceRun,
destinationRun,moveCheckboxes)
```

Description

`alignedSignals = Simulink.sdi.copyRunViewSettings(sourceRun, destinationRun,moveCheckboxes)` copies the signal line style and color from one run to another run. The function returns an array of signal identifiers that match between runs where the settings were copied.

Examples

Copy Signal Line Style and Color

```
% Configure model 'slexAircraftExample' for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create the first Simulation Data Inspector run
runID_1 = Simulink.sdi.createRun('First Run','base',{'simOut'});

% Simulate the model again
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create second run in the Simulation Data Inspector
runID_2 = Simulink.sdi.createRun('Second Run','base',{'simOut'});

% Copy the signal settings from first run to the second run
```

```
alignedSignals = Simulink.sdi.copyRunViewSettings(runID_1,runID_2,true);
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`sourceRun` — Source run identifier
integer or object

A run ID, specified as an integer, or a Simulink.sdi.Run object, specified as an object, for the source run.

`destinationRun` — Destination run identifier
integer or object

A run ID, specified as an integer, or a Simulink.sdi.Run object, specified as an object, for the destination run.

`moveCheckboxes` — Move plotted signals
`false` (default) | `true`

Indicate to move the check boxes in the **Runs** pane, which plot signals, from the source run to the destination run, specified as `true` or `false`.

Output Arguments

`alignedSignals` — Aligned signals between runs
integer array

Signals that match between runs where the settings were copied, returned as an array of signal identifiers. See "How the Simulation Data Inspector Aligns Signals" for more information on signal alignment.

See Also
`Simulink.sdi.copyRun`

Introduced in R2016a

Simulink.sdi.createRun

Create run in Simulation Data Inspector

Syntax

```
runID = Simulink.sdi.createRun
runID = Simulink.sdi.createRun(runName)
runID = Simulink.sdi.createRun(runName,'base',varName)
runID = Simulink.sdi.createRun(runName,'model',modelNameOrHandle)
runID = Simulink.sdi.createRun(runName,'vars',var)
runID = Simulink.sdi.createRun(runName,'namevalue',dataName,
dataValue)
runID = Simulink.sdi.createRun(runName,'file',fileName)

[runID,runIndex] = Simulink.sdi.createRun( ___ )
[runID,runIndex,signalIDs] = Simulink.sdi.createRun( ___ )
```

Description

`runID = Simulink.sdi.createRun` creates an empty unnamed run in the Simulation Data Inspector and returns the corresponding run ID.

`runID = Simulink.sdi.createRun(runName)` creates an empty run named `runName` in the Simulation Data Inspector and returns the corresponding run ID.

`runID = Simulink.sdi.createRun(runName,'base',varName)` creates a run in the Simulation Data Inspector with data, `varName`, from the base workspace.

`runID = Simulink.sdi.createRun(runName,'model',modelNameOrHandle)` creates a run in the Simulation Data Inspector with model simulation output data, as specified on the **Data Import/Export** pane of the Configuration Parameters dialog box. `modelNameOrHandle` is a string specifying the model name of a model handle. Open the model before you use this syntax.

`runID = Simulink.sdi.createRun(runName,'vars',var)` creates a run in the Simulation Data Inspector with data stored in variables, `var`. These variables must be in the calling workspace.

`runID = Simulink.sdi.createRun(runName,'namevalue',dataName, dataValue)` creates a run in the Simulation Data Inspector from simulation data `dataValue`. `dataName` specifies a name for the data.

`runID = Simulink.sdi.createRun(runName,'file',fileName)` creates a run in the Simulation Data Inspector with data from a MAT-file, `fileName`.

`[runID,runIndex] = Simulink.sdi.createRun( ___ )` creates a run in the Simulation Data Inspector and returns the run ID and the run index. Use this option with any of the input argument combinations in the previous syntaxes.

`[runID,runIndex,signalIDs] = Simulink.sdi.createRun( ___ )` creates a run in the Simulation Data Inspector and returns the run ID, the run index, and the signal IDs. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Empty Run With No Name

```
runID = Simulink.sdi.createRun;
```

Create Empty Run With a Name

```
runID = Simulink.sdi.createRun('My Run');
```

Create Run From Simulation Output In the Base Workspace

```
% Configure the model slexAircraftExample for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run from the simulation
% output data in the base workspace
Simulink.sdi.createRun('My Run','base',{'simOut'});

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;
```

Create Run Using Simulation Output From a Model

The model must be open to use this function signature.

```
% Open the model sldemo_absbrake
sldemo_absbrake;
```

Click **Run** to simulate the model. The model is already configured for signal logging.

```
% Create a Simulation Data Inspector run named 'My Run' using
% simulation output data from the model
Simulink.sdi.createRun('My Run','model','sldemo_absbrake');

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;
```

Create Run Using Passed Variables

```
% Configure the model slexAircraftExample for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run named 'My Run' using
% simulation output data from the model
Simulink.sdi.createRun('My Run','vars',simOut);

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;
```

Create Run and Include Name of Simulation Data

```
% Configure the model slexAircraftExample for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run named 'My Run' using
% simulation output data from the model
Simulink.sdi.createRun('My Run','namevalue',{'MyData'},{simOut});

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;
```

Create Run Using MAT-File Data

This example includes data from a code generation build and requires Simulink Coder.

```
% Load the model slexAircraftExample
```

```
load_system('slexAircraftExample');

% Configure the model for logging and simulate
set_param('slexAircraftExample','SolverType','Fixed-Step',...
          'SaveOutput','on',...
          'SaveFormat','StructureWithTime',...
          'ReturnWorkspaceOutputs','on');

% Build the model to a temporary directory
cd(tempdir);
rtwbuild('slexAircraftExample');

% Run the executable
if ispc
   system('slexAircraftExample');
elseif unix
   system('./slexAircraftExample');
end

% A MAT-file is generated in the current directory

% Create a Simulation Data Inspector run using the data in the MAT-file
Simulink.sdi.createRun('My Run','file','slexAircraftExample.mat');

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;
```

A run named, `My Run`, appears in the Simulation Data Inspector.

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`runName` — Run name

string

Name of the run as it appears in the Simulation Data Inspector, specified as a string.

`varName` — Base workspace data

cell array

The names of variables in the base workspace, specified as a cell array of strings.

Example: `{'simOut'}`

`modelNameOrHandle` — Model name
string

The model name, or a model handle, specified as a string.

Example: `'sldemo_absbrake'`

`var` — Variable data
variable

Data stored as variables. These variables are assumed to be in the calling workspace.

Example: `simOut`

`dataName` — Signal data name
call array

Name of the data in the run, specified as a cell array.

Example: `{'MyData'}`

`dataValue` — Signal data values
cell array

Values of the signal data, specified as a cell array.

Example: `{simOut}`

`fileName` — Simulation data file name
string

The file name and path of a MAT-file containing simulation data, specified as a string.

Example: `'slexAircraftExample.mat'`

Output Arguments

`runID` — Unique run identifier
integer

Unique number identifying a run in the Simulation Data Inspector, returned as an integer.

`runIndex` — Simulation run index

integer

Number representing an index to the list of runs currently in the Simulation Data Inspector, returned as an integer.

`signalIDs` — Unique signal identifiers

array

Vector of numbers, where each element is a unique ID for a signal in a run.

More About

Tips

- Before calling `Simulink.sdi.createRun` with either `'base'` or `'model'` as an input argument, you must configure the model for logging and simulate the model.
- When you create and add a run, the Simulation Data Inspector maintains a list of these runs. The first run in the list is given a `runIndex` of 1. If you delete a run from the Simulation Data Inspector, the subsequent runs move up the list and each `runIndex` changes. However, the run IDs remain the same.

See Also

`Simulink.sdi.deleteRun` | `Simulink.sdi.getRun` | Simulink.sdi.Run

Introduced in R2011b

Simulink.sdi.createRunsFromSimulationOutput

Create run from simulation output results

Syntax

```
[runIDs,runIndices] = Simulink.sdi.createRunsFromSimulationOutput(
simOutput)
```

Description

`[runIDs,runIndices] = Simulink.sdi.createRunsFromSimulationOutput(simOutput)` creates a run for the Simulation Data Inspector from model simulation output, Simulink.SimulationOutput. The function returns the run ID and run indices currently in the Simulation Data Inspector. The runs are named using the default run naming rule in the Simulation Data Inspector.

Examples

Create Run from Simulation Output

Create a run in the Simulation Data Inspector from simulation output.

```
% Simulate the model
simOutput = sim('sldemo_autotrans','SaveOutput','on');

% Create a run in the Simulation Data Inspector from the output
runIDs = Simulink.sdi.createRunsFromSimulationOutput(simOutput);

% View the run in the Simulation Data Inspector
Simulink.sdi.view
```

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`simOutput` — Simulation output

object array

An array of one or more simulation output results, specified as Simulink.SimulationOutput objects.

Output Arguments

`runIDs` — Unique run identifier

integer array

Unique run identifiers, returned as an array of integers.

`runIndices` — Location in Runs pane

integer array

Location in the Simulation Data Inspector **Runs** pane of the runs created by the function, returned as an array of integers.

See Also

`Simulink.sdi.createRun`

Introduced in R2015b

Simulink.sdi.createRunsFromSLDV

Import Simulink Design Verifier test case inputs

Syntax

```
runIDs = Simulink.sdi.createRunsFromSLDV(sldvData)
```

Description

`runIDs = Simulink.sdi.createRunsFromSLDV(sldvData)` returns unique run identifiers for the runs created from Simulink Design Verifier test case inputs, specified as `sldvData`. The runs are imported into the Simulation Data Inspector.

Examples

- "Simulink Design Verifier Data Files"
- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`sldvData` — Data file

string

Path and file name of the data file generated by Simulink Design Verifier analysis, specified as a string. For more information about data files, see "Simulink Design Verifier Data Files".

Example: `'Controller_sldvdata.mat'`

Data Types: `char`

Output Arguments

`runIDs` — Run identifiers

integer array

Unique run identifiers, returned as an array of integers. Each run has a corresponding unique integer associated to it.

See Also

`Simulink.sdi.createRun`

Introduced in R2015b

Simulink.sdi.deleteRun

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. After deleting a run, the subsequent runs move up the list and the run index for each run changes. However, the unique run IDs remain the same.

Examples

Remove Run from the Simulation Data Inspector

```
% Configure model 'slexAircraftExample' for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

Simulink.sdi.deleteRun(runID);
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`runID` — Unique run identifier

integer

A unique simulation run identifier, specified as an integer. The `Simulink.sdi.createRun` function creates a unique run ID. The run ID can also be accessed for a particular run using the `Simulink.sdi.getRunIDByIndex` function.

See Also

`Simulink.sdi.clear` | `Simulink.sdi.copyRun` | `Simulink.sdi.createRun` | Simulink.sdi.Run

Introduced in R2011b

Simulink.sdi.deleteSignal

Delete signal from Simulation Data Inspector

Syntax

```
Simulink.sdi.deleteSignal(signalID)
```

Description

`Simulink.sdi.deleteSignal(signalID)` deletes a signal from the Simulation Data Inspector **Runs** pane.

Examples

Delete Signal From Simulation Data Inspector

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat',...
             'StructureWithTime');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

% Delete the first signal in this run
run = Simulink.sdi.getRun(runID);
signalID = run.getSignalIDByIndex(1)
Simulink.sdi.deleteSignal(signalID);
```

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`signalID` — Unique signal identifier

integer

Unique number identifying a signal in a run, specified as an integer. To find the signal ID, you can use the method Simulink.sdi.Run.getSignalIDByIndex.

See Also

`Simulink.sdi.clear` | `Simulink.sdi.deleteRun`

Introduced in R2016a

Simulink.sdi.discardDataFromPriorSessions

Delete data from prior MATLAB session

Syntax

```
Simulink.sdi.discardDataFromPriorSessions
```

Description

`Simulink.sdi.discardDataFromPriorSessions` deletes data from prior MATLAB sessions. The data is no longer available to the Simulation Data Inspector.

Examples

Delete Data from Prior Session

If you open a new MATLAB session, and there is data from a from a prior session, you can delete the data from the Simulation Data Inspector repository.

Delete the data from the Simulation Data Inspector repository.

```
Simulink.sdi.discardDataFromPriorSessions
```

- "Inspect and Compare Signal Data Programmatically"

See Also

`Simulink.sdi.hasDataFromPriorSessions` | `Simulink.sdi.importDataFromPriorSessions`

Introduced in R2015b

Simulink.sdi.getRun

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.

Examples

Get `Simulink.sdi.Run` Object For Run in the Simulation Data Inspector

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
[runID,runIndex,signalIDs] = Simulink.sdi.createRun('My Run','base',{'simOut'});

runObj = Simulink.sdi.getRun(runID);
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`runID` — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as an integer.

Output Arguments

`runObj` — Run object handle
object

Object containing the signal data and metadata, returned as a Simulink.sdi.Run object handle.

See Also

`Simulink.sdi.createRun` | Simulink.sdi.Run

Introduced in R2011b

Simulink.sdi.getRunCount

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.

Examples

Check Run Count After Simulation

Call `Simulink.sdi.getRunCount` to get the number of runs currently in the Simulation Data Inspector after a simulation.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

% Get run count in the Simulation Data Inspector
runCount = Simulink.sdi.getRunCount()
```

```
runCount =

           1
```

- “Inspect and Compare Signal Data Programmatically”

Output Arguments

`runCount` — Number of runs
integer

Number of runs that exist in the Simulation Data Inspector, returned as an integer.

See Also

`Simulink.sdi.createRun` | Simulink.sdi.Run

Introduced in R2011b

Simulink.sdi.getRunIDByIndex

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.

Examples

Get Run IDs For All Runs in the Simulation Data Inspector

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
[runID, runIndex, signalIDs] = Simulink.sdi.createRun('Run1','base',{'simOut'});

% Get the ID of the previously created run by index
runID2 = Simulink.sdi.getRunIDByIndex(runIndex);

% Both runID and runID2 reference the same run and should be equal
isequal(runID, runID2)

ans =

     1
```

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`runIndex` — Simulation run index

integer

Number representing an index to the list of runs currently in the Simulation Data Inspector, specified as an integer. The run index is an output of the `Simulink.sdi.createRun` function.

Output Arguments

`runID` — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, returned as an integer.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRunCount` | `Simulink.sdi.isValidRunID`

Introduced in R2011b

Simulink.sdi.getRunNamingRule

Return run naming rule for next Simulation Data Inspector run

Syntax

```
runNamingRule = Simulink.sdi.getRunNamingRule()
```

Description

`runNamingRule = Simulink.sdi.getRunNamingRule()` returns the run naming rule for the next Simulation Data Inspector run.

Examples

Change and Restore the Simulation Run Naming Rule

```
% Load the model
sldemo_absbrake;

% Save the prior run naming rule and set a new rule
runNamingRule = Simulink.sdi.getRunNamingRule();
Simulink.sdi.setRunNamingRule('My special simulation');

% Record logged signals and send them to the Simulation Data Inspector
set_param('sldemo_absbrake','InspectSignalLogs',1);

% Run the simulation
set_param('sldemo_absbrake','SimulationCommand','Start');

% Restore the previous run naming rule
Simulink.sdi.setRunNamingRule(runNamingRule);

% Open the Simulation Data Inspector to see the new run
```

```
Simulink.sdi.view;
```

Output Arguments

`runNamingRule` — Current Simulation Data Inspector run naming rule

string

Current Simulation Data Inspector run naming rule, returned as a string containing any of the run naming rule tokens `<run_index>`, `<model_name>`, `<time_stamp>`, `<sim_mode>`, or a custom run naming rule.

See Also

`Simulink.sdi.resetRunNamingRule` | `Simulink.sdi.setRunNamingRule`

Introduced in R2015a

Simulink.sdi.getSignal

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.

Examples

Modify Signal Properties in Simulation Data Inspector

Get the `Simulink.sdi.Signal` object for a signal in the Simulation Data Inspector. With the signal object you can modify its comparison and visualization properties.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
     'SaveFormat','StructureWithTime');

% Create a Simulation Data Inspector run
[runID,runIndex,signalIDs] = Simulink.sdi.createRun('My Run','base',{'simOut'});

signalObj = Simulink.sdi.getSignal(signalIDs(1));

% Specify the comparison and visualization signal properties
signalObj.absTol = .5;
signalObj.syncMethod = 'intersection';
signalObj.interpMethod = 'linear';
signalObj.lineColor = [1,0.4,0.6];
signalObj.lineDashed = '-';
signalObj.checked = true;
```

```
% View the signals in Simulation Data Inspector GUI
Simulink.sdi.view;
```

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`signalID` — Unique signal identifier

integer

Signal ID, a unique number identifying a signal in the Simulation Data Inspector, specified as an integer.

Output Arguments

`signal` — Signal time series object

object

A signal properties object, returned as a Simulink.sdi.Signal handle object.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRun` | Simulink.sdi.Run | Simulink.sdi.Signal

Introduced in R2011b

Simulink.sdi.getSource

Return repository location for storing simulation data

Syntax

```
source = Simulink.sdi.getSource
```

Description

`source = Simulink.sdi.getSource` returns the location of the Simulation Data Inspector repository for storing simulation data.

Call this function before running multiple simulations in a `parfor` loop.

Examples

Record Data During Parallel Simulations

This example shows how to run parallel simulations using a `parfor` loop and record each run in the Simulation Data Inspector.

Open the Simulation Data Inspector.

```
Simulink.sdi.view;
```

Load the model.

```
mdl = 'sldemo_absbrake';
load_system(mdl);
```

Log signals to the Simulation Data Inspector.

```
set_param(mdl,'InspectSignalLogs','on');
```

Start a parallel pool with four workers.

```
myPool = parpool(4);
```

Run the simulation in a `parfor` loop.

```
parfor idx = 1:4
   % Run the simulation
   simOut = sim(mdl,'SaveOutput','on',...
                'SaveFormat','StructureWithTime',...
                'ReturnWorkspaceOutputs','on');
   % Create a simulation run in the Simulation Data Inspector
   Simulink.sdi.createRun(['Run' num2str(idx)],'namevalue',...
                          {'simout'},{simOut});
end
```

Delete the current parallel pool and close the model.

```
delete(myPool);
bdclose all;
```

Import the data from the parallel pool into the Simulation Data Inspector.

```
Simulink.sdi.importDataFromPriorSessions;
```

- "Inspect and Compare Signal Data Programmatically"

Output Arguments

`source` — Repository location

string

Location of the Simulation Data Inspector repository, returned as a string.

See Also

`Simulink.sdi.refresh`

Introduced in R2012a

Simulink.sdi.hasDataFromPriorSessions

Indicate if data is available from prior MATLAB session

Syntax

```
ret = Simulink.sdi.hasDataFromPriorSessions
```

Description

`ret = Simulink.sdi.hasDataFromPriorSessions` returns `true` if there is Simulation Data Inspector data available from a prior MATLAB session and `false` otherwise.

Examples

Check the Repository

If you open a new MATLAB session, then you can check if there is data in the Simulation Data Inspector repository from a prior session.

```
Simulink.sdi.hasDataFromPriorSessions
```

```
ans =

     1
```

In this case, the function returns `true` because there is data from a prior MATLAB session.

- "Inspect and Compare Signal Data Programmatically"

Output Arguments

`ret` — Data indicator

logical

Indicator that data is available from a prior MATLAB session, returned as `true` or `false`.

See Also

`Simulink.sdi.discardDataFromPriorSessions` | `Simulink.sdi.importDataFromPriorSessions`

Introduced in R2015b

Simulink.sdi.importDataFromPriorSessions

Import data from prior MATLAB session

Syntax

```
Simulink.sdi.importDataFromPriorSessions
```

Description

`Simulink.sdi.importDataFromPriorSessions` imports data from prior MATLAB sessions into the current session. The signals and runs become visible in the Simulation Data Inspector and available in the programmatic API.

Examples

Import Data from Prior Session

If you open a new MATLAB session, and there is data from a from a prior session, then you can import it into the Simulation Data Inspector.

Import the data into the Simulation Data Inspector, and make the data available in the API.

```
Simulink.sdi.importDataFromPriorSessions
```

- "Inspect and Compare Signal Data Programmatically"

See Also

`Simulink.sdi.discardDataFromPriorSessions` | `Simulink.sdi.hasDataFromPriorSessions`

Introduced in R2015b

Simulink.sdi.isValidRunID

Determine if run ID is valid

Syntax

```
isValid = Simulink.sdi.isValidRunID(runID)
```

Description

`isValid = Simulink.sdi.isValidRunID(runID)` returns `true` if the `runID` corresponds to a run currently in the Simulation Data Inspector. Otherwise, it returns `false`.

Examples

Verify Run IDs

Before comparing the simulation data of two runs, you can verify that the run IDs are valid.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Data Inspector run
run1ID = Simulink.sdi.createRun('First Run','base',{'simOut'});
run2ID = Simulink.sdi.createRun('Second Run','base',{'simOut'});

% Check if run IDs are valid in Simulation Data Inspector
run1ID_valid = Simulink.sdi.isValidRunID(run1ID);
run2ID_valid = Simulink.sdi.isValidRunID(run2ID);

if run1ID_valid & run2ID_valid
% Compare two runs
   difference = Simulink.sdi.compareRuns(run1ID,run2ID);
```

```
end
```

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`runID` — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as an integer.

Output Arguments

`isValid` — Valid run indicator

logical

Indication of whether the run is a valid Simulation Data Inspector run, returned as a Boolean value: `1`, if the run exists; `0`, otherwise.

See Also

`Simulink.sdi.compareRuns` | `Simulink.sdi.createRun` | Simulink.sdi.Run

Introduced in R2011b

Simulink.sdi.load

Load saved Simulation Data Inspector session

Syntax

```
isValidSDIMatFile = Simulink.sdi.load(fileName)
```

Description

`isValidSDIMatFile = Simulink.sdi.load(fileName)` loads the runs, signals, tolerances, and signal selections from a MAT-file `fileName`.

Examples

Load Previous Simulation Data Inspector Session from a MAT-File

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');

% Create a run in the Simulation Data Inspector
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Save the current Simulation Data Inspector session
Simulink.sdi.save('my_runs.mat');

% Clear all data from the Simulation Data Inspector
Simulink.sdi.clear;

% Import saved MAT-file into the Simulation Data Inspector
Simulink.sdi.load('my_runs.mat');

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`fileName` — Session file name

string

File name, specified as a string, of the full or partial path of a Simulation Data Inspector session MAT-file.

Output Arguments

`isValidSDIMatFile` — Valid file indicator

logical

Indication of whether the MAT-file is a valid Simulation Data Inspector session file, returned as a Boolean value. Returns `1` if the file is valid and `0` if the file is not valid.

Alternatives

In the Simulation Data Inspector, click **Open** on the **Visualize** tab. For more information, see "Share Simulation Data Inspector Data and Views".

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.save` | `Simulink.sdi.view`

Introduced in R2011b

Simulink.sdi.markSignalForStreaming

Turn streaming on or off for a signal

Syntax

```
Simulink.sdi.markSignalForStreaming(block,port_index,state)
Simulink.sdi.markSignalForStreaming(port_handle,state)
Simulink.sdi.markSignalForStreaming(line_handle,state)
```

Description

`Simulink.sdi.markSignalForStreaming(block,port_index,state)` turns streaming on or off for a signal by specifying the block, `block`, and port index, `port_index`.

`Simulink.sdi.markSignalForStreaming(port_handle,state)` turns streaming on or off for a signal by specifying the block output port, `port_handle`.

`Simulink.sdi.markSignalForStreaming(line_handle,state)` turns streaming on or off for a signal by specifying the signal line handle, `line_handle`.

Examples

Use Block Path to Mark Signal for Streaming

```
% Open the model
sldemo_absbrake;

% Turn on streaming for the Weight block signal outport
% in sldemo_absbrake model
Simulink.sdi.markSignalForStreaming('sldemo_absbrake/Weight',1,'on');
```

Use Port Handle to Mark Signal for Streaming

Open a model, and select a block with a signal outport to stream.

```
% Get the selected block port handles
```

```
phs = get_param(gcb,'PortHandles')

% Turn streaming on for the block outport
Simulink.sdi.markSignalForStreaming(phs.Outport(1),'on');
```

Use Line Handle to Turn Signal Streaming Off

Open a model, and select a signal marked for streaming.

```
% Get the signal line handles for the model
slhs = get_param(gcs,'Lines');

% Turn streaming off for the first signal in the structure
Simulink.sdi.markSignalForStreaming(slhs(1).Handle,'off');
```

Save, Remove, and Restore Streamed Signals

You can save the set of streamed signals to a file and then restore them to your model at a later time using the Simulink.HMI.InstrumentedSignals class.

```
% Save signals that are currently streamed in a model
sigs = get_param(bdroot,'InstrumentedSignals');
save my_instrumented_signals.mat sigs

% Remove all streamed signals from the model
set_param(bdroot,'InstrumentedSignals',[]);

% Restore streamed signals to the model
load my_instrumented_signals.mat
set_param(bdroot,'InstrumentedSignals',sigs);
```

- “Stream Data to the Simulation Data Inspector”
- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`block` — Source block path or handle

string | handle

Block path or handle that contains the signal source, specified as a string.

Example: `'sf_car/shift_logic'`

`port_index` — Source block output port index

integer

Source block output index of the associated signal, specified as an integer.

Example: `1`

`state` — Streaming state toggle

`'on' | 'off'`

Turn streaming for a signal on or off, specified as a logical value.

Example: `'on'`

`port_handle` — Output port handle

handle

Source block output port of the associated signal, specified as a handle.

Example: `phs.Outport(1)`

`line_handle` — Signal line handle

handle

Source block signal line, specified as a handle.

Example: `sls(1).Handle`

See Also

`Simulink.sdi.changeLoggedToStreamed` | `Simulink.sdi.view`

Introduced in R2015b

Simulink.sdi.refresh

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 parallel simulations using a `parfor` loop and record each run in the Simulation Data Inspector.

Open the Simulation Data Inspector.

```
Simulink.sdi.view;
```

Load the model.

```
mdl = 'sldemo_absbrake';
load_system(mdl);
```

Log signals to the Simulation Data Inspector.

```
set_param(mdl,'InspectSignalLogs','on');
```

Start a parallel pool with four workers.

```
myPool = parpool(4);
```

Run the simulation in a `parfor` loop.

```
parfor idx = 1:4
```

```
    % Run the simulation
    simOut = sim(mdl,'SaveOutput','on',...
                 'SaveFormat','StructureWithTime',...
                 'ReturnWorkspaceOutputs','on');
    % Create a simulation run in the Simulation Data Inspector
    Simulink.sdi.createRun(['Run' num2str(idx)],'namevalue',...
                           {'simout'},{simOut});
end
```

Delete the current parallel pool and close the model.

```
delete(myPool);
bdclose all;
```

Import the data from the parallel pool into the Simulation Data Inspector.

```
Simulink.sdi.importDataFromPriorSessions;
```

- "Inspect and Compare Signal Data Programmatically"

See Also

`Simulink.sdi.getSource`

Introduced in R2012a

Simulink.sdi.report

Generate report from Simulation Data Inspector

Syntax

```
Simulink.sdi.report
Simulink.sdi.report(Name,Value)
```

Description

`Simulink.sdi.report` creates a report of the current view and data in the **Runs** pane in the Simulation Data Inspector.

`Simulink.sdi.report(Name,Value)` uses additional option specified by one or more `Name,Value` pair arguments.

Examples

Create Report from Runs Pane Plots

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create a Data Inspector run
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Select signals for plotting
for i = 1:length(signalIDs)
   signal = Simulink.sdi.getSignal(signalIDs(i));
   signal.checked = true;
end

% Create default report, which is the Runs pane view
```

```
Simulink.sdi.report;
```

Create Report from Comparisons Pane Plots

```
% Configure model "slexAircraftExample" for logging and simulate
set_param('slexAircraftExample/Pilot','WaveForm','square');
simOut = sim('slexAircraftExample','SaveOutput','on', ...
                  'SaveFormat','StructureWithTime', ...
                  'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run, Simulink.sdi.Run, from
% simOut in the base workspace
runID1 = Simulink.sdi.createRun('First Run','namevalue',{'simOut'},{simOut});

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample','SaveOutput','on', ...
             'SaveFormat','StructureWithTime', ...
             'ReturnWorkspaceOutputs','on');

% Create another Simulation Data Inspector run
runID2 = Simulink.sdi.createRun('Second Run','namevalue',{'simOut'},{simOut});

% Compare two runs
difference = Simulink.sdi.compareRuns(runID1,runID2);

% Specify columns to include in the report
metaDataOfInterest = [Simulink.sdi.SignalMetaData.Result, ...
                      Simulink.sdi.SignalMetaData.BlockPath1, ...
                      Simulink.sdi.SignalMetaData.RelTol];
% Report on the run comparison
Simulink.sdi.report('ReportToCreate','Compare Runs', ...
                    'ColumnsToReport',metaDataOfInterest, ...
                    'SignalsToReport','ReportAllSignals');
```

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ColumnsToReport','SignalName'`

`'ReportToCreate'` — Simulation Data Inspector pane and plots set

`'Inspect Signals'` (default) | `'Compare Runs'`

Simulation Data Inspector pane and plots to include in the report, specified as the comma-separated pair consisting of `'ReportToCreate'` and one of these values:

- `'Inspect Signals'` — Information in the **Runs** pane.
- `'Compare Runs'` — Information in the **Comparisons** pane.

Example: `'ReportToCreate','Inspect Signals'`

`'ReportOutputFolder'` — File output location

<current working folder>/slprj/sdi (default) | string

Location of the generated report, specified as the comma-separated pair consisting of `'ReportOutputFolder'` and a string.

`'ReportOutputFile'` — Report file name

`'SDI_report.html'` (default) | string

Report file name, specified as the comma-separated pair consisting of `'ReportOutputFile'` and a string.

`'PreventOverwritingFile'` — Report overwrite prevention

`true` (default) | `false`

Report overwrite prevention, specified as the comma-separated pair consisting of `'PreventOverwritingFile'` and one of the Boolean `true` or `false` values. If the report file exists and the value is `true`, then the report generator increments the file name. If `false`, the report generator overwrites the report file, if it exists.

`'ColumnsToReport'` — Metadata column specifier

array

Signal and run metadata column specifier, specified as the comma-separated pair consisting of `'ColumnsToReport'` and an array of column enumerations.

Array specifying values from the enumeration class, `Simulink.sdi.SignalMetaData`, which lists all signal metadata available in the Simulation Data Inspector. For example, to include the **Run** and **Synchronization Method** pane columns in a report, create an array `signal_metadata`.

```
signal_metadata = [Simulink.sdi.SignalMetaData.Run, Simulink.sdi.SignalMetaData.SyncMethod];
```

Specify the array as the value in the comma-separated pair.

```
Simulink.sdi.report('ColumnsToReport',signal_metadata);
```

These table columns are available for the **Runs** pane.

Column Enumeration Element	Description
`BlockPath` (default)	Block path
`SignalName` (default)	Signal name
`Line` (default)	Line style
`AbsTol` (default)	Absolute tolerance
`RelTol` (default)	Relative tolerance
`SyncMethod`	Method to align time vector: **union**, **intersection**, **uniform**
`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**
`Port`	Index of the signal output port
`Dimensions`	Number of signal dimensions
`Channel`	Channel of matrix data
`Run`	Name of a simulation run
`Model`	Model name

These table columns are available for the **Comparisons** pane.

Column Enumeration Element	Description
`Result` (default)	Result of the signal comparison between **Baseline** and **Compare To** runs
`AbsTol` (default)	Absolute tolerance
`RelTol` (default)	Relative tolerance
`AlignedBy` (default)	Metadata used to align signal data between **Baseline** and **Compare To** runs
`LinkToPlot` (default)	Link to a plot of each comparison result
`BlockPath1`	Block path for signal from **Baseline** run
`BlockPath2`	Block path for signal from **Compare To** run
`SignalName`	Signal name from **Baseline** run
`DataSource1`	Name for the data from **Baseline** run
`DataSource2`	Name for the data from **Compare To** run
`SyncMethod`	Synchronization method specified for the **Baseline** run
`InterpMethod`	Interpolation method specified for the **Baseline** run
`Channel1`	Channel specified for the **Baseline** run
`Channel2`	Channel specified for the **Compare To** run

`'ShortenBlockPath'` — Block path name length

`true` (default) | `false`

Block path name length, specified as the comma-separated pair consisting of `'ShortenBlockPath'` and one of the Boolean `true` or `false` values. If the value is `true` and the block path name is too long, the Simulation Data Inspector shortens the name in the report. If the value is `false`, then the report displays the entire block path name.

`'LaunchReport'` — Launch report after creation

`true` (default) | `false`

Launch report after creation, specified as the comma-separated pair consisting of `'LaunchReport'` and one of the Boolean `true` or `false` values. If the value is `true` after creation, the generated report opens.

`'SignalsToReport'` — Comparison signals set
`'ReportOnlyMismatchedSignals'` (default) | `'ReportAllSignals'`

For the **Comparisons** pane report only. Comparison signals set, specified as the comma-separated pair consisting of `'SignalsToReport'` and one of these values:

- `'ReportOnlyMismatchedSignals'` — Include only the mismatched signals from the comparison between the **Baseline** and **Compare To** runs.
- `'ReportAllSignals'` — Include all signals from the comparison between the **Baseline** and **Compare To** runs.

Alternatives

In the Simulation Data Inspector, on the **Visualize** or **Compare** tab, click **Create Report**. For more information, see "Create Simulation Data Inspector Report".

See Also

`Simulink.sdi.compareRuns` | `Simulink.sdi.createRun`

Introduced in R2011b

Simulink.sdi.resetRunNamingRule

Reset default run naming rule for next Simulation Data Inspector run

Syntax

```
Simulink.sdi.resetRunNamingRule
```

Description

`Simulink.sdi.resetRunNamingRule` resets the run naming rule to the default rule tokens for the next Simulation Data Inspector run.

Examples

Reset the Run Naming Rule to Default

```
% Load the model
sldemo_absbrake;

% Change the run naming rule
Simulink.sdi.setRunNamingRule('My special simulation');

% Record logged signals and send them to the Simulation Data Inspector
set_param('sldemo_absbrake','InspectSignalLogs',1);

% Run the simulation
set_param('sldemo_absbrake','SimulationCommand','Start');

% Reset the naming rule
Simulink.sdi.resetRunNamingRule;

% Run the simulation again
set_param('sldemo_absbrake','SimulationCommand','Start');

% Open the Simulation Data Inspector
Simulink.sdi.view;
```

The first run in the **Runs** pane of the Simulation Data Inspector uses the run naming rule `My special simulation`, and the second run uses the default run naming rule `Run <run_index>: <model_name>`, which appears as `Run 2: sldemo_absbrake`.

See Also

`Simulink.sdi.getRunNamingRule` | `Simulink.sdi.setRunNamingRule`

Introduced in R2015a

Simulink.sdi.save

Save current Simulation Data Inspector session

Syntax

```
Simulink.sdi.save(fileName)
```

Description

`Simulink.sdi.save(fileName)` saves all runs, signals, tolerances, and signal selections to a MAT-file `fileName`.

Examples

Save Simulation Data Inspector Runs

Save the Simulation Data Inspector simulation runs and specified tolerances to a MAT-file.

```
% Configure model "sldemo_fuelsys" for logging and simulate
simOut = sim('sldemo_fuelsys', 'SaveOutput','on', ...
             'SaveFormat', 'StructureWithTime', ...
             'ReturnWorkspaceOutputs', 'on');
% Create a run in the Simulation Data Inspector
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Save the current Simulation Data Inspector session
Simulink.sdi.save('my_runs.mat');
```

You can also load the information back in to the Simulation Data Inspector using the `Simulink.sdi.load` function.

- “Inspect and Compare Signal Data Programmatically”

Input Arguments

`fileName` — Session file name

String

A string specifying the target file to save.

Alternatives

In the Simulation Data Inspector, click **Save** on the **Visualize** tab. For more information, see "Share Simulation Data Inspector Data and Views".

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.load`

Introduced in R2011b

Simulink.sdi.setRunNamingRule

Specify a run naming rule

Syntax

```
Simulink.sdi.setRunNamingRule(rule)
```

Description

`Simulink.sdi.setRunNamingRule(rule)` specifies a template for naming a run in the Simulation Data Inspector.

Examples

Specify Run Naming Rule

```
Simulink.sdi.setRunNamingRule(...
   'Run <run_index> : <model_name> : <sim_mode>');
```

After recording a run, in the Signal Browser table of the Simulation Data Inspector, the run appears with a name similar to the following:

```
Run 1 : slexAircraftExample : normal
```

- "Inspect and Compare Signal Data Programmatically"
- "Run Management Configuration"

Input Arguments

`rule` — Rune naming rule

`<run_index>` | `<model_name>` | `<time_stamp>` | `<sim_mode>`

A string using predefined tokens and regular characters to create a template for run names. The available tokens are

- `<run_index>` — Sequential number of each run
- `<model_name>` — Name of model
- `<time_stamp>` — Run creation time
- `<sim_mode>` — Simulation mode for recorded run

The tokens are contained within a string. For example:

```
Simulink.sdi.setRunNamingRule('Run <run_index> : <model_name>');
```

Alternatives

In the Simulation Data Inspector, on the **Visualize** tab, click **Run Options**. In the dialog box, enter a string in the **Run naming rule** box.

Introduced in R2011b

Simulink.sdi.setRunOverwrite

Mark simulation run for overwrite

Syntax

```
Simulink.sdi.setRunOverwrite(runID,overwrite)
```

Description

`Simulink.sdi.setRunOverwrite(runID,overwrite)` marks a run identified by `runID` for overwriting in the Simulation Data Inspector with the next simulation run.

Examples

Mark Run for Overwriting

This example shows how to mark a run for overwriting in the Simulation Data Inspector.

Open the `sldemo_fuelsys` model.

On the Simulink Editor toolbar, click the **Simulation Data Inspector** button arrow and select **Send Logged Workspace Data to Data Inspector**.

Simulate the model.

Open the Simulation Data Inspector.

At the MATLAB Command Window, create a `runID` variable that uses the value of the **Run ID** for a run. You can find the **Run ID** for a run using the function `Simulink.sdi.getRunIDByIndex`.

```
runID = 1;
```

Set the overwrite condition to `true`.

```
Simulink.sdi.setRunOverwrite(runID,true);
```

In the Simulation Data Inspector, you can see the run is now marked to overwrite during the next simulation.

Simulate `sldemo_fuelsys`.

In the Simulation Data Inspector, the new simulation data replaces the previous run.

- "Inspect and Compare Signal Data Programmatically"
- "Run Management Configuration"

Input Arguments

`runID` — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as an integer.

`overwrite` — Overwrite run setting

`true | false`

The run overwrite setting, specified as a Boolean value. When set to `true`, the next simulation overwrites the run.

Example: `Simulink.sdi.setRunOverwrite(1,true);`

Alternatives

In the Simulation Data Inspector, select a run in the **Runs** pane and then click **Overwrite** on the **Visualize** tab.

Introduced in R2011b

Simulink.sdi.setSubPlotLayout

Set number of subplots

Syntax

```
Simulink.sdi.setSubPlotLayout(numRows,numColumns)
```

Description

`Simulink.sdi.setSubPlotLayout(numRows,numColumns)` sets the number of subplots in the Simulation Data Inspector in a grid layout using the number of rows and columns.

Examples

Change Subplot Layout

```
% Change subplot layout to 4 rows and 2 columns
Simulink.sdi.setSubPlotLayout(4,2);
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`numRows` — Number of rows

integer

The number of subplot rows, specified as an integer greater or equal to one and less than or equal to eight.

Example: 2

`numColumns` — Number of columns

integer

The number of subplot columns, specified as an integer greater or equal to one and less than or equal to eight.

Example: 3

See Also

`Simulink.sdi.view`

Introduced in R2016a

Simulink.sdi.setTableGrouping

Change signal grouping hierarchy of Runs pane

Syntax

```
Simulink.sdi.setTableGrouping
Simulink.sdi.setTableGrouping(group1,group2,group3)
```

Description

`Simulink.sdi.setTableGrouping` groups signals in the Runs pane as a flat list.

`Simulink.sdi.setTableGrouping(group1,group2,group3)` groups signals in the Runs pane in the order specified by each hierarchy group.

Examples

Group Data by Model and Then Data Hierarchy

```
Simulink.sdi.setTableGrouping('Subsystems','DataHierarchy');
```

- "Inspect and Compare Signal Data Programmatically"

Input Arguments

`group1,group2,group3` — Signal hierarchy

string

Signal hierarchy grouping, specified as a string. You can use one, two, or three grouping arguments. The signals are grouped according to the order in which they are specified: first by group one, then by group two, and lastly by group three.

- `'DataHierarchy'` — Group signals according to grouping, such as in buses and `Mux` blocks

- `'SubSystems'` — Group signals according to model subsystem hierarchy
- `'PhysmodHierarchy'` — Group logged signals according to Simscape block structure

Example: `'SubSystems','DataHierarchy','PhysmodHierarchy'`

See Also

`Simulink.sdi.setRunNamingRule`

Introduced in R2016a

Simulink.sdi.view

Open Simulation Data Inspector

Syntax

```
Simulink.sdi.view
```

Description

`Simulink.sdi.view` opens the Simulation Data Inspector.

Examples

Create and View a Run in the Simulation Data Inspector

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 Simulation Runs

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});

% Get and change one of the model parameters in the model workspace
mws = get_param('slexAircraftExample','modelworkspace');
wsMq = mws.evalin('Mq');
mws.assignin('Mq',3*wsMq);

% 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 in the Comparisons pane
Simulink.sdi.view(Simulink.sdi.GUITabType.CompareRuns);
```

- “Inspect and Compare Signal Data Programmatically”

Alternatives

To open the Simulation Data Inspector from the Simulink Editor toolbar, click the **Simulation Data Inspector** button  .

See Also

`Simulink.sdi.createRun`

Introduced in R2011b

Simulink.SimulationData.createStructOfTimeseries

Create a structure with MATLAB `timeseries` object leaf nodes

Syntax

```
struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(
tsArrayObject)

struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(
busObj,structOfTimeseries)

struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(
busObj,cellOfTimeseries)
struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(
busObj,cellOfTimeseries,dims)
```

Description

`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(tsArrayObject)` creates a structure of MATLAB `timeseries` objects from a `Simulink.TsArray` object. Use this syntax for signal logging data for a model simulated in a release earlier than R2016a that used `ModelDataLogs` signal logging format.

`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(busObj,structOfTimeseries)` creates a structure that matches the attributes of the bus object `busObj` and sets the values of structure leaf nodes using a structure of MATLAB `timeseries` objects `structOfTimeseries`. Use this syntax when using a partial structure as the basis for creating a full structure to load into a model.

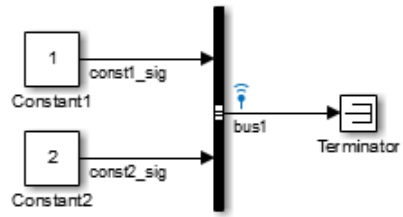
`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(busObj,cellOfTimeseries)` creates a structure that matches the attributes of the bus object `busObj` and sets the values of structure leaf nodes using a cell array of MATLAB `timeseries` objects `cellOfTimeseries`.

`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(busObj,cellOfTimeseries,dims)` creates a structure with the dimensions `dims`. Use this syntax to create a structure to load into an array of buses.

Examples

Structure Based on `Simulink.TsArray`

Suppose you had signal logging data from simulating a model in a release earlier than R2016a, using the `ModelDataLogs` format. The logged output is `logsout`.



View the logged data.

```
logsout
```

```
logsout =

Simulink.ModelDataLogs (log_modeldatalogs):
  Name                  Elements  Simulink Class

  bus1                     2      TsArray
```

Convert the logged data to a structure of MATLAB `timeseries` objects.

```
struct_of_ts = ...
Simulink.SimulationData.createStructOfTimeseries(logsout.bus1)
```

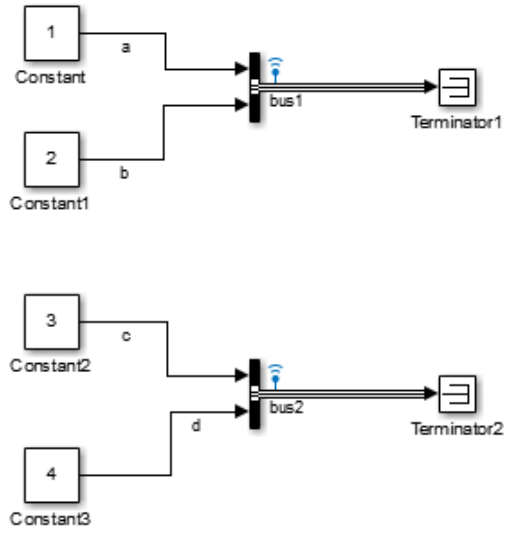
```
struct_of_ts =

    const1-sig: [1x1 timeseries]
    const2_sig: [1x1 timeseries]
```

Structure Based on Bus Object and a Partial Structure of Timeseries Data

Create a structure of MATLAB `timeseries` objects based on a `Simulink.Bus` object and a partial structure of MATLAB `timeseries` objects. Use this structure to load into another model. Open a model and simulate it, producing signal logging data.

Open a model and simulate it, producing signal logging data.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_log_structTimeSeries')))
sim('ex_log_structTimeSeries')
```



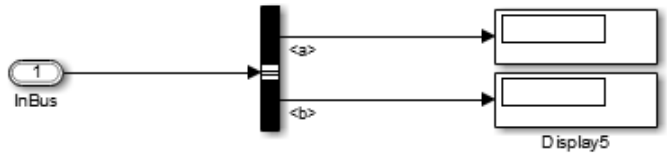
View the logged signal data.

```
ex_log_structTimeSeries_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 **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]
```

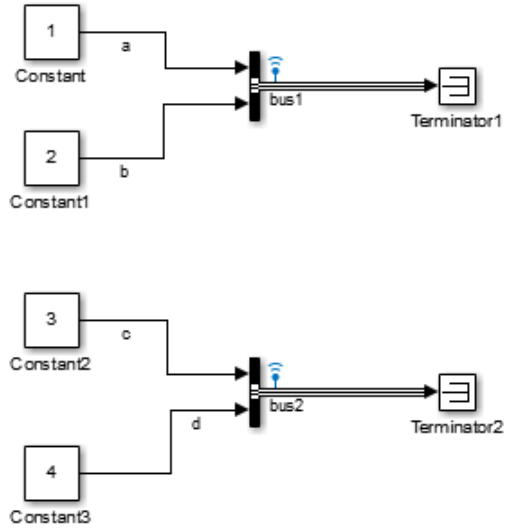
Structure to Use with an Array of Buses

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:



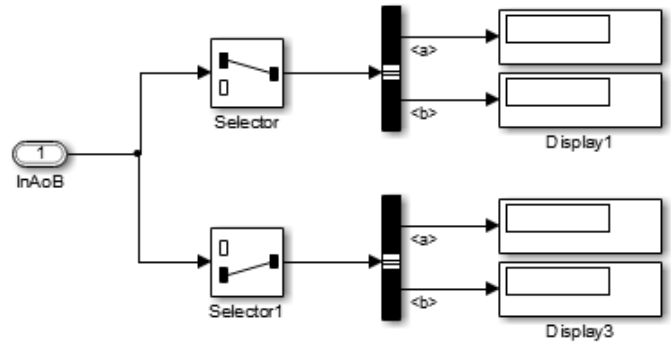
View the logged signal data.

```
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` — `Simulink.TsArray` object to convert

`Simulink.TsArray` object

`Simulink.TsArray` object to convert to a structure of MATLAB `timeseries` objects

In releases earlier than R2016a, when you log signals using the `ModelDataLogs` format, the logged data is a collection of `Simulink.TsArray` objects.

`busObj` — Bus object for creating a structure of MATLAB `timeseries` objects
`Simulink.Bus` object

Bus object for creating a structure of MATLAB `timeseries` objects, specified as the name of a `Simulink.Bus` object.

Data Types: `char`

`structOfTimeseries` — Structure object for values to override ground values, specified as a structure of MATLAB `timeseries` objects.
structure of MATLAB `timeseries` objects

Structure object for values to override ground values, specified as a structure of MATLAB `timeseries` objects. The structure must have the same hierarchy as the bus object. However, the names of the fields in the structure do not have to match the names of the corresponding bus object nodes.

Data Types: `struct`

`cellOfTimeseries` — Cell array objects for values to override ground values, specified as a cell array of MATLAB `timeseries` objects.
cell array of MATLAB `timeseries` objects

Cell array object for values to override ground values, specified as a cell array of MATLAB `timeseries` objects. If you specify a cell array of MATLAB `timeseries` objects and you specify a `dims` argument, then the length of the cell array must be equal to the result of `Simulink.BusObject.getNumLeafBusElements` times the product of the specified dimensions.

Data Types: `cell`

`dims` — Dimensions of the structure that this function creates.
vector

Dimensions of the structure that this function creates, specified as a vector. The length of the cell array is equal to the result of `Simulink.BusObject.getNumLeafBusElements` times the product of the specified dimensions.

If you specify a dimension in the form `[n]`, then Simulink interprets the dimension to be `1xn`.

Data Types: `double`

Output Arguments

`struct_of_ts` — Structure of MATLAB `timeseries` objects.
MATLAB structure

MATLAB `timeseries` objects, returned as a structure. The structure has the same hierarchy and attributes as the `Simulink.TsArray` object or `Simulink.Bus` object that you specify.

The dimensions of `structOfTimeseries` depend on the input arguments:

- If you specify `tsArrayObject`, then the dimension is 1.
- If you specify the `busObj` and a structure of MATLAB `timeseries`, then the dimension matches the dimensions of the specified structure.
- If you specify only the `busObj` and a cell array of MATLAB `timeseries`, then the dimension is 1.
- If you specify the `busObj` argument, a cell array of MATLAB `timeseries`, and the `dims` argument, then the dimensions match the dimensions of `dims`.

Related Links

`Simulink.Bus |Simulink.TsArray |Simulink.ModelDataLogs |Simulink.ModelDataLogs.convertToDataset`

Introduced in R2013a

Simulink.SimulationData.DatasetRef.getDatasetVariableNam

List names of `Dataset` variables in MAT-file

Syntax

```
varNames = Simulink.SimulationData.DatasetRef(matFile)
```

Description

`varNames = Simulink.SimulationData.DatasetRef(matFile)` lists the names of variables for `Dataset` data in a MAT-file.

Examples

List Variable Names in MAT-File

Suppose that you simulate a model using the default variable names for signal logging data and states data. You enable the **Configuration Parameters** > **Data Import/Export** > **Log Dataset data to file** and use the default MAT-file name of `out.mat`.

List the variable names in the MAT-file.

```
varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('out.mat')
```

```
varNames =

    'xout'
    'logsout'
```

- "Log Data Using Persistent Storage"
- "Access Logged Data from Persistent Storage"

Tips

To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who`, or `whos` functions.

Input Arguments

`matFile` — MAT-file that contains `Dataset` variables

string

MAT-file that contains `Dataset` variables, specified as a string. The string specifies the path to the MAT-file.

Output Arguments

`varNames` — Names of `Dataset` variables in MAT-file

cell array

Names of `Dataset` variables in MAT-file, returned as a cell array.

See Also

Simulink.SimulationData.DatasetRef | Simulink.SimulationData.Dataset

Introduced in R2016a

Simulink.SimulationData.signalLoggingSelector

Open Signal Logging Selector

Syntax

```
Simulink.SimulationData.signalLoggingSelector(modelName)
```

Description

`Simulink.SimulationData.signalLoggingSelector(modelName)` opens the Signal Logging Selector dialog box for the model that you specify with `modelName`.

Input Arguments

`modelName`

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')
```

More About

- "Override Signal Logging Settings"

See Also

Simulink.SimulationData.Dataset | `Simulink.ModelDataLogs`

Introduced in R2011a

setName

Class: Simulink.SimulationData.Unit
Package: Simulink.SimulationData

Specify name of logging data units

Syntax

```
unitObject = setName(unitObj,unitName)
```

Description

`unitObject = setName(unitObj,unitName)` sets the name for the `Simulink.SimulationData.Unit` object to the name specified in `unitName`.

Input Arguments

`unitObj` — Logging data unit object to name

`Simulink.SimulationData.Unit` object

Logging data unit object to name, specified as a `Simulink.SimulationData.Unit` object.

`unitName` — Name of logging data unit

string

Name of logging data unit, specified as a string.

Output Arguments

`name` — Name of logging data units

string

Name of logging data units, returned as a string.

Examples

Name a Logging Data Unit Object

```
inchesUnit = Simulink.SimulationData.Unit('in');
inchesUnit = setName(inchesUnit,'inches')
```

```
inchesUnit =

  Units with properties:

    Name: 'inches'
```

- "Log Signal Data That Uses Units"
- "Convert Logged Data to Dataset Format"
- "Prepare Model Inputs and Outputs"

See Also

Simulink.SimulationData.Unit

Introduced in R2011a

Simulink.SimulationData.updateDatasetFormatLogging

Convert model and its referenced models to use `Dataset` format for signal logging

Syntax

```
Simulink.SimulationData.updateDatasetFormatLogging(top_model)
Simulink.SimulationData.updateDatasetFormatLogging(top_model,
variants)
```

Description

Note: The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format. You do not need to use this command to update the signal logging format for a model that uses model referencing. Opening the model in R2016a or later uses `Dataset` format for all signal logging.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

`Simulink.SimulationData.updateDatasetFormatLogging(top_model)` converts the top-level model and all of its referenced models to use the `Dataset` format for signal logging instead of the `ModelDataLogs` format. You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If a Model block has the **Generate preprocessor conditionals** option selected, the function converts all the variants; otherwise, the function converts only the active variant.

`Simulink.SimulationData.updateDatasetFormatLogging(top_model, variants)` specifies which variant models to convert to use the `Dataset` signal logging format. For details about the `variants` argument, see "Input Arguments" on page 2-636

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.

More About

Dataset

The *`Dataset`* format causes Simulink to use a `Simulink.SimulationData.Dataset` object to store the logged signal data. The `Dataset` format use MATLAB `timeseries` objects to formatting the data.

ModelDataLogs

The *`ModelDataLogs`* format causes Simulink to use a `Simulink.ModelDataLogs` object to store the logged signal data.`Simulink.Timeseries` and `Simulink.TsArray` objects provide the format for the data.

Tips

- The conversion function sets the `SignalLoggingSaveFormat` parameter value to `Dataset` for all the updated models.
- If you want to save the format updates that the conversion function makes, then ensure that the top-level model, referenced models, and variant models are accessible and writable.
- If a model has no other unsaved changes, the conversion function saves the format updates to the model. If the model has unsaved changes, the function updates the format, but does not save those changes.
- If you use this function for a model that does not include any referenced models, the function converts the top-level model use the `Dataset` format.
- "Migrate Scripts That Use ModelDataLogs API"

See Also

Simulink.SimulationData.Dataset | `Simulink.ModelDataLogs` | `Simulink.ModelDataLogs.convertToDataset`

Introduced in R2011a

find

Class: Simulink.SimulationOutput
Package: Simulink

Access and display values of simulation results

Syntax

```
output = simOut.find('VarName')
```

Description

`output = simOut.find('VarName')` accepts one variable name. Specify *VarName* inside single quotes.

Input Arguments

`VarName`

Name of logged variable for which you seek values.

Default:

Output Arguments

`Value`

Value of the logged variable name specified in input.

Examples

Simulate `vdp` and store the values of the variable *youtNew* in *yout*.

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...
         'SaveState','on','StateSaveName','xoutNew',...
         'SaveOutput','on','OutputSaveName','youtNew');
yout = simOut.find('youtNew')
```

Alternatives

`Simulink.SimulationOutput.get`

See Also

Simulink.SimulationOutput.get | Simulink.SimulationOutput.who

get

Class: Simulink.SimulationOutput
Package: Simulink

Access and display values of simulation results

Syntax

```
output = simOut.get('VarName')
```

Description

`output = simOut.get('VarName')` accepts one variable name. Specify *VarName* inside single quotes.

Input Arguments

`VarName`

Name of logged variable for which you seek values.

Default:

Output Arguments

`Value`

Value of the logged variable name specified in input.

Examples

Simulate `vdp` and store the values of the variable *youtNew* in *yout*.

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...
         'SaveState','on','StateSaveName','xoutNew',...
         'SaveOutput','on','OutputSaveName','youtNew');
yout = simOut.get('youtNew')
```

Alternatives

`Simulink.SimulationOutput.find`

See Also

Simulink.SimulationOutput.find | Simulink.SimulationOutput.who

getSimulationMetadata

Class: Simulink.SimulationOutput
Package: Simulink

Return `SimulationMetadata` object for simulation

Syntax

```
mData = simout.getSimulationMetadata()
```

Description

`mData = simout.getSimulationMetadata()` retrieves metadata information in a `SimulationMetadata` object from the `simout` `SimulationOutput` object.

Input Arguments

`simout` — Simulation object to get metadata from

object

Simulation object to get metadata from, specified as a `SimulationOutput` object.

Output Arguments

`mData` — `SimulationMetadata` object stored in the `simout` `SimulationOutput` object

object

`SimulationMetadata` object stored in the `simout` `SimulationOutput` object, returned as an object.

Examples

Retrieve Metadata From `vdp` Simulation

Simulate the `vdp` model and retrieve metadata information from the simulation.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`

```
open_system('vdp')
simout = sim(bdroot,'ReturnWorkspaceOutputs','on');
```

Retrieve metadata information about this simulation using `mData`.

```
mData=simout.getSimulationMetadata()
SimulationMetadata with properties:
    ModelInfo: [1x1 struct]
   TimingInfo: [1x1 struct]
   UserString: ''
     UserData: []
```

See Also

Simulink.SimulationMetadata | Simulink.SimulationOutput.setUserData | Simulink.SimulationOutput.setUserString

setUserData

Class: Simulink.SimulationOutput
Package: Simulink

Store custom data in `SimulationMetadata` object that `SimulationOutput` object contains

Syntax

```
simoutNew = simout.setUserData(CustomData)
```

Description

`simoutNew = simout.setUserData(CustomData)` assigns a copy of the `simout` `SimulationOutput` object to `simoutNew`. The copy contains `CustomData` in its `SimulationMetadata` object.

Input Arguments

`simout` — Simulation object to get metadata from

object

Simulation object to get metadata from, specified as a `SimulationOutput` object.

`CustomData` — Data to store in a metadata object

data

Any custom data you want to store in the metadata object.

Output Arguments

`simoutNew` — Simulation object that stores metadata object with custom data

object

A copy of the `simout` `SimulationOutput` object that contains `CustomData` in its `SimulationMetadata` object, returned as an object.

Examples

Store Data in `SimulationMetadata` Object of `vdp` Simulation

Simulate the `vdp` model. Store custom data in the `SimulationMetadata` object that the `SimulationOutput` object contains.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.

```
open_system('vdp')
simout=sim(bdroot,'ReturnWorkspaceOutputs','on');
```

Store custom data about the simulation in the `SimulationMetadata` object that `simout` contains.

```
simout=simout.setUserData(struct('param1','value1','param2','value2','param3','value3')
```

Use `SimulationOutput.getSimulationMetadata` to retrieve the information you stored.

```
mData=simout.getSimulationMetadata();
disp(mData.UserData)
```

```
  param1: 'value1'
  param2: 'value2'
  param3: 'value3'
```

See Also

Simulink.SimulationMetadata | Simulink.SimulationOutput.getSimulationMetadata | Simulink.SimulationOutput.setUserString

setUserString

Class: Simulink.SimulationOutput
Package: Simulink

Store custom string in `SimulationMetadata` object that `SimulationOutput` object contains

Syntax

```
simoutNew = simout.setUserString(CustomString)
```

Description

`simoutNew = simout.setUserString(CustomString)` assigns a copy of the `simout` `SimulationOutput` object to `simoutNew`. The copy contains `CustomString` in its `SimulationMetadata` object.

Input Arguments

`simout` — Simulation object to get metadata from

object

Simulation object to get metadata from, specified as a `SimulationOutput` object.

`CustomString` — String to store in a metadata object

string

Any custom string you want to store in the metadata object.

Output Arguments

`simoutNew` — Simulation object that stores metadata object with custom string

object

A copy of the `simout` `SimulationOutput` object that contains `CustomString` in its `SimulationMetadata` object, returned as an object.

Examples

Store a String in `SimulationMetadata` Object of `vdp` Simulation

Simulate the `vdp` model. Store a custom string in the `SimulationMetadata` object that the `SimulationOutput` object contains.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.

```
open_system('vdp')
simout=sim(bdroot,'ReturnWorkspaceOutputs','on');
```

Store a string to describe the simulation.

```
simout=simout.setUserString('First Simulation');
```

Use `SimulationOutput.getSimulationMetadata` to retrieve the information you stored.

```
mData=simout.getSimulationMetadata();
disp(mData.UserString)
```

```
   First Simulation
```

See Also

Simulink.SimulationMetadata | Simulink.SimulationOutput.getSimulationMetadata | Simulink.SimulationOutput.setUserData

who

Class: Simulink.SimulationOutput
Package: Simulink

Access and display output variable names of simulation

Syntax

```
simOutVar = simOut.who
```

Description

`simOutVar = simOut.who` returns the names of all simulation output variables, including workspace variables.

Output Arguments

`simOutVar`

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.SubSystem.convertToModelReference

Convert subsystem to model reference

Syntax

```
Simulink.SubSystem.convertToModelReference(gcb,'UseConversionAdvisor',true)

[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(
subsys,mdlRefs)
[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(
subsys,mdlRefs,Name,Value)
```

Description

`Simulink.SubSystem.convertToModelReference(gcb,'UseConversionAdvisor',true)` opens the Model Reference Conversion Advisor for the currently selected subsystem block.

`[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference( subsys,mdlRefs)` converts the specified subsystems to referenced models using the `mdlRefs` value.

For each subsystem that the function converts, it:

- Creates a model
- Copies the contents of the subsystem into the new model
- Updates any root-level Inport, Outport, Trigger, and Enable blocks and the configuration parameters of the model to match the compiled attributes of the original subsystem
- Copies the contents of the model workspace of the original model to the new model

Before you use the function, load the model containing the subsystem.

`[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference( subsys,mdlRefs,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Open the Model Reference Conversion Advisor

Open the `f14` model.

```
open_system('f14');
```

In the `f14` model, select the Controller subsystem output signal, click the **Simulation Data Inspector** button arrow  , and select **Log Selected Signals to Workspace**.

In the Simulink Editor, select the `Controller` subsystem. Then open the Model Reference Conversion Advisor from the command line.

```
Simulink.SubSystem.convertToModelReference(gcb,'UseConversionAdvisor',true);
```

Perform the conversion using the advisor.

Convert Subsystem to Referenced Model

Convert the `Bus Counter` subsystem to a referenced model named `bus_counter_ref_model`.

```
open_system('sldemo_mdlref_conversion');
Simulink.SubSystem.convertToModelReference(...
   'sldemo_mdlref_conversion/Bus Counter', ...
   'bus_counter_ref_model', ...
   'AutoFix',true,...
   'ReplaceSubsystem',true,...
   'CheckSimulationResults',true);
```

```
success =

     1

>> mr_handle

mr_handle =

   79.0018
```

Convert Multiple Subsystems to Referenced Models

Convert the two subsystems with one command.

```
open_system('f14');
Simulink.SubSystem.convertToModelReference(...
{'f14/Controller','f14/Aircraft Dynamics Model'},...
{'controller_ref_model','aircraft_dynamics_ref_model'},...
'ReplaceSubsystem',true,...
'CheckSimulationResults',true)
```

- sldemo_mdlref_conversion
- “Convert a Subsystem to a Referenced Model”
- “Model Reference”

Input Arguments

`subsys` — Subsystems to convert

string | subsystem handle | cell array of strings | array of subsystem handles

Subsystems to convert, specified as a string, subsystem handle, or cell array of stings or array of subsystem handles.

Each subsystem must be one of these kinds of subsystem:

- Atomic
- Function-call
- Triggered
- Enabled

The **Configuration Parameters > Diagnostics > Data Validity > Signal resolution** parameter must be set to `Explicit only`. To do set that parameter programmatically, use this command, substituting your model name for `myModelName`.

```
set_param(myModellName,'SignalResolutionControl','UseLocalSettings');
```

Data Types: `double`

`mdlRefs` — Referenced model names

string | cell array of strings

Referenced model names, specified as a string or cell array of strings. Each model name must be 59 characters or less.

If you specify a cell array of subsystems to convert, specify a cell array of referenced model names. Each model name corresponds to the specified subsystem, in the same order.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`Simulink.SubSystem.convertToModelReference...(engineSubsys,engineModelRef,'Rep`

`'AutoFix'` — Fix all conversion issues that can be fixed automatically

`false` (default) | `true`

If you set `AutoFix` to `true`, the function fixes all conversion issues that it can fix.

For issues that the function cannot fix, the conversion process generates error messages that you address by modifying the model.

Note: If you set `'Force'` to `true`, then the function does not automatically fix conversion issues.

Data Types: `logical`

`'Force'` — Complete conversion even with errors

`false` (default) | `true`

If you set `'Force'` to `true`, the function returns conversion errors as warnings and continues with the conversion without fixing the errors. This option allows you to use the function to do the initial steps of conversion and then complete the conversion process yourself.

If you set `Force` to `true`, then the function does not fix conversion issues, even if you set `'AutoFix'` to `true`. However, the `success` output argument is `true`, regardless of whether any conversion errors occurred.

`'CheckSimulationResults'` — Compare simulation results before and after conversion

`false` (default) | `true`

Compare simulation results before and after conversion, specified as `true` or `false`.

Before performing the conversion, enable signal logging for the subsystem output signals of interest in the model.

For the `Simulink.SubSystem.convertToModelReference` command, set:

- `'CheckSimulationResults'` to `true`
- `'AbsoluteTolerance'`
- `'RelativeTolerance'`
- `'SimulationModes'` to the same as the simulation mode as in the original model

If the difference between simulation results exceeds the tolerance level, the function displays a message.

`'AbsoluteTolerance'` — Absolute signal tolerance for comparison

`'1e-06'` (default) | double

Absolute signal tolerance for comparison, specified as a double. Use the option only if you set `CheckSimulationResults` to `true`.

Data Types: `double`

`'RelativeTolerance'` — Relative signal tolerance for comparison

`'1e-06'` (default) | double

Relative signal tolerance for comparison, specified as a double. Use the option only if you set `CheckSimulationResults` to `true`.

Data Types: `double`

`'DataFileName'` — Name of file for storing conversion data

string

Name of file for storing conversion data, specified as a string. You can specify an absolute or relative path.

You can save the conversion data in a MAT-file (default) or a MATLAB file. If you use a `.m` file extension, the function serializes all variables to a MATLAB file.

By default, the function uses a file name consisting of the model name plus `_conversion_data.mat`.

`'ReplaceSubsystem'` — Replace content of each subsystem with Model blocks

`false` (default) | `true`

Replace subsystem blocks with Model blocks, specified as `true` or `false`. The Model block references the referenced model.

By default, the function displays the referenced models in separate Simulink Editor windows.

If you set the value to `true`, consider making a backup of the original model before you convert the subsystems. If you want to undo the conversion, having a backup makes it easier to restore the model.

If you set `ReplaceSubsystem` to `true`, the conversion action depends on whether you use the automatic fix options.

- If you use the automatic fixes, then the conversion replaces the Subsystem block with a Model block unless the automatic fixes change the input or output ports. If the ports change, then the conversion includes the contents of the subsystem in a Model block that is inserted in the Subsystem block.
- If you do not use the automatic fixes, then the conversion replaces the Subsystem block with a Model block.

Data Types: `logical`

`'CreateWrapperSubsystem'` — Insert wrapper subsystem to preserve model layout

`false` (default) | `true`

Insert wrapper subsystem to preserve model layout, specified as `true` or `false`. When you convert a subsystem to a referenced model, you can have the conversion process insert a wrapper subsystem to preserve the layout of a model. The subsystem wrapper contains the Model block from the conversion.

If the automatic fixes change the subsystem input or output ports, the conversion process enables this option.

Data Types: `logical`

`'SimulationModes'` — Simulation mode for Model blocks

`'Normal'` (default) | `'Accelerator'`

Simulation mode for Model blocks, specified as a `'Normal'` or `'Accelerator'`. The simulation mode setting applies to the Model blocks that reference the models that the conversion creates.

`'BuildTarget'` — Model reference targets to generate
`'Sim' | 'RTW'`

Model reference targets to generate, specified as a string.

- `'Sim'` — Model reference simulation target
- `'RTW'` — Code generation target

Output Arguments

`success` — Conversion status
`1 | 0`

Conversion status. A value of `1` indicates a successful conversion.

If you set `'Force'` to `true`, the function returns a value of `1` if the conversion completes. However, the simulation results can differ from the simulation results for the model before conversion.

`mdlRefBlkHs` — Handles of created Model blocks
handle of Model block | array of handles of Model blocks

Handles of created Model blocks, returned as a double or cell array.

Data Types: `double`

More About

Tips

- Specifying multiple subsystems to convert with one command can save time, compared to converting each subsystem separately. The multiple-subsystem conversion process compiles the model one time.
- If you specify multiple subsystems to convert, the conversion process attempts to convert each subsystem. Successfully converted subsystems are produce referenced models, even if the conversions of other subsystems fail.

- If you specify multiple subsystems, consider:
  - Specifying these `'Autofix'`, `'ReplaceSubsystem'`, `'Verify Simulation Results'` name and value pairs, set to `true`.
  - In the model, setting a short simulation time.
- Simulink uses the data dictionary to save the bus objects that it creates as part of the conversion processing when both these conditions exist:
  - The top model uses a data dictionary.
  - All changes to the top model are saved.
- After you complete the conversion, update the model as necessary to meet your modeling requirements. For details, see "Integrate the Referenced Model into the Parent Model".
- Converting a masked subsystem can require you to perform additional tasks to maintain the same general behavior that the masked subsystem provided.

  If the subsystem that you convert contains a masked block, consider masking the Model block in your new referenced model (see "Block Masks"). Configure the referenced model to support the functionality of the masked subsystem.

  Note: A referenced model does not support the functionality that you can achieve with mask initialization code to create masked parameters.

  For masked parameters:

  1. In the model workspace of the referenced model, create a variable for each masked parameter.
  2. In the Model Explorer, select the `Model Workspace` node. In the Dialog pane, in the **Model arguments** parameter, enter the variables that you want for model arguments.
  3. In the new Model block, for the **Model arguments** parameter, specify the values for the model arguments.

  For masked callbacks, icons, ports, and documentation:

  1. In the backup copy, open the Mask Editor on the masked subsystem and copy the content you want into the masked Model block.

2 In the Mask Editor for the new Model block, paste the masked subsystem content.

- "Subsystem to Model Reference Conversion"

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.Bus.save` | `Simulink.SubSystem.copyContentsToBlockDiagram`

Introduced in R2006a

Simulink.SubSystem.copyContentsToBlockDiagram

Copy contents of subsystem to empty block diagram

Syntax

```
Simulink.SubSystem.copyContentsToBlockDiagram(subsys, bdiag)
```

Description

`Simulink.SubSystem.copyContentsToBlockDiagram(`*`subsys`*`, `*`bdiag`*`)` copies the contents of the subsystem *`subsys`* to the block diagram *`bdiag`*. The subsystem and block diagram must have already been loaded. The subsystem cannot be part of the block diagram. The function affects only blocks, lines, and annotations; it does not affect nongraphical information such as configuration sets.

This function cannot be used if the destination block diagram contains any blocks or signals. Other types of information can exist in the destination block diagram and are unaffected by the function. Use `Simulink.BlockDiagram.deleteContents` if necessary to empty the block diagram before using `Simulink.SubSystem.copyContentsToBlockDiagram`.

Tip To flatten a model hierarchy by expanding the contents of a subsystem to the system that contains that subsystem, do not use the `Simulink.SubSystem.copyContentsToBlockDiagram` function. Instead, expand the subsystem, as described in "Expand Subsystem Contents".

Input Arguments

`subsys`

Subsystem name or handle

`bdiag`

Block diagram name or handle

Examples

Copy the graphical contents of `f14/Controller`, including all nested subsystems, to a new block diagram:

```
% open f14
open_system('f14');

% create a new model
newbd = new_system;
open_system(newbd);

% copy the subsystem
Simulink.SubSystem.copyContentsToBlockDiagram('f14/Controller', newbd);

% close f14 and the new model
close_system('f14', 0);
close_system(newbd, 0);
```

More About

- "Modeling Fundamentals"
- "Create a Subsystem"
- "Expand Subsystem Contents"

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.BlockDiagram.deleteContents` | `Simulink.SubSystem.convertToModelReference` | `Simulink.SubSystem.deleteContents`

Introduced in R2007a

Simulink.SubSystem.deleteContents

Delete contents of subsystem

Syntax

```
Simulink.SubSystem.deleteContents(subsys)
```

Description

`Simulink.SubSystem.deleteContents(subsys)` deletes the contents of the subsystem *`subsys`*. The function affects only blocks, lines, and annotations. The subsystem must have already been loaded.

Note: This function does not delete library blocks in a subsystem.

Input Arguments

`subsys`

Subsystem name or handle

Examples

Delete the graphical contents of `Controller`, including all nested subsystems:

```
Simulink.SubSystem.deleteContents('f14/Controller');
```

More About

- "Model Hierarchy"
- "Create a Subsystem"

See Also

`Simulink.BlockDiagram.copyContentsToSubSystem` | `Simulink.BlockDiagram.deleteContents` | `Simulink.SubSystem.convertToModelReference` | `Simulink.SubSystem.copyContentsToBlockDiagram`

Introduced in R2007a

Simulink.SubSystem.getChecksum

Return checksum of nonvirtual subsystem

Syntax

```
[checksum,details] = Simulink.SubSystem.getChecksum(subsys)
```

Description

`[checksum,details] = Simulink.SubSystem.getChecksum(subsys)` returns the checksum of the specified nonvirtual subsystem. Simulink computes the checksum based on the subsystem parameter settings and the blocks the subsystem contains. Virtual subsystems do not have checksums.

One use of this command is to determine why code generated for a subsystem is not being reused. 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`

Introduced in R2006b

sint

Create `Simulink.NumericType` object describing signed integer data type

Syntax

```
a = sint(WordLength)
```

Description

`sint(WordLength)` returns a `Simulink.NumericType` object that describes the data type of a signed integer with a word size given by *`WordLength`*.

Note: `sint` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `sint(WordLength)` with `fixdt(1,WordLength,0)`.

Examples

Define a 16-bit signed integer data type.

```
a = sint(16)
```

```
a =

  NumericType with properties:

      DataTypeMode: 'Fixed-point: binary point scaling'
        Signedness: 'Signed'
        WordLength: 16
    FractionLength: 0
           IsAlias: 0
         DataScope: 'Auto'
        HeaderFile: ''
       Description: ''
```

See Also

`fixdt` | `Simulink.NumericType` | `float` | `sfix` | `sfrac` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

slbuild

Build standalone executable or model reference target for model

Syntax

```
slbuild(model)
slbuild(model,'StandaloneRTWTarget')
slbuild(model,'StandaloneRTWTarget','ForceTopModelBuild',true)

slbuild(model,'CleanTopModel')

slbuild(model,mdlreftarget)
slbuild(model,mdlreftarget,
'UpdateThisModelReferenceTarget',buildcond)
```

Description

Note: Except where noted, this function requires a Simulink Coder license.

`slbuild(model)` builds a standalone Simulink Coder target executable from `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 `model` (same as previous).

`slbuild(model,'StandaloneRTWTarget','ForceTopModelBuild',true)` allows you to additionally force regeneration of code for the top model of a system that includes referenced models. If `ForceTopModelBuild` is omitted or set to `false`, the build process determines whether to regenerate top model code based on model and model parameter changes.

`slbuild(model,'CleanTopModel')` cleans the model build area enough to trigger regeneration of the top model code at the next build.

Note: The following function calls for building the model reference target:

- Honor the setting of the **Rebuild** parameter on the **Model Referencing** pane of the Configuration Parameters dialog box.
- Require a Simulink Coder license only if you build a model reference Simulink Coder target, not if you build only a model reference simulation target.

`slbuild(model,mdlreftarget)` builds a model reference target, of the type specified by `mdlreftarget`, from `model`. The `mdlreftarget` argument must be one of the following:

- `'ModelReferenceSimTarget'` — Builds a model reference simulation target (does not require a Simulink Coder license)
- `'ModelReferenceRTWTarget'` — Builds a model reference Simulink Coder target and the corresponding model reference simulation target
- `'ModelReferenceRTWTargetOnly'` — Builds only a model reference Simulink Coder target

`slbuild(model,mdlreftarget, 'UpdateThisModelReferenceTarget',buildcond)` allows you to specify a conditional rebuild option for the model reference target build when the **Rebuild** parameter on the **Model Referencing** pane of the Configuration Parameters dialog box is set to `Never`.

Note: The `'UpdateThisModelReferenceTarget'` setting applies only to `model`, not to any models referenced by `model`.

The `buildcond` argument must be one of the following:

- `'Force'`

  Unconditionally rebuilds the model. This option is equivalent to the `Always` rebuild option on the **Model Referencing** pane of the Configuration Parameters dialog box.
- `'IfOutOfDateOrStructuralChange'`

Rebuilds the model if the build process detects any changes. This option is equivalent to the `If any changes detected` rebuild option on the **Model Referencing** pane of the Configuration Parameters dialog box.

- `'IfOutOfDate'`

  Rebuilds the model if the build process detects any changes in known dependencies of this model. This option is equivalent to the `If any changes in known dependencies detected` rebuild option on the **Model Referencing** pane of the Configuration Parameters dialog box.

Note: You cannot use `slbuild` to build subsystems.

Examples

Generate Code and Build Executable Image for Model

Generate C code for model `rtwdemo_rtwintro`.

```
slbuild('rtwdemo_rtwintro')
```

For the GRT target, the coder generates the following code files and places them in folders `rtwdemo_rtwintro_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Files	Interface Files	Other Files

`rtwdemo_rtwintro.c`
`rtwdemo_rtwintro.h`
`rtwdemo_rtwintro_pri`
`rtwdemo_rtwintrotype`
	`rtwtypes.h`
	`multiword_types.`
	`builtin_typeid_t`
		`rtmodel.h`	none

If the following model configuration parameters settings apply, the coder generates additional results.

Parameter Setting	Results
Code Generation > **Generate code only** pane is cleared	Executable image `rtwdemo_rtwintro.exe`

Parameter Setting	Results
Code Generation > Report > Create code generation report is selected	Report appears, providing information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Force Top Model Build

Generate code and build an executable image for `rtwdemo_mdlreftop`, which refers to model `rtwdemo_mdlrefbot`, regardless of model checksums and parameter settings.

```
slbuild('rtwdemo_mdlreftop','StandaloneRTWTarget','ForceTopModelBuild',true)
```

Clean Top Model Build

Clean the model build area enough to trigger regeneration of the top model code at the next build.

```
slbuild('rtwdemo_rtwintro','CleanTopModel')
```

Input Arguments

`model` — Model for which to run the build process

handle | name

Model for which to build a standalone executable or model reference target, specified as a handle or a string representing the model name.

Example: `gcs`

`mdlreftarget` — Model reference target

`'ModelReferenceSimTarget' | 'ModelReferenceRTWTarget' | 'ModelReferenceRTWTargetOnly'`

Model reference target to use for generating model code

Example: `'ModelReferenceSimTarget'`

More About

- "Model Reference Simulation Targets"

- “What Is Acceleration?”
- “Perform Acceleration”

See Also

`rtwbuild` | `rtwrebuild`

Introduced before R2006a

slCharacterEncoding

Change MATLAB character set encoding

Syntax

```
currentCharacterEncoding = slCharacterEncoding()
slCharacterEncoding(encoding)
```

Description

This command allows you to change the current MATLAB character set encoding to be compatible with the character encoding of a model that you want to open.

`currentCharacterEncoding = slCharacterEncoding()` returns the current MATLAB character set encoding.

`slCharacterEncoding(encoding)` changes the MATLAB character set encoding to the specified `encoding`. You should only specify these values:

- `'US-ASCII'`
- `'Windows-1252'`
- `'ISO-8859-1'`
- `'Shift_JIS'`
- `'UTF-8'`

If you want to use a different character encoding, you need to start MATLAB with the appropriate locale settings for your operating system. Consult your operating system manual to change the locale setting. Simulink can support any character encoding that uses single-byte or double-byte characters.

If you open a model that uses a particular character set encoding in a MATLAB session that uses a different encoding, a warning appears. For example, suppose that you create a model in a MATLAB session configured for `Shift_JIS` and open it in a session configured for `US_ASCII`. The warning message shows the encoding of the current session and the encoding used to create the model. If you encounter any problems with

corrupted characters, for example when using MATLAB files associated with the model, then try using the `slCharacterEncoding` function to change the character encoding

1. Close all open models.
2. Use `slCharacterEncoding` to change the character encoding of the current MATLAB session to match the model character encoding.
3. Reopen the model.

Note You must close all open models or libraries before changing the MATLAB character set encoding except when changing from `'US-ASCII'` to another encoding.

More About

- "Open a Model with Different Character Encoding"
- "Save Models with Different Character Encodings"

Introduced before R2006a

sldebug

Start simulation in debug mode

Syntax

```
sldebug('sys')
```

Description

`sldebug('sys')` starts a simulation in debug mode. See “Debugger Command-Line Interface” for information about using the debugger.

Examples

The following command:

```
sldebug('vdp')
```

loads the Simulink example model `vdp` into memory and starts the simulation in debug mode. Alternatively, you can achieve the same result by using the `sim` command:

```
sim('vdp', 'debug', 'on')
```

See Also

`sim`

Introduced in R2006a

sldiagnostics

Display diagnostic information about Simulink system

Syntax

```
sldiagnostics('sys')
[txtRpt, sRpt] = sldiagnostics('sys')
[txtRpt, sRpt] = sldiagnostics('sys', options)
[txtRpt, sRpt] = sldiagnostics('sys', 'CompileStats')
[txtRpt, sRpt] = sldiagnostics('sys', 'RTWBuildStats')
```

Description

`sldiagnostics('sys')` displays the following diagnostic information associated with the model or subsystem specified by `sys`:

- Number of each type of block
- Number of each type of Stateflow object
- Number of states, outputs, inputs, and sample times of the root model.
- Names of libraries referenced and instances of the referenced blocks
- Time and additional memory used for each compilation phase of the root model

If the model specified by `sys` is not loaded, then `sldiagnostics` loads the model before performing the analysis.

The command `sldiagnostics('sys', options)` displays only the diagnostic information associated with the specific operations listed as *`options`* 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.
`RTWBuildStats`	root model	Lists the same information as the `CompileStats` diagnostic. When issued with the second output argument `sRpt`, it captures the same statistics included in `CompileStats` and also the Simulink Coder build statistics. You must explicitly specify this option, because it is not part of the default analysis.
`All`	not applicable	Performs all diagnostics.

Note: Running the `CompileStats` diagnostic before simulating a model for the first time will show greater memory usage. However, subsequent runs of the `CompileStats` diagnostic on the model will require less memory usage.

`[txtRpt, sRpt] = sldiagnostics('sys')` returns the diagnostic information as a textual report `txtRpt` and a structure array `sRpt`, which contains the following fields that correspond to the diagnostic options:

- `blocks`
- `stateflow`
- `sizes`
- `links`
- `compilestats`

`[txtRpt, sRpt] = sldiagnostics('sys', options)` returns only the specified options. If your chosen options specify just one type of analysis, then `sRpt` contains the results of only that analysis.

`[txtRpt, sRpt] = sldiagnostics('sys', 'CompileStats')` returns information on time and memory usage in `txtRpt` and `sRpt`.

`[txtRpt, sRpt] = sldiagnostics('sys', 'RTWBuildStats')` includes Simulink Coder build statistics in addition to the information reported for `CompileStats` in the `sRpt` output.

- `txtRpt` contains the formatted textual output of time spent in each of the phases in Simulink and Simulink Coder (if you specified `RTWBuildStats`), for example:

  ```
  Compile Statistics For: rtwdemo_counter
   Cstat1: 0.00 seconds Model compilation pre-start
   Cstat2: 0.00 seconds Stateflow compile pre-start notification
   Cstat3: 0.10 seconds Post pre-comp-start engine event
   Cstat4: 10.00 seconds Stateflow compile start notification
   Cstat5: 0.00 seconds Model compilation startup completed
  ```

- `sRpt` is a MATLAB structure containing time and memory usage for each of the phases, for example:

  ```
  sRpt =
  Model: ‘myModel1’
  ```

```
Statistics: [1x134 struct]
```

The size of the `sRpt.Statistics` array indicates the number of compile and build phases executed during the operation. Examine the Statistics fields:

```
sRpt.Statistics(1) =
Description: 'Phase1'
CPUTime: 7.2490
WallClockTime 4.0092
ProcessMemUsage: 26.2148
ProcessMemUsagePeak: 28.6680
ProcessVMSize: 15.9531
```

`CPUTime` and `WallClockTime` show the elapsed time for the phase in seconds. `ProcessMemUsage`, `ProcessMemUsagePeak` and `ProcessVMSize` show the memory consumption during execution of the phase in MB.

Examine these key metrics to understand the performance:

- `WallClockTime`—The real-time elapsed in each phase in seconds. Sum the `WallClockTime` in each phase to get the total time taken to perform the operation:

  ```
  ElapsedTime = sum([statRpt.Statistics(:).WallClockTime]);
  ```

- `ProcessMemUsage`—The amount of memory consumed in each phase. Sum the `ProcessMemUsage` across all the phases to get the memory consumption during the entire operation:

  ```
  TotalMemory = sum([statRpt.Statistics(:).ProcessMemUsage]);
  ```

- `ProcessMemUsagePeak`—The maximum amount of allocated memory in each phase. Get the maximum of this metric across all the phases to find the peak memory allocation during the operation:

  ```
  PeakMemory = max([statRpt.Statistics(:).ProcessMemUsagePeak]);
  ```

Note: Memory statistics are available only on the Microsoft Windows platform.

Examples

The following command counts and lists each type of block used in the `sldemo_bounce` model that comes with Simulink software.

```
sldiagnostics('sldemo_bounce', 'CountBlocks')
```

The following command counts and lists both the unique blocks and Stateflow objects used in the `sf_boiler` model that comes with Stateflow software; the textual report returned is captured as `myReport`.

```
myReport = sldiagnostics('sf_boiler', 'CountBlocks', 'CountSF')
```

The following commands open the `f14` model that comes with Simulink software, and counts the number of blocks used in the `Controller` subsystem.

```
sldiagnostics('f14/Controller', 'CountBlocks')
```

The following command runs the `Sizes` and `CompileStats` diagnostics on the `f14` model, capturing the results as both a textual report and structure array.

```
[txtRpt, sRpt] = sldiagnostics('f14', 'Sizes', 'CompileStats')
```

See Also

`find_system` | `get_param`

Introduced in R2006a

sldiagviewer.diary

Log simulation warnings and errors and build information to file

Syntax

```
sldiagviewer.diary
sldiagviewer.diary(filename)
sldiagviewer.diary(toggle)
sldiagviewer.diary(filename,'UTF-8')
```

Description

`sldiagviewer.diary` intercepts build information, warnings, and errors transmitted to the Command Window or the Diagnostic Viewer and logs them to a text file `diary.txt` in the current folder.

`sldiagviewer.diary(filename)` toggles the logging state of the text file specified by `filename`.

`sldiagviewer.diary(toggle)` turns logging to the log file on or off. The setting applies to the last file name you specified for logging or to `diary.txt` if you did not specify a file name.

`sldiagviewer.diary(filename,'UTF-8')` specifies the character encoding for the log file `filename`.

Examples

Log Build Information and Simulation Warnings and Errors

Start logging build information and simulation warnings and errors to `diary.txt`.

```
sldiagviewer.diary
open_system('vdp')
rtwbuild('vdp')
```

Open `diary.txt` to view logs.

```
### Starting build procedure for model: vdp
### Build procedure for model: 'vdp' aborted due to an error.
...
```

Log to Specific File

Set up logging to a file.

```
sldiagviewer.diary('C:\MyLogs\log1.txt')
```

Toggle File Logging State

Switch the logging state of a file.

```
sldiagviewer.diary('C:\MyLogs\log1.txt') % Start logging
open_system('vdp')
rtwbuild('vdp')

sldiagviewer.diary('off') % Switch off logging
open_system('sldemo_fuelsys')
rtwbuild('sldemo_fuelsys')

sldiagviewer.diary('on') % Resume logging
```

Specify Log File Name and Character Encoding

Set the file name to log to and the character encoding to use.

```
sldiagviewer.diary('C:\MyLogs\log1.txt','UTF-8')
```

- "View Diagnostics"
- "Customize Diagnostic Messages"

Input Arguments

`toggle` — Logging state

`'off' | 'on'`

Logging state, specified as `'on'` or `'off'`.

Example: `sldiagviewer.diary('on')`

`filename` — Name of file to log data to

string

Name of file to log data to, specified as a string.

Example: `sldiagviewer.diary('C:\Simulations\mySimulationDiary.txt')`

Introduced in R2014a

sldiscmdl

Discretize model that contains continuous blocks

Syntax

```
sldiscmdl('model_name',sample_time)
sldiscmdl('model_name',sample_time,method)
sldiscmdl('model_name',sample_time,options)
sldiscmdl('model_name',sample_time,method,freq)
sldiscmdl('model_name',sample_time,method,options)
sldiscmdl('model_name',sample_time,method,freq,options)
[old_blks,new_blks] =
sldiscmdl('model_name',sample_time,method,freq,options)
```

Description

`sldiscmdl('model_name',sample_time)` discretizes the model named `'model_name'` using the specified `sample_time`. The model does not need to be open, and the units for `sample_time` are simulation seconds.

`sldiscmdl('model_name',sample_time,method)` discretizes the model using `sample_time` and the transform method specified by `method`.

`sldiscmdl('model_name',sample_time,options)` discretizes the model using `sample_time` and criteria specified by the `options` cell array. This array consists of four elements: {`target`, `replace_with`, `put_into`, `prompt`}.

`sldiscmdl('model_name',sample_time,method,freq)` discretizes the model using `sample_time`, `method`, and the critical frequency specified by `freq`. The units for `freq` are Hz. When you specify `freq`, `method` must be `'prewarp'`.

`sldiscmdl('model_name',sample_time,method,options)` discretizes the model using `sample_time`, `method`, and `options`.

`sldiscmdl('model_name',sample_time,method,freq,options)` discretizes the model using `sample_time`, `method`, `freq`, and `options`. When you specify `freq`, `method` must be `'prewarp'`.

`[old_blks,new_blks] = sldiscmdl('model_name',sample_time,method,freq,options)` discretizes the model using *`sample_time`*, *`method`*, *`freq`*, and *`options`*. When you specify *`freq`*, *`method`* must be `'prewarp'`. The function also returns two cell arrays that contain full path names of the original, continuous blocks and the new, discretized blocks.

Input Arguments

`model_name`

Name of the model to discretize.

`sample_time`

Sample-time specification for the model:

Scalar value	Sample time with zero offset, such as `1`
Two-element vector	Sample time with nonzero offset, such as `[1 0.1]`

`method`

Method of converting blocks from continuous to discrete mode:

`'zoh'` (default)	Zero-order hold on the inputs
`'foh'`	First-order hold on the inputs
`'tustin'`	Bilinear (Tustin) approximation
`'prewarp'`	Tustin approximation with frequency prewarping
`'matched'`	Matched pole-zero method For single-input, single-output (SISO) systems only

`freq`

Critical frequency in Hz. This input applies only when the *`method`* input is `'prewarp'`.

`options`

Cell array {*`target`*, *`replace_with`*, *`put_into`*, *`prompt`*}, where each element can take the following values:

`target`	`'all'` (default)	Discretize all continuous blocks
	`'selected'`	Discretize only selected blocks in the model
	`'`*`full_blk_path`*`'`	Discretize specified block
`replace_with`	`'parammask'` (default)	Create discrete blocks whose parameters derive from the corresponding continuous blocks
	`'hardcoded'`	Create discrete blocks with hard-coded parameters placed directly into each block dialog box
`put_into`	`'copy'` (default)	Create discretization in a copy of the original model
	`'configurable'`	Create discretization candidate in a configurable subsystem
	`'current'`	Apply discretization to the current model
	`'untitled'`	Create discretization in a new untitled window
`prompt`	`'on'` (default)	Show discretization information at the command prompt
	`'off'`	Do not show discretization information at the command prompt

Examples

Discretize all continuous blocks in the `slexAircraftExample` model using a 1-second sample time:

```
sldiscmdl('slexAircraftExample',1);
```

Discretize the `Aircraft Dynamics Model` subsystem in the `slexAircraftExample` model using a 1-second sample time, a 0.1-second offset, and a first-order hold transform method:

```
sldiscmdl('slexAircraftExample',[1 0.1],'foh',...
{'slexAircraftExample/Aircraft Dynamics Model',...
'parammask','copy','on'});
```

Discretize the `Aircraft Dynamics Model` subsystem in the `slexAircraftExample` model and retrieve the full path name of the second discretized block:

```
[old_blks,new_blks] = sldiscmdl('slexAircraftExample',[1 0.1],...
'foh',{'slexAircraftExample/Aircraft Dynamics Model','parammask',...
'copy','on'});
% Get full path name of the second discretized block
new_blks{2}
```

More About

- "Discretize a Model with the sldiscmdl Function"

See Also

`slmdldiscui`

Introduced before R2006a

slIsFileChangedOnDisk

Determine whether model has changed since it was loaded

Syntax

```
Changed = slIsFileChangedOnDisk('sys')
```

Description

`Changed = slIsFileChangedOnDisk('sys')` Returns true if the file which contains block diagram *`sys`* was changed on disk since the block diagram was loaded.

Examples

To ensure that code is not generated for a model whose file has changed on disk since it was loaded, include the following in the `'entry'` section of the *`STF`*`_make_rtw_hook.m` file:

```
if (slIsFileChangedOnDisk(sys))
 error('File has changed on disk since it was loaded. Aborting code generation.');
end
```

More About

- "Customize Build Process with STF_make_rtw_hook File"
- "Model File Change Notification"

Introduced in R2007b

slLibraryBrowser

Open Simulink Library Browser

Syntax

```
slLibraryBrowser
slLibraryBrowser('open')
slLibraryBrowser('noshow')
libraryhandle = slLibraryBrowser
slLibraryBrowser('close')
```

Description

`slLibraryBrowser` opens the Simulink Library Browser.

If you want to load the Simulink block library, use `load_system simulink` instead.

If you want to start Simulink without opening any windows, use the faster `start_simulink` instead.

`slLibraryBrowser('open')` opens the Library Browser.

`slLibraryBrowser('noshow')` loads the Library Browser in memory without making it visible. Use this to make future calls to `slLibraryBrowser('open')` faster.

`libraryhandle = slLibraryBrowser` returns the handle of the Library Browser object.

`slLibraryBrowser('close')` closes the Library Browser.

Examples

Open and Close the Library Browser

```
slLibraryBrowser
```

```
slLibraryBrowser('close')
```

Load the Library Browser and Get a Handle

```
libraryhandle = slLibraryBrowser('noshow')
```

See Also

`simulink` | `start_simulink`

Introduced in R2016a

slmdldiscui

Open Model Discretizer GUI

Syntax

```
slmdldiscui
slmdldiscui('model')
```

Description

`slmdldiscui` opens the Model Discretizer. A model does not need to be open.

`slmdldiscui('model')` opens the Model Discretizer for the model or library called `'name'`.

To use the Model Discretizer, you must have a Control System Toolbox license, version 5.2 or later.

Examples

Open the Model Discretizer for the `slexAircraftExample` model:

```
slmdldiscui('slexAircraftExample')
```

Open the Model Discretizer for the `discretizing` library:

```
slmdldiscui('discretizing')
```

More About

- "Discretize a Model with the Model Discretizer"

See Also

`sldiscmdl`

Introduced before R2006a

slprofreport

Regenerate profiler report from data, `ProfileData`, saved from previous run

Syntax

`slprofreport(`*`model_name`*`ProfileData)`

Description

When you run a model with the profiler enabled, the simulation generates the data and saves it in the variable, *`model_name`*`ProfileData`. `slprofreport(`*`model_name`*`ProfileData)` generates a profiler report based on the data in *`model_name`*`ProfileData`, saved from the model run.

Input Arguments

`ProfileData`

Variable that contains profiler data from a model run. The variable name consists of the model name and `ProfileData`, for example, `vdpProfileData`.

Default: None

Examples

Regenerate Simulink Profiler Results

Regenerate the Profiler report for model `vdp`

In the MATLAB Command Window, start the `vdp` model.

In the Simulink editor window, run `vdp` model with Simulink Profiler enabled.

Simulink stores the data to the variable `vdpProfileData`.

To review the report, in the MATLAB Command Window

```
slprofreport(vdpProfileData)
```

The Simulink Profiler Report window is displayed.

- "Save Profiler Results"

More About

- "How Profiler Captures Performance Data"

Introduced in R2012a

slproject.getCurrentProject

Manipulate current Simulink Project at command line

Syntax

```
proj = slproject.getCurrentProject
```

Description

`proj = slproject.getCurrentProject` gets the current project open in the Simulink Project Tool and returns a project object `proj` that you can use to manipulate the project programmatically. If no project is open, then you see an error.

Note: `slproject.getCurrentProject` will be removed in a future release. Use `slproject.getCurrentProjects` instead.

Examples

Get Airframe Example Project

Open the Airframe project and use `slproject.getCurrentProject` to get a project object to manipulate the project at the command line.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProject
```

```
proj =

  ProjectManager with properties:

         Name: 'Simulink Project Airframe Example'
       Categories: [1x1 slproject.Category]
        Shortcuts: [1x8 slproject.Shortcut]
      ProjectPath: [1x7 slproject.PathFolder]
   ProjectReferences: [1x0 slproject.ProjectReference]
            Files: [1x30 slproject.ProjectFile]
```

```
RootFolder: 'C:\Work\Simulink\Projects\slexamples\airframe'
```

Output Arguments

`proj` — Project

project object

Project, returned as a project object. Use the project object to manipulate the currently open Simulink Project at the command line.

See Also

Functions

`simulinkproject` | `slproject.getCurrentProjects` | `slproject.loadProject`

Introduced in R2013a

slproject.getCurrentProjects

List all top-level Simulink projects

Syntax

```
projects = slproject.getCurrentProjects
```

Description

`projects = slproject.getCurrentProjects` returns a list of all top-level projects open in Simulink Project. Currently only one or zero top-level projects can be loaded. Returns an object array of 1 or 0 `ProjectManager` objects `projects` that you can use to manipulate the project programmatically. Use `slproject.getCurrentProjects` for project automation scripts.

If you execute `slproject.getCurrentProjects` inside a project shortcut, it returns only the project that the shortcut belongs to. If the shortcut belongs to a referenced project, it returns the referenced project.

Examples

Get Airframe Example Project

Open the Airframe project and use `slproject.getCurrentProjects` to get a project object to manipulate the project at the command line.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects
```

```
proj =

  ProjectManager with properties:

          Name: 'Simulink Project Airframe Example'
           Categories: [1x1 slproject.Category]
```

```
          Shortcuts: [1x8 slproject.Shortcut]
        ProjectPath: [1x7 slproject.PathFolder]
  ProjectReferences: [1x0 slproject.ProjectReference]
              Files: [1x30 slproject.ProjectFile]
         RootFolder: 'C:\Work\Simulink\Projects\airframe'
```

Find Project Commands

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProject
```

```
proj =

  ProjectManager with properties:

         Name: 'Simulink Project Airframe Example'
       Categories: [1x1 slproject.Category]
        Shortcuts: [1x8 slproject.Shortcut]
      ProjectPath: [1x7 slproject.PathFolder]
  ProjectReferences: [1x0 slproject.ProjectReference]
              Files: [1x30 slproject.ProjectFile]
         RootFolder: 'C:\Work\Simulink\Projects\airframe'
```

Find out what you can do with your project.

```
methods(proj)
```

```
Methods for class slproject.ProjectManager:

addFile                       findCategory
addFolderIncludingChildFiles  findFile
close                         isLoaded
createCategory                listModifiedFiles
export                        refreshSourceControl


reload
removeCategory
removeFile
```

Examine Project Properties

After you get a project object, you can examine project properties.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Examine the project files.

```
files = proj.Files
```

```
files =

  1x30 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Examine the labels of the eighth file.

```
proj.Files(8).Labels
```

```
ans =

  Label with properties:

File: 'C:\Work\airframe\data\system_model.sldd'
            Data: []
        DataType: 'none'
            Name: 'Design'
    CategoryName: 'Classification'
```

Get a particular file.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
```

```
myfile =

  ProjectFile with properties:

   Path:  'C:\Temp\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
  Revision: '2'
SourceControlStatus: Unmodified
```

Find out what you can do with the file.

```
methods(myfile)

Methods for class slproject.ProjectFile:

addLabel
removeLabel
findLabel
```

Output Arguments

`projects` — Projects

object array of 1 or 0 `ProjectManager` objects

Projects, returned as an object array of 1 or 0 `ProjectManager` objects. Use the project object to manipulate the currently open Simulink Project at the command line.

Properties of `ProjectManager` objects in output argument.

Project Property	Description
Name	Project name
Categories	Categories of project labels
Shortcuts	Shortcut files in project
ProjectPath	Folders that the project puts on the MATLAB path
ProjectReferences	Folders that contain referenced projects
Files	Paths and names of project files
RootFolder	Full path to project root folder

More About

Tips

Alternatively, you can use `simulinkproject` to get a project object, but `simulinkproject` also opens and gives focus to the Simulink Project Tool. Use `simulinkproject` to open projects and explore projects interactively. Use `slproject.getCurrentProjects` for project automation scripts.

See Also

Functions

`simulinkproject` | `slproject.getCurrentProject` | `slproject.loadProject`

Introduced in R2016a

slproject.loadProject

Load Simulink project

Syntax

```
slproject.loadProject(projectPath);
proj = slproject.loadProject(projectPath)
```

Description

`slproject.loadProject(projectPath);` loads the project specified by the `.prj` file or folder `projectPath` in the Simulink Project Tool, and closes any currently open project.

`proj = slproject.loadProject(projectPath)` loads the project and returns a project object `proj` for manipulating the project. Use `slproject.loadProject` for project automation scripts.

Examples

Load Project

Load a project from a folder called `'C:/projects/project1/'`. Replace this path with the location of your project.

```
proj = slproject.loadProject('C:/projects/project1/')
```

Get Airframe Example Project

Open the Airframe project and use `slproject.getCurrentProjects` to get a project object to manipulate the project at the command line.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects
```

```
proj =
```

```
  ProjectManager with properties:

          Name: 'Simulink Project Airframe Example'
        Categories: [1x1 slproject.Category]
         Shortcuts: [1x8 slproject.Shortcut]
       ProjectPath: [1x7 slproject.PathFolder]
  ProjectReferences: [1x0 slproject.ProjectReference]
             Files: [1x30 slproject.ProjectFile]
        RootFolder: 'C:\Work\Simulink\Projects\airframe'
```

Find Project Commands

Get the Airframe project.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Find project commands.

```
methods(proj)
```

```
Methods for class slproject.ProjectManager:

addFile                         findCategory
addFolderIncludingChildFiles    findFile
close                           isLoaded
createCategory                  listModifiedFiles
export                          refreshSourceControl


reload
removeCategory
removeFile
```

Examine Project Properties

After you get a project object, you can examine project properties.

Get the airframe project.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Examine the project files.

```
files = proj.Files
```

```
files =

  1x30 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Examine the labels of the 13th file.

```
proj.Files(13).Labels
```

```
ans =

  Label with properties:

File:  'C:\Temp\airframe\models\AnalogControl.mdl'
           Data: []
       DataType: 'none'
           Name: 'Design'
   CategoryName: 'Classification'
```

Get a particular file by name.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
```

```
myfile =

  ProjectFile with properties:

      Path:  'C:\Temp\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
  Revision: '2'
SourceControlStatus: Unmodified
```

Find out what you can do with the file.

```
methods(myfile)
```

```
Methods for class slproject.ProjectFile:

addLabel
removeLabel
```

```
findLabel
```

Input Arguments

`projectPath` — Full path to project file or folder

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
Name	Project name
Categories	Categories of project labels
Shortcuts	Shortcut files in project
ProjectPath	Folders that the project puts on the MATLAB path
ProjectReferences	Folders that contain referenced projects
Files	Paths and names of project files
RootFolder	Full path to project root folder

More About

- “What Are Simulink Projects?”

See Also

Functions

`simulinkproject` | `slproject.getCurrentProjects`

Introduced in R2013a

listModifiedFiles

List modified files in Simulink project

Syntax

```
modifiedfiles = listModifiedFiles(proj)
```

Description

`modifiedfiles = listModifiedFiles(proj)` returns the list of modified project files in the project object `proj`. `listModifiedFiles` refreshes the source control statuses in the project and then returns an array of the project files which are listed in the Modified Files view of the Simulink Project.

Examples

Get a List of Modified Files in the Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Modify a project model file by adding an arbitrary block.

```
open_system('AnalogControl')
add_block('built-in/SubSystem', 'AnalogControl/test')
save_system('AnalogControl')
```

Get all the modified files in the project.

```
modifiedfiles = listModifiedFiles(proj)
```

```
modifiedfiles =

  1x2 ProjectFile array with properties:

    Path
```

```
    Labels
    Revision
    SourceControlStatus
```

Observe two modified files. Compare with the Modified Files view in Simulink Project, where you can see a modified model file, and the corresponding `.SimulinkProject` definition file.

Get the second modified file.

```
modifiedfiles(2)
```

```
ans =

  ProjectFile with properties:

Path: 'C:\Work\temp\slexamples\airframe2\models\AnalogControl.mdl'
                Labels: [1x1 slproject.Label]
              Revision: '2'
    SourceControlStatus: Modified
```

Observe the file `SourceControlStatus` property is `Modified`. Similarly, `listModifiedFiles` returns any files that are added, conflicted, deleted, etc., that show up in the Modified Files view in Simulink Project.

Get all the project files with a particular source control status. For example, get the files that are `Unmodified`.

```
proj.Files(ismember([proj.Files.SourceControlStatus], matlab.sourcecontrol.Status.Unmoc
```

```
ans =

  1x29 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Input Arguments

`proj` — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

Output Arguments

`modifiedfiles` — Modified files

file object | array

Modified files, returned as an array of file objects.

See Also

Functions

`refreshSourceControl` | `simulinkproject`

Introduced in R2016a

refreshSourceControl

Update source control status of Simulink project files

Syntax

```
refreshSourceControl(proj)
```

Description

`refreshSourceControl(proj)` updates the source control status for all files in the Simulink project `proj`. Use this to get the latest source control information before querying the `SourceControlStatus` property on individual files.

If you use `listModifiedFiles` to find all modified files in the project, you do not need to call `refreshSourceControl` first.

Examples

Refresh Source Control Information on Files in the Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Refresh source control status before querying individual files.

```
refreshSourceControl(proj)
```

Input Arguments

`proj` — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

See Also

Functions
`listModifiedFiles` | `simulinkproject`

Introduced in R2016a

slreplace_mux

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

More About

- “Connectivity Diagnostics Overview”
- “Prevent Bus and Mux Mixtures”
- “Mux Signals”
- “Composite Signals”

See Also

`Bus Creator` | `Mux`

Introduced before R2006a

start_simulink

Start Simulink without opening any windows

Syntax

```
start_simulink
```

Description

`start_simulink` starts Simulink without opening any models, the Start Page, or the Simulink Library Browser. Use this in startup scripts to start Simulink without any other window taking the focus away from the MATLAB Desktop. For example, use `start_simulink` in the MATLAB `startup.m` file, when starting MATLAB with the `-r` command line option, or in Simulink project startup scripts. Opening a model for the first time in a MATLAB session is much quicker after running `start_simulink`.

If you want to open the Simulink Start Page to create or open models, use the `simulink` function instead.

If you want to open the Library Browser, use `slLibraryBrowser`.

Examples

Start Simulink When Starting MATLAB

Use the `-r` command line option to start Simulink when starting MATLAB, without opening any windows.

On Windows, create a desktop shortcut with the following target:

```
matlabroot\bin\win64\matlab.exe -r start_simulink
```

On Linux® and Mac, enter:

```
matlab -r start_simulink
```

- "Automate Startup Tasks with Shortcuts"

See Also

`simulink` | `simulinkproject` | `slLibraryBrowser`

Introduced in R2015b

slupdate

Replace blocks from previous releases with latest versions

Note: `slupdate` will be removed in a future release. The `slupdate` command can only upgrade some parts of your model. Use the Upgrade Advisor instead. See "Model Upgrades".

Syntax

```
slupdate('sys')
slupdate('sys', prompt)
AnalysisResult = slupdate('sys', 'OperatingMode', 'Analyze')
```

Description

`slupdate('sys')` replaces blocks in model *`sys`* from a previous release of Simulink software with the latest versions. The `slupdate` function alone cannot perform all upgrade checks on your model. Use the Upgrade Advisor to access the `slupdate` checks and also advice and fixes for all other upgrade checks. See "Model Upgrades".

Note Best practice is to first open the model, and press CTRL+D to update the model, before you call `slupdate`.

`slupdate('sys', prompt)` specifies whether to prompt you before replacing a block. If *`prompt`* equals `1`, the command prompts you before replacing the block. The prompt asks whether you want to replace the block. Valid responses are

- `y`

  Replace the block (the default).

- `n`

Do not replace the block.

- `a`

  Replace this and all subsequent obsolete blocks without further prompting.

If *`prompt`* equals `0`, the command replaces all obsolete blocks without prompting you.

In addition to replacing obsolete blocks, `slupdate`

- Reconnects broken links to masked blocks in libraries provided by MathWorks to ensure that the model reflects changes made to the blocks in this release. This will overwrite any custom changes you made to the masks of these blocks.
- Updates obsolete configuration settings for the model.

`AnalysisResult = slupdate('sys', 'OperatingMode', 'Analyze')` performs only the analysis portion without updating or changing the model. This command analyzes referenced models, linked libraries, and S-functions, and then returns a data structure with the following fields:

- `Message` — 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`)

More About

- "Model Upgrades"

See Also

`upgradeadvisor`

Introduced before R2006a

trim

Find trim point of dynamic system

Syntax

```
[x,u,y,dx] = trim('sys')
[x,u,y,dx] = trim('sys',x0,u0,y0)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx)
[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options)
[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)
```

Description

A trim point, also known as an equilibrium point, is a point in the parameter space of a dynamic system at which the system is in a steady state. For example, a trim point of an aircraft is a setting of its controls that causes the aircraft to fly straight and level. Mathematically, a trim point is a point where the system's state derivatives equal zero. `trim` starts from an initial point and searches, using a sequential quadratic programming algorithm, until it finds the nearest trim point. You must supply the initial point implicitly or explicitly. If `trim` cannot find a trim point, it returns the point encountered in its search where the state derivatives are closest to zero in a min-max sense; that is, it returns the point that minimizes the maximum deviation from zero of the derivatives. `trim` can find trim points that meet specific input, output, or state conditions, and it can find points where a system is changing in a specified manner, that is, points where the system's state derivatives equal specific nonzero values.

`[x,u,y,dx] = trim('sys')` finds the equilibrium point of the model '`sys`', nearest to the system's initial state, `x0`. Specifically, `trim` finds the equilibrium point that minimizes the maximum absolute value of `[x-x0,u,y]`. If `trim` cannot find an equilibrium point near the system's initial state, it returns the point at which the system is nearest to equilibrium. Specifically, it returns the point that minimizes `abs(dx)` where `dx` represents the derivative of the system. You can obtain `x0` using this command.

```
[sizes,x0,xstr] = sys([],[],[],0)
```

`[x,u,y,dx] = trim('sys',x0,u0,y0)` finds the trim point nearest to `x0`, `u0`, `y0`, that is, the point that minimizes the maximum value of

```
abs([x-x0; u-u0; y-y0])
```

`[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy)` finds the trim point closest to `x0`, `u0`, `y0` that satisfies a specified set of state, input, and/or output conditions. The integer vectors `ix`, `iu`, and `iy` select the values in `x0`, `u0`, and `y0` that must be satisfied. If `trim` cannot find an equilibrium point that satisfies the specified set of conditions exactly, it returns the nearest point that satisfies the conditions, namely,

```
abs([x(ix)-x0(ix); u(iu)-u0(iu); y(iy)-y0(iy)])
```

`[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx)` finds specific nonequilibrium points, that is, points at which the system's state derivatives have some specified nonzero value. Here, `dx0` specifies the state derivative values at the search's starting point and `idx` selects the values in `dx0` that the search must satisfy exactly.

`[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options)` specifies an array of optimization parameters that `trim` passes to the optimization function that it uses to find trim points. The optimization function, in turn, uses this array to control the optimization process and to return information about the process. `trim` returns the `options` array at the end of the search process. By exposing the underlying optimization process in this way, `trim` allows you to monitor and fine-tune the search for trim points.

The following table describes how each element affects the search for a trim point. Array elements 1, 2, 3, 4, and 10 are particularly useful for finding trim points.

No.	Default	Description
1	0	Specifies display options. 0 specifies no display; 1 specifies tabular output; -1 suppresses warning messages.
2	10^{-4}	Precision the computed trim point must attain to terminate the search.
3	10^{-4}	Precision the trim search goal function must attain to terminate the search.
4	10^{-6}	Precision the state derivatives must attain to terminate the search.
5	N/A	Not used.

No.	Default	Description
6	N/A	Not used.
7	N/A	Used internally.
8	N/A	Returns the value of the trim search goal function (λ in goal attainment).
9	N/A	Not used.
10	N/A	Returns the number of iterations used to find a trim point.
11	N/A	Returns the number of function gradient evaluations.
12	0	Not used.
13	0	Number of equality constraints.
14	100*(Number of variables)	Maximum number of function evaluations to use to find a trim point.
15	N/A	Not used.
16	10^{-8}	Used internally.
17	0.1	Used internally.
18	N/A	Returns the step length.

`[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)` sets the time to `t` if the system is dependent on time.

Note: If you fix any of the state, input or output values, `trim` uses the unspecified free variables to derive the solution that satisfies these constraints.

Examples

Consider a linear state-space system modeled using a State-Space block

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

The A, B, C, and D matrices to enter at the command line or in the block parameters dialog are:.

```
A = [-0.09 -0.01;  1   0];
B = [ 0    -7;     0  -2];
C = [ 0     2;     1  -5];
D = [-3     0;     1   0];
```

Example 1

To find an equilibrium point in this model called `sys`, use:

```
[x,u,y,dx,options] = trim('sys')
x =
   0
   0
u =
   0
   0
y =
   0
   0
dx =
   0
   0
```

The number of iterations taken is:

```
options(10)
ans =
      7
```

Example 2

To find an equilibrium point near `x = [1;1], u = [1;1]`, enter

```
x0 = [1;1];
u0 = [1;1];
[x,u,y,dx,options] = trim('sys', x0, u0);
x =
    1.0e-13 *
   -0.5160
   -0.5169
u =
    0.3333
    0.0000
```

```
y =
   -1.0000
    0.3333
dx =
    1.0e-12 *
    0.1979
    0.0035
```

The number of iterations taken is

```
options(10)
ans =
      25
```

Example 3

To find an equilibrium point with the outputs fixed to 1, use:

```
y = [1;1];
iy = [1;2];
[x,u,y,dx] = trim('sys', [], [], y, [], [], iy)
x =
    0.0009
   -0.3075
u =
   -0.5383
    0.0004
y =
    1.0000
    1.0000
dx =
    1.0e-15 *
   -0.0170
    0.1483
```

Example 4

To find an equilibrium point with the outputs fixed to 1 and the derivatives set to 0 and 1, use

```
y = [1;1];
iy = [1;2];
dx = [0;1];
```

```
idx = [1;2];
[x,u,y,dx,options] = trim('sys',[],[],y,[],[],iy,dx,idx)
x =
    0.9752
   -0.0827
u =
   -0.3884
   -0.0124
y =
    1.0000
    1.0000
dx =
    0.0000
    1.0000
```

The number of iterations taken is

```
options(10)
ans =
      13
```

Limitations

The trim point found by `trim` starting from any given initial point is only a local value. Other, more suitable trim points may exist. Thus, if you want to find the most suitable trim point for a particular application, it is important to try a number of initial guesses for `x`, `u`, and `y`.

More About

Algorithms

`trim` uses a sequential quadratic programming algorithm to find trim points. See "Sequential Quadratic Programming (SQP)" for a description of this algorithm.

Introduced before R2006a

tunablevars2parameterobjects

Create Simulink parameter objects from tunable parameters

Syntax

```
tunablevars2parameterobjects ('modelName')
tunablevars2parameterobjects ('modelName', class)
```

Description

`tunablevars2parameterobjects ('modelName')` creates `Simulink.Parameter` objects in the base workspace for the variables listed in the specified model's Tunable Parameters dialog, then deletes the source information from the dialog. To preserve the information, save the resulting Simulink parameter objects into a MAT-file.

If a tunable variable is already defined as a numeric variable in the base workspace, the variable will be replaced by a parameter object and the original variable will be copied to the object's Value property.

If a tunable variable is already defined as a Simulink parameter object, the object will not be modified but the information for the variable will still be deleted from the Tunable Parameters dialog.

If a tunable variable is defined as any other class of variable, the variable will not be modified and the information for the variable will not be deleted from the Tunable Parameters dialog.

`tunablevars2parameterobjects ('modelName', class)` creates objects of the specified class rather than `Simulink.Parameter` objects.

Input Arguments

`modelName`

Model name or handle

`class`

Parameter class to use for creating objects

Default: `Simulink.Parameter`

More About

- "Tunable Parameters"

See Also

`Simulink.Parameter`

Introduced in R2007b

ufix

Create `Simulink.NumericType` object describing unsigned fixed-point data type

Syntax

```
a = ufix(WordLength)
```

Description

`ufix(WordLength)` returns a `Simulink.NumericType` object that describes an unsigned fixed-point data type with the specified word length and unspecified scaling.

Note: `ufix` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `ufix(WordLength)` with `fixdt(0,WordLength)`.

Examples

Define a 16-bit unsigned fixed-point data type.

```
a = ufix(16)
```

```
a =

  NumericType with properties:

    DataTypeMode: 'Fixed-point: unspecified scaling'
      Signedness: 'Unsigned'
      WordLength: 16
         IsAlias: 0
       DataScope: 'Auto'
      HeaderFile: ''
     Description: ''
```

See Also

`fixdt` | `Simulink.NumericType` | `float` | `sfix` | `sfrac` | `sint` | `ufrac` | `uint`

Introduced before R2006a

ufrac

Create `Simulink.NumericType` object describing unsigned fractional data type

Syntax

```
a = ufrac(WordLength)
a = ufrac(WordLength, GuardBits)
```

Description

`ufrac(WordLength)` returns a `Simulink.NumericType` object that describes the data type of an unsigned fractional data type with a word size given by `WordLength`.

`ufrac(WordLength, GuardBits)` returns a `Simulink.NumericType` object that describes the data type of an unsigned fractional data type. The total word size is given by *`WordLength`* with *`GuardBits`* bits located to the left of the binary point.

Note: `ufrac` is a legacy function. In new coder, use `fixdt` instead. In existing code, replace `ufrac(WordLength)` with `fixdt(0,WordLength,WordLength)` and `ufrac(WordLength,GuardBits)` with `fixdt(0,WordLength,(WordLength-GuardBits))`.

Examples

Define an 8-bit unsigned fractional data type with 4 guard bits. Note that the range of this data type is from 0 to $(1 - 2^{-8}).2^4 = 15.9375$.

```
a = ufrac(8,4)
```

```
a =

  NumericType with properties:

      DataTypeMode: 'Fixed-point: binary point scaling'
        Signedness: 'Unsigned'
```

```
    WordLength: 8
FractionLength: 4
       IsAlias: 0
     DataScope: 'Auto'
    HeaderFile: ''
   Description: ''
```

See Also

`fixdt` | `Simulink.NumericType` | `float` | `sfix` | `sfrac` | `sint` | `ufix` | `uint`

Introduced before R2006a

uint

Create `Simulink.NumericType` object describing unsigned integer data type

Syntax

```
a = uint(WordLength)
```

Description

`uint(WordLength)` returns a `Simulink.NumericType` object that describes the data type of an unsigned integer with a word size given by `WordLength`.

Note: `uint` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `uint(WordLength)` with `fixdt(0,WordLength,0)`.

Examples

Define a 16-bit unsigned integer.

```
a = uint(16)
```

```
a =

  NumericType with properties:

      DataTypeMode: 'Fixed-point: binary point scaling'
        Signedness: 'Unsigned'
        WordLength: 16
    FractionLength: 0
           IsAlias: 0
         DataScope: 'Auto'
        HeaderFile: ''
       Description: ''
```

See Also

`fixdt` | `Simulink.NumericType` | `float` | `sfix` | `sfrac` | `sint` | `ufix` | `ufrac`

Introduced before R2006a

unpack

Extract signal logging objects from signal logs and write them into MATLAB workspace

Syntax

```
log.unpack
tsarray.unpack
log.unpack('systems')
log.unpack('all')
```

Description

Note: The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

`log.unpack` or `unpack(log)` extracts the top level elements of the `Simulink.ModelDataLogs` or `Simulink.SubsysDataLogs` object named **`log`** (e.g., `logsout`).

`log.unpack('systems')` or `unpack(`*`log`*`, 'systems')` extracts `Simulink.Timeseries` and `Simulink.TsArray` objects from the `Simulink.ModelDataLogs` or `Simulink.SubsysDataLogs` object named `log` . This command does not extract `Simulink.Timeseries` objects from `Simulink.TsArray` objects nor does it write intermediate `Simulink.ModelDataLogs` or `Simulink.SubsysDataLogs` objects to the MATLAB workspace.

`log.unpack('all')` or `unpack(log, 'all')` extracts all the `Simulink.Timeseries` objects contained by the `Simulink.ModelDataLogs`, `Simulink.TsArray`, or `Simulink.SubsysDataLogs` object named `log`.

`tsarray.unpack` extracts the time-series objects of class `Simulink.Timeseries` from the `Simulink.TsArray` object named `tsarray`.

More About

- "Export Signal Data Using Signal Logging"

See Also

`Simulink.ModelDataLogs` | `Simulink.SubsysDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` | `who` | `whos`

Introduced before R2006a

upgradeadvisor

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.

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`.

More About

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.

See Also

`modeladvisor`

Introduced in R2012b

view_mdlrefs

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.

More About

- "Model Reference"
- "Model Dependency Viewer"

See Also

`Model` | `find_mdlrefs`

Tutorials

- sldemo_mdlref_depgraph

Introduced before R2006a

who

List names of top-level data logging objects in Simulink `ModelDataLogs` data log

Syntax

```
log.who

tsarray.who

log.who('systems')

log.who('all')
```

Description

Note: To list names of top-level data logging objects in `Dataset` format, use `Simulink.SimulationData.Dataset.find`.

The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

`log.who` or `who(log)` lists the names of the top-level signal logging objects contained by `log`, where `log` is the handle of a `Simulink.ModelDataLogs` object name.

`tsarray.who` or `who(tsarray)` lists the names of `Simulink.TimeSeries` objects contained by the `Simulink.TsArray` object named `tsarray`.

`log.who('systems')` or `who(log, 'systems')` lists the names of all signal logging objects contained by `log` except for `Simulink.Timeseries` objects stored in `Simulink.TsArray` objects contained by `log`.

`log.who('all')` or `who(log, 'all')` lists the names of all the `Simulink.Timeseries` objects contained by the `Simulink.ModelDataLogs`, `Simulink.TsArray`, or `Simulink.SubsysDataLogs` object named `log`.

For information about other uses of `who`, execute help who in the MATLAB Command Window.

Tip To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

More About

- "Access Logged Data from Persistent Storage"

See Also

`Simulink.SimulationData.Dataset.find` | `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` | `Simulink.ModelDataLogs` | `Simulink.SubsysDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` | `whos` | `unpack`

Introduced before R2006a

whos

List names and types of top-level data logging objects in Simulink `ModelDataLogs` data log

Syntax

```
log.whos

tsarray.whos

log.whos('systems')

log.whos('all')
```

Description

Note: To list names of top-level data logging objects in `Dataset` format, use `Simulink.SimulationData.Dataset.find`.

The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

`log.whos` or `whos(log)` lists the names and types of the top-level signal logging objects contained by `log`, where `log` is the handle of a `Simulink.ModelDataLogs` object name.

`tsarray.whos` or `whos(tsarray)` lists the names and types of `Simulink.TimeSeries` objects contained by the `Simulink.TsArray` object named `tsarray`.

`log.whos('systems')` or `whos(log, 'systems')` lists the names and types of all signal logging objects contained by `log` except for `Simulink.Timeseries` objects stored in `Simulink.TsArray` objects contained by `log`.

`log.whos('all')` or `whos(log, 'all')` lists the names and types of all the `Simulink.Timeseries` objects contained by the `Simulink.ModelDataLogs`, `Simulink.TsArray` or `Simulink.SubsysDataLogs` object named `log`.

For information about other uses of `whos`, execute help whos in the MATLAB Command Window.

Tip To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

More About

- "Access Logged Data from Persistent Storage"

See Also

`Simulink.SimulationData.Dataset.find` | `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` | `Simulink.ModelDataLogs` | `Simulink.SubsysDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` | `who` | `unpack`

Introduced before R2006a

Mask Icon Drawing Commands

color	Change drawing color of subsequent mask icon drawing commands
disp	Display text on masked subsystem icon
dpoly	Display transfer function on masked subsystem icon
droots	Display transfer function on masked subsystem icon
fprintf	Display variable text centered on masked subsystem icon
image	Display RGB image on masked subsystem icon
patch	Draw color patch of specified shape on masked subsystem icon
plot	Draw graph connecting series of points on masked subsystem icon
port_label	Draw port label on masked subsystem icon
text	Display text at specific location on masked subsystem icon

color

Change drawing color of subsequent mask icon drawing commands

Syntax

```
color(colorstr)
```

Description

`color(colorstr)` sets the drawing color of all subsequent mask drawing commands to the color specified by the string *`colorstr`*.

`colorstr` must be one of the following supported color strings.

```
blue
green
red
cyan
magenta
yellow
black
```

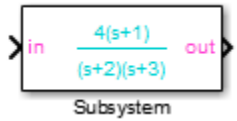
Entering any other string or specifying the color using RGB values results in a warning at the MATLAB command prompt; Simulink ignores the color change. The specified drawing color does not influence the color used by the `patch` or `image` drawing commands.

Examples

The following commands

```
color('cyan');
droots([-1], [-2 -3], 4)
color('magenta')
port_label('input',1,'in')
port_label('output',1,'out')
```

draw the following mask icon.



See Also

`droots` | `port_label`

Introduced in R2006b

disp

Display text on masked subsystem icon

Syntax

```
disp(text)
disp(text, 'texmode', 'on')
```

Description

`disp(text)` displays *`text`* centered on the block icon. *`text`* is any MATLAB expression that evaluates to a string.

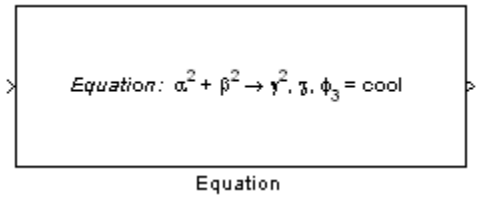
`disp``(text, 'texmode', 'on')` allows you to use TeX formatting commands in *`text`*. The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See Interpreter for information on the TeX formatting commands supported by Simulink software.

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` | `text` | `port_label`

Introduced in R2007a

dpoly

Display transfer function on masked subsystem icon

Syntax

```
dpoly(num, den)
dpoly(num, den, 'character')
```

Description

`dpoly(num, den)` displays the transfer function whose numerator is *num* and denominator is *den*.

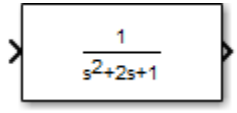
`dpoly(num, den, 'character')` specifies the name of the transfer function independent variable. The default is *s*.

When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- To display a continuous transfer function in descending powers of *s*, enter

  ```
  dpoly(num, den)
  ```

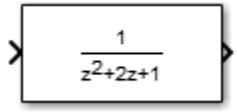
  For example, for `num = [0 0 1];` and `den = [1 2 1]` the icon looks like:

- To display a discrete transfer function in descending powers of *z*, enter

  ```
  dpoly(num, den, 'z')
  ```

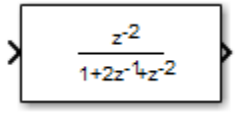
  For example, for `num = [0 0 1];` and `den = [1 2 1];` the icon looks like:

- To display a discrete transfer function in ascending powers of *1/z*, enter

  ```
  dpoly(num, den, 'z-')
  ```

  For example, for *num* and *den* as defined previously, the icon looks like:



If the parameters are not defined or have no values when you create the icon, Simulink software displays three question marks (`? ? ?`) in the icon. When you define parameter values in the Mask Settings dialog box, Simulink software evaluates the transfer function and displays the resulting equation in the icon.

See Also

`disp` | `port_label` | `text` | `droots`

droots

Display transfer function on masked subsystem icon

Syntax

```
droots(zero, pole, gain)
droots(zero, pole, gain,'z')
droots(zero, pole, gain,'z-')
```

Description

`droots(zero, pole, gain)` displays the transfer function whose zero is *`zero`*, pole is *`pole`*, and gain is *`gain`*.

`droots(zero, pole, gain,'z')` and `droots(zero, pole, gain,'z-')` expresses the transfer function in terms of *z* or *1/z*.

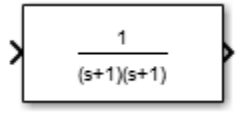
When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- To display a zero-pole gain transfer function, enter

  ```
  droots(z, p, k)
  ```

  For example, the preceding command creates this icon for these values:

  ```
  z = []; p = [-1 -1]; k = 1;
  ```

If the parameters are not defined or have no values when you create the icon, Simulink software displays three question marks (`? ? ?`) in the icon. When you define parameter values in the Mask Settings dialog box, Simulink software evaluates the transfer function and displays the resulting equation in the icon.

See Also

`disp` | `port_label` | `text` | `dpoly`

Introduced in R2007a

fprintf

Display variable text centered on masked subsystem icon

Syntax

```
fprintf(text)
fprintf(format, var)
```

Description

The `fprintf` command displays formatted text centered on the icon and can display *`format`* along with the contents of *`var`*.

Note While this `fprintf` function is identical in name to its corresponding MATLAB function, it provides only the functionality described on this page.

Examples

The command

```
fprintf('Hello');
```

displays the 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`

Introduced before R2006a

image

Display RGB image on masked subsystem icon

Syntax

```
image(a)
image(a, position)
image(a, position, rotation)
```

Description

`image(a)` displays the image `a`, where `a` is an *m*-by-*n*-by-3 array of RGB values. If necessary, use the MATLAB commands `imread` and `ind2rgb` to read and convert bitmap files (such as GIF) to the necessary matrix format.

`image(a, position)` creates the image at the specified position as follows.

Position	Description
[*x*, *y*, *w*, *h*]	Position (*x*, *y*) and size (*w*, *h*) of the image where the position is relative to the lower-left corner of the mask. The image scales to fit the specified size.
`'center'`	Center of the mask
`'top-left'`	Top left corner of the mask, unscaled
`'bottom-left'`	Bottom left corner of the mask, unscaled
`'top-right'`	Top right corner of the mask, unscaled
`'bottom-right'`	Bottom right corner of the mask, unscaled

`image(a, position, rotation)` allows you to specify whether the image rotates (`'on'`) or remains stationary (`'off'`) as the icon rotates. The default is `'off'`.

Examples

The command

```
image('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 **Icon & Ports** pane of the Mask Editor dialog box.

```
[data, map]=image('label.gif');
pic=ind2rgb(data,map);
```

Then type the command

```
image(pic)
```

in the **Icon & Ports** pane of the Mask Editor to read the converted label image.

See Also

`patch` | `plot`

Introduced before R2006a

patch

Draw color patch of specified shape on masked subsystem icon

Syntax

```
patch(x, y)
patch(x, y, [r g b])
```

Description

`patch(x, y)` creates a solid patch having the shape specified by the coordinate vectors `x` and `y`. The patch's color is the current foreground color.

`patch(x, y, [r g b])` creates a solid patch of the color specified by the vector `[r g b]`, where `r` is the red component, `g` the green, and `b` the blue. For example,

```
patch([0 .5 1], [0 1 0], [1 0 0])
```

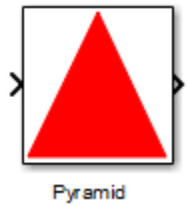
creates a red triangle on the mask's icon.

Examples

The command

```
patch([0 .5 1], [0 1 0], [1 0 0])
```

creates a red triangle on the mask's icon.



See Also

`image` | `plot`

Introduced before R2006a

plot

Draw graph connecting series of points on masked subsystem icon

Syntax

```
plot(Y)
plot(X1,Y1,X2,Y2,...)
```

Description

`plot(Y)` plots, for a vector *Y*, each element against its index. If *Y* is a matrix, it plots each column of the matrix as though it were a vector.

`plot(X1,Y1,X2,Y2,...)` plots the vectors *Y1* against *X1*, *Y2* against *X2*, and so on. Vector pairs must be the same length and the list must consist of an even number of vectors.

Plot commands can include `NaN` and `inf` values. When Simulink software encounters `NaN`s or `inf`s, it stops drawing, and then begins redrawing at the next numbers that are not `NaN` or `inf`. The appearance of the plot on the icon depends on the units defined by the **Icon units** option in the Mask Editor.

Simulink software displays three question marks (`? ? ?`) in the block icon and issues warnings in these situations:

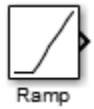
- When you have not defined values for the parameters used in the drawing commands (for example, when you first create the mask, but have not yet entered values in the Mask Settings dialog box)
- When you enter a masked block parameter or drawing command incorrectly

Examples

The command

```
plot([0 1 5], [0 0 4])
```

generates the plot that appears on the icon for the Ramp block, in the Sources library.



See Also

`image`

Introduced before R2006a

port_label

Draw port label on masked subsystem icon

Syntax

```
port_label('port_type', port_number, 'label')
port_label('port_type', port_number, 'label', 'texmode', 'on')
```

Description

`port_label('port_type', port_number, 'label')` draws a label on a port. Valid values for *`port_type`* include the following.

Value	Description
`input`	Simulink input port
`output`	Simulink output port
`lconn`	Physical Modeling connection port on the left side of a masked subsystem
`rconn`	Physical Modeling connection port on the right side of a masked subsystem
`Enable`	Label for the trigger port in a masked `Triggered` or`Enabled and Triggered` subsystem.
`trigger`	Label for the trigger port in a masked `Triggered` or`Enabled and Triggered` subsystem.
`action`	Label for the action port in a masked `Swith Case Action 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 Interpreter for information on the TeX formatting commands that the Simulink software supports.

Examples

The command

```
port_label('input', 1, 'a')
```

defines `a` as the label of input port 1.

The command

```
port_label('Enable','En')
```

defines `En` as the label of Enable port.

The command

```
port_label('trigger','Tr')
```

defines `Tr` as the label of trigger port.

The command

```
port_label('action','Switch():')
```

defines `Switch():` as the label of action port.

The command

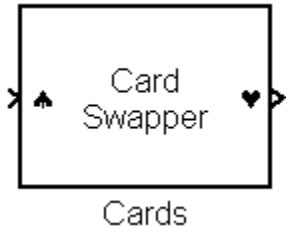
```
port_label('trigger','$\sqrt m$','interp','latex')
```

defines the label of trigger port with latex interpretation.

The commands

```
disp('Card\nSwapper');
port_label('input',1,'\spadesuit','texmode','on');
port_label('output',1,'\heartsuit','texmode','on');
```

draw playing card symbols as the labels of the ports on a masked subsystem.



See Also

`disp` | `fprintf` | `text`

Introduced before R2006a

text

Display text at specific location on masked subsystem icon

Syntax

```
text(x, y, 'text')
text(x, y, 'text', 'horizontalAlignment', 'halign',
  'verticalAlignment', 'valign')
text(x, y, 'text', 'texmode', 'on')
```

Description

The `text` command places a character string at a location specified by the point `(x,y)` whose units are defined by the **Icon units** option in the Mask Editor.

`text(x,y, text, 'texmode', 'on')` allows you to use TeX formatting commands in *`text`*. The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See Interpreter 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

Text Alignment

Center the mask icon text `foobar`.

```
text(0.5, 0.5, 'foobar', 'horizontalAlignment', 'center')
```

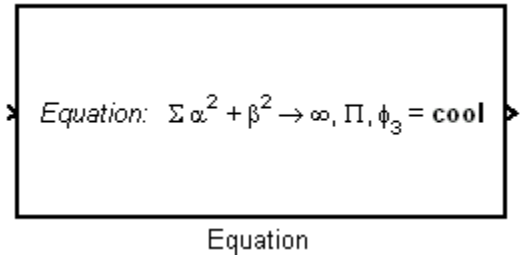
Equation in Mask Icon

Draw a left-aligned equation as the mask icon.

In the Icons & Ports dialog of the Mask Editor, set **Icon units** to `Normalized`.

In the **Icon drawing commands** text box, enter the following command.

```
text(.05,.5,'{\itEquation:}  \Sigma \alpha^2 +
\beta^2 \rightarrow \infty, \Pi, \phi_3 = {\bfcool}',
'hor','left','texmode','on')
```



See Also

`disp` | `fprintf` | `port_label`

Introduced before R2006a

Simulink Debugger Commands

ashow	Show algebraic loop
atrace	Set algebraic loop trace level
bafter	Insert breakpoint after specified method
break	Insert breakpoint before specified method
bshow	Show specified block
clear	Clear breakpoints from model
continue	Continue simulation
disp	Display block's I/O when simulation stops
ebreak	Enable (or disable) breakpoint on solver errors
elist	List simulation methods in order in which they are executed during simulation
emode	Toggle model execution between accelerated and normal mode
etrace	Enable or disable method tracing
help	Display help for debugger commands
nanbreak	Set or clear nonfinite value break mode
next	Advance simulation to start of next method at current level in model's execution list
probe	I/O and state data for blocks
quit	Stop simulation debugger
rbreak	Break simulation before solver reset
run	Run simulation to completion
slist	Sorted list of model blocks
states	Current state values

status	Debugging options in effect
step	Advance simulation by one or more methods
stimes	Model sample times
stop	Stop simulation
strace	Set solver trace level
systems	List nonvirtual systems of model
tbreak	Set or clear time breakpoint
trace	Display block's I/O each time block executes
undisp	Remove block from debugger's list of display points
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

ashow

Show algebraic loop

Syntax

```
ashow <gcb | s:b | s#n | clear>
as <gcb | s:b | s#n | clear>
```

Arguments

`gcb`	Current block.
`s:b`	The block whose system index is `s` and block index is `b`.
`s#n`	The algebraic loop numbered `n` in system `s`.
`clear`	Switch that clears loop coloring.

Description

`ashow` without any arguments lists all of a model's algebraic loops in the MATLAB Command Window. `ashow gcb` or `ashow s:b` highlights the algebraic loop that contains the specified block. `ashow s#n` highlights the `n`th algebraic loop in system `s`. The `ashow` **`clear`** command removes algebraic loop highlights from the model diagram.

See Also

`atrace` | `slist`

Introduced before R2006a

atrace

Set algebraic loop trace level

Syntax

```
atrace level
```

```
at level
```

Arguments

`level` Trace level (0 = none, 4 = everything).

Description

The `atrace` command sets the algebraic loop trace level for a simulation.

Command	Displays for Each Algebraic Loop
`atrace 0`	No information
`atrace 1`	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
`atrace 2`	Same as level 1
`atrace 3`	Level 2 plus Jacobian matrix used to solve loop
`atrace 4`	Level 3 plus intermediate solutions of the loop variable

See Also

`states` | `systems`

Introduced before R2006a

bafter

Insert breakpoint after specified method

Syntax

```
bafter
ba
bafter m:mid
bafter <sysIdx:blkIdx | gcb> [mth] [tid:TID]
bafter <s:sysIdx | gcs> [mth] [tid:TID]
bafter model [mth] [tid:TID]
```

Arguments

mid	Method ID
sysIdx:*blkIdx*	Block ID
gcb	Currently selected block
sysIdx	System ID
gcs	Currently selected system
model	Currently selected model
mth	A method name, e.g., `Outputs.Major`
TID	Task ID

Description

`bafter` inserts a breakpoint after the current method.

Instead of `bafter`, you can use the short form of `ba` with any of the syntaxes.

`bafter m:`*`mid`* inserts a breakpoint after the method specified by *mid* (see "Method ID").

`bafter` *`sysIdx`*`:`*`blkIdx`* inserts a breakpoint after each invocation of the method of the block specified by *`sysIdx`*`:`*`blkIdx`* (see "Block ID") in major time steps. `bafter gcb` inserts a breakpoint after each invocation of a method of the currently selected block (see `gcb`) in major times steps.

`bafter` **`s:`***`sysIdx`* inserts a breakpoint after each method of the root system or nonvirtual subsystem specified by the system ID: *`sysIdx`*.

Note The `systems` command displays the system IDs for all nonvirtual systems in the currently selected model.

`bafter` **`gcs`** inserts a breakpoint after each method of the currently selected nonvirtual system.

`bafter` **`model`** inserts a breakpoint after each method of the currently selected model.

The optional *`mth`* parameter allow you to set a breakpoint after a particular block, system, or model method and task. For example, `bafter` **`gcb`** `Outputs` sets a breakpoint after the Outputs method of the currently selected block.

The optional TID parameter allows you to set a breakpoint after invocation of a method by a particular task. For example, suppose that the currently selected nonvirtual subsystem operates on task 2 and 3. Then `bafter` **`gcs`** `Outputs` **`tid:`**`2` sets a breakpoint after the invocation of the subsystem's Outputs method that occurs when task 2 is active.

See Also

`break` | `ebreak` | `tbreak` | `xbreak` | `nanbreak` | `zcbreak` | `rbreak` | `clear` | `where` | `slist` | `systems`

Introduced before R2006a

break

Insert breakpoint before specified method

Syntax

```
break

b

break m:mid
break <sysIdx:blkIdx | gcb> [mth] [tid:TID]

break <s:sysIdx | gcs> [mth] [tid:TID]

break model [mth] [tid:TID]
```

Arguments

mid	Method ID
sysIdx:*blkIdx*	Block ID
gcb	Currently selected block
sysIdx	System ID
gcs	Currently selected system
model	Currently selected model
mth	A method name, e.g., `Outputs.Major`
TID	task ID

Description

`break` inserts a breakpoint before the current method.

Instead of `break`, you can use the short form of `b` with any of the syntaxes.

`break m:`*`mid`* inserts a breakpoint before the method specified by *`mid`* (see "Method ID").

`break` *`sysIdx`*`:`*`blkIdx`* inserts a breakpoint before each invocation of the method of the block specified by *`sysIdx`*`:`*`blkIdx`* (see "Block ID") in major time steps. `break gcb` inserts a breakpoint before each invocation of a method of the currently selected block (see `gcb`) in major times steps.

`break s:`*`sysIdx`* inserts a breakpoint at each method of the root system or nonvirtual subsystem specified by the system ID: *`sysIdx`*.

Note The `systems` command displays the system IDs for all nonvirtual systems in the currently selected model.

`break gcs` inserts a breakpoint at each method of the currently selected nonvirtual system.

`break model` inserts a breakpoint at each method of the currently selected model.

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`

Introduced before R2006a

bshow

Show specified block

Syntax

```
bshow s:b
```

```
bs s:b
```

Arguments

`s:b`	The block whose system index is `s` and block index is `b`.

Description

The `bshow` command opens the model window containing the specified block and selects the block.

See Also

`slist`

Introduced before R2006a

clear

Clear breakpoints from model

Syntax

```
clear

cl

clear m:mid

clear id

clear <sysIdx:blkIdx | gcb>
```

Arguments

mid	Method ID
id	Breakpoint ID
sysIdx:*blkIdx*	Block ID
gcb	Currently selected block

Description

`clear` clears a breakpoint from the current method.

Instead of `clear`, you can use the short form of `cl` with any of the syntaxes.

`clear m:`*`mid`* clears a breakpoint from the method specified by *`mid`*.

`clear` *`id`* clears the breakpoint specified by the breakpoint ID *`id`*.

`clear` *`sysIdx`*`:`*`blkIdx`* clears any breakpoints set on the methods of the block specified by *`sysIdx:blkIdx`*.

`clear gcb` clears any breakpoints set on the methods of the currently selected block.

See Also

`break` | `bafter` | `slist`

Introduced before R2006a

continue

Continue simulation

Syntax

```
continue
```

```
c
```

Description

The `continue` command continues the simulation from the current breakpoint. If animation mode is not enabled, the simulation continues until it reaches another breakpoint or its final time step. If animation mode is enabled, the simulation continues in animation mode to the first method of the next major time step, ignoring breakpoints.

See Also

`run` | `stop` | `quit`

Introduced before R2006a

disp

Display block's I/O when simulation stops

Syntax

```
disp

d

disp gcb

disp s:b
```

Arguments

`s:b`	The block whose system index is `s` and block index is `b`.
`gcb`	Current block.

Description

The `disp` command registers a block as a display point. The debugger displays the inputs and outputs of all display points in the MATLAB Command Window whenever the simulation halts. Invoking `disp` without arguments shows a list of display points. Use `undisp` to unregister a block.

Instead of `disp`, you can use the short form of `d` with any of the syntaxes.

See Also

`undisp` | `slist` | `probe` | `trace`

Introduced before R2006a

ebreak

Enable (or disable) breakpoint on solver errors

Syntax

```
ebreak
```

```
eb
```

Description

This command causes the simulation to stop if the solver detects a recoverable error in the model. If you do not set or disable this breakpoint, the solver recovers from the error and proceeds with the simulation without notifying you.

See Also

`break` | `bafter` | `tbreak` | `xbreak` | `nanbreak` | `zcbreak` | `rbreak` | `clear` | `where` | `slist` | `systems`

Introduced before R2006a

elist

List simulation methods in order in which they are executed during simulation

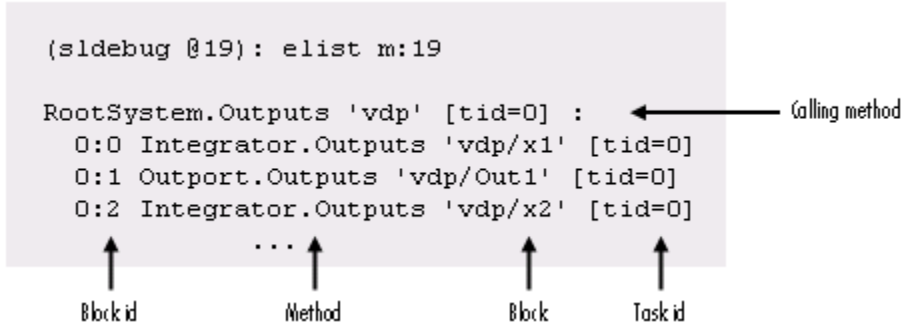
Syntax

```
elist

el

elist m:mid [tid:TID]

elist <gcs | s:sysIdx> [mth] [tid:TID]

elist <gcb | sysIdx:blkIdx> [mth] [tid:TID]
```

Description

Instead of `elist`, you can use the short form of `el` with any of the syntaxes.

`elist m:`*`mid`* lists the methods invoked by the system or nonvirtual subsystem method corresponding to the method id *`mid`* (see the `where` command for information on method IDs), e.g.,



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`

Introduced before R2006a

emode

Toggle model execution between accelerated and normal mode

Syntax

```
emode
```

```
em
```

Description

Toggles the simulation between accelerated and normal mode when using the Accelerator mode in Simulink software. For more information, see “Run Accelerator Mode with the Simulink Debugger”.

Introduced before R2006a

etrace

Enable or disable method tracing

Syntax

```
etrace level level-number
```

```
et level level-number
```

Description

This command enables or disables method tracing, depending on the value of `level`:

Level	Description
0	Turn tracing off.
1	Trace model methods.
2	Trace model and system methods.
3	Trace model, system, and block methods.

When method tracing is on, the debugger prints a message at the command line every time a method of the specified level is entered or exited. The message specifies the current simulation time, whether the simulation is entering or exiting the method, the method id and name, and the name of the model, system, or block to which the method belongs.

See Also

`elist` | `where` | `trace`

Introduced before R2006a

help

Display help for debugger commands

Syntax

```
help
```

```
?
```

```
h
```

Description

The `help` command displays a list of debugger commands in the command window. The list includes the syntax and a brief description of each command.

Introduced before R2006a

nanbreak

Set or clear nonfinite value break mode

Syntax

```
nanbreak
```

```
na
```

Description

The `nanbreak` command causes the debugger to break whenever the simulation encounters a nonfinite (`NaN` or `Inf`) value. If nonfinite break mode is set, `nanbreak` clears it.

More About

- `ebreak`

See Also

`break` | `bafter` | `rbreak` | `tbreak` | `xbreak` | `zcbreak`

Introduced before R2006a

next

Advance simulation to start of next method at current level in model's execution list

Syntax

```
next
```

```
n
```

Description

The `next` command advances the simulation to the start of the next method at the current level in the model's method execution list.

Note The next command has the same effect as the `step over` command. See `step` for more information.

See Also

`step`

Introduced before R2006a

probe

I/O and state data for blocks

Syntax

```
probe
probe s:b
probe gcb
probe level level-type
p
```

Description

`probe` sets the Simulink debugger to interactive probe mode. In this mode, the debugger displays the I/O of a selected block. To exit interactive probe mode, enter a debugger command or press the **Enter** key.

`probe s:b` displays the I/O of the block whose system index is *`s`* and block index is *`b`*.

`probe gcb` displays the I/O of the currently selected block.

`probe level level-type` sets the verbosity level for `probe`, `trace`, and `dis`. If *`level-type`* is `io`, the debugger displays block I/O. If *`level-type`* is `all` (default), the debugger displays all information for the current state of a block, including inputs and outputs, states, and zero crossings.

`p` is the short form of the command.

Examples

Display I/O for the currently selected block `Out2` in the model `vdp` using the Simulink debugger.

1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2 Enter:

```
probe gcb
```

The MATLAB Command Window displays:

```
probe: Data of 0:3 Outport block 'vdp/Out2':
U1     = [0]
```

See Also

`disp` | `trace`

Introduced in R2007a

quit

Stop simulation debugger

Syntax

```
quit
q
```

Description

`quit` stops the Simulink debugger and returns to the MATLAB command prompt.

`q` is the short form of the command.

Examples

Start the Simulink debugger for the model `vdp` and then stop it.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   ```
   quit
   ```

See Also

`stop`

Introduced before R2006a

rbreak

Break simulation before solver reset

Syntax

```
rbreak
rb
```

Description

`rbreak` enables (or disables) a solver reset breakpoint if the breakpoint is disabled (or enabled). The breakpoint causes the debugger to halt the simulation whenever an event requires a solver reset. The halt occurs before the solver resets.

`rb` is the short form of the command.

Examples

Start Simulink debugger for the model `vdp` and a set breakpoint before a solver reset.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> is replaced with the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   ```
   rbreak
   ```

   The MATLAB Command Window displays:

   ```
   Break on solver reset request              : enabled
   ```

See Also

`break` | `bafter` | `nanbreak` | `ebreak` | `tbreak` | `xbreak` | `zcbreak`

Introduced before R2006a

run

Run simulation to completion

Syntax

```
run
r
```

Description

`run` starts the simulation from the current breakpoint to its final time step. It ignores breakpoints and display points.

`r` is the short form of the command

Examples

Continue the simulation for the model `vdp` using the Simulink debugger.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   ```
   run
   ```

See Also

`continue` | `stop` | `quit`

Introduced before R2006a

slist

Sorted list of model blocks

Syntax

```
slist
sli
```

Description

`slist` displays a list of blocks for the root system and each nonvirtual subsystem sorted according to data dependencies and other criteria.

For each system (root or nonvirtual), `slist` displays:

- Title line specifying the name of the system, the number of nonvirtual blocks that the system contains, and the number of blocks in the system that have direct feedthrough ports.
- Entry for each block in the order in which the blocks appear in the sorted list.

For each block entry, `slist` displays the block ID and the name and type of the block. The block ID consists of a system index and a block index separated by a colon (`sysIdx:blkIdx`).

- Block index is the position of the block in the sorted list.
- System index is the order in which the Simulink software generated the system sorted list. The system index has no special significance. It simply allows blocks that appear in the same position in different sorted lists to have unique identifiers.

Simulink software uses sorted lists to create block method execution lists (see `elist`) for root system and nonvirtual subsystem methods. In general, root system and nonvirtual subsystem methods invoke the block methods in the same order as the blocks appear in the sorted list.

Exceptions occur in the execution order of block methods. For example, execution lists for multicast models group together all blocks operating at the same rate and in the same

task. Slower groups appear later than faster groups. The grouping of methods by task can result in a block method execution order that is different from the block sorted order. However, within groups, methods execute in the same order as the corresponding blocks appear in the sorted list.

`sli` is the short form of the command.

Examples

Display a sorted list of the root system in the `vdp` model using the Simulink debugger.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   ```
   slist
   ```

   The MATLAB Command Window displays:

   ```
   ---- Sorted list for 'vdp' [9 nonvirtual blocks, directFeed=0]
     0:0    'vdp/x1' (Integrator)
     0:1    'vdp/Out1' (Outport)
     0:2    'vdp/x2' (Integrator)
     0:3    'vdp/Out2' (Outport)
     0:4    'vdp/Scope' (Scope)
     0:5    'vdp/Fcn' (Fcn)
     0:6    'vdp/Product' (Product)
     0:7    'vdp/Mu' (Gain)
     0:8    'vdp/Sum' (Sum)
   ```

See Also

`systems` | `elist`

Introduced before R2006a

states

Current state values

Syntax

```
states
```

Description

`states` displays a list of the current states of the model. The list includes the index, current value, system:block:element ID, state vector name, and block name for each state.

Examples

Display information about the states for the `vdp` model.

1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2 Enter:

```
states
```

The MATLAB Command Window displays:

```
Continuous States:
Idx  Value                  (system:block:element  Name   'BlockName')
  0  0                      (0:0:0  CSTATE  'vdp/x1')
  1  0                      (0:2:0  CSTATE  'vdp/x2')
```

Introduced before R2006a

status

Debugging options in effect

Syntax

```
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
```

Introduced before R2006a

step

Advance simulation by one or more methods

Syntax

```
step
step in
step over
step out
step top
step blockmth
s
```

Description

`step` or `step in` advances the simulation to the next method in the current time step.

`step over` advances the simulation over the next method.

`step out` advances the simulation the end of the current simulation point hierarchy.

`step top` advances the simulation to the first method executed in the next time step.

`step blockmth` advances the simulation to the next method that operates on a block.

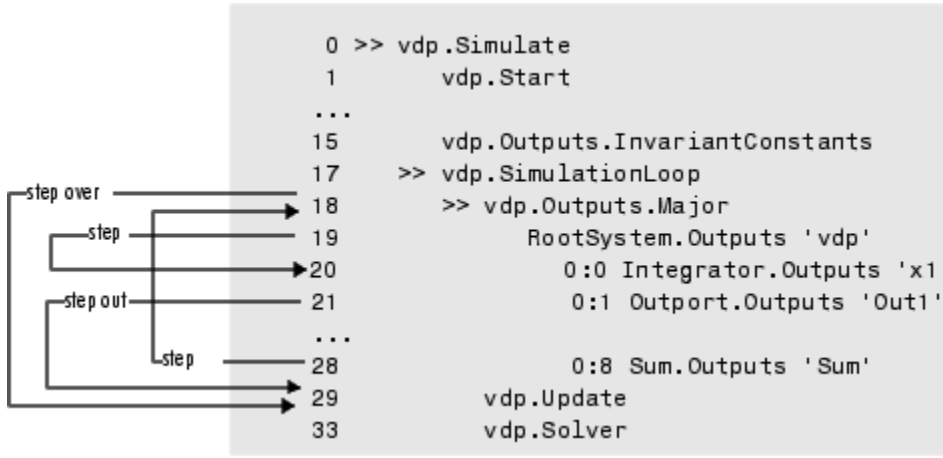
`s` is the short form of the command.

If `step` advances the simulation to the start of a block method, the debugger points at the block on which the method operates.

.

Examples

The following diagram illustrates the effect of various forms of the `step` command for the model `vdp`.



See Also

`next` | `where` | `elist`

Introduced in R2007a

stimes

Model sample times

Syntax

```
stimes
sti
```

Description

`stimes` displays information about the model sample times, including the sample time period, offset, and task ID.

`sti` is the short form of the command.

Examples

Display sample times for the model `vdp` using the Simulink debugger.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   ```
   stimes
   ```

   The MATLAB Command Window displays:

   ```
   --- Sample times  for 'vdp' [Number of sample times = 1]
     1. [0     , 0     ]  tid=0 (continuous sample time)
   ```

Introduced before R2006a

stop

Stop simulation

Syntax

```
stop
```

Description

`stop` stops the simulation of the model you are debugging.

Examples

Start and stop a simulation for the model `vdp` using the Simulink debugger.

1. Start a debugger session. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Start a simulation of the model. Enter:

   ```
   run
   ```

3. Stop the simulation. Enter:

   ```
   stop
   ```

See Also

`continue` | `run` | `quit`

Introduced before R2006a

strace

Set solver trace level

Syntax

```
strace level
i
```

Description

`strace level` causes the solver to display diagnostic information in the MATLAB Command Window, depending on the value of `level`. Values are `0` (no information) or `1` (maximum information about time steps, integration steps, zero crossings, and solver resets).

`i` is the short form of the command.

Examples

Display maximum information about a simulation for the model `vdp` using the Simulink debugger.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'vdp'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Get information about the notation . Enter:

   ```
   help time
   ```

   The MATLAB Command Window displays:

   ```
   Time is displayed as:
    TM = <time while in MajorTimeStep>
    Tm = <time while in MinorTImeStep>
    Tr = <time while in solver reset>
    Tz = <time at or just after zero crossing>
   ```

```
 TzL = <time while in major step just before (at left post of) zero crossing>
 TzR = <time while in major step at or just after (at right post of) zero crosing>
 Ts = <time of successful integration step>
 Tf = <time of failed integration step>
 Tn = <time while in Newton iteration> (when using implicit solvers)
 Tj = <time during Jacobian evaluation> (when using implicit solvers)

Step size is displayed as:
 Hm = <step size at the start of solver phase>
 Hs = <successful  integration step size>
 Hf = <failed integration step size>
 Hn = <step size during Newton iteration> (when using implicit solvers)
 Hz = <value of 'TM - TzL' during zero crossing search>
 Iz= <value of 'Tz - TzL' during zero crossing search>
```

3 Set trace to display all information. Enter:

```
strace 1
```

When diagnostic tracing is on, the debugger displays the sizes of major and minor time steps.

```
[TM = 13.21072088374186 ] Start of Major Time Step
[Tm = 13.21072088374186 ] Start of Minor Time Step
```

The debugger displays integration information. This information includes the time step of the integration method, step size of the integration method, outcome of the integration step, normalized error, and index of the state.

```
[Tm = 13.21072088374186 ] [H  = 0.2751116230148764 ] Begin Integration Step
[Tf = 13.48583250675674 ] [Hf = 0.2751116230148764 ] Fail  [Er = 1.0404e+000]
 [Ix = 1]
[Tm = 13.21072088374186 ] [H  = 0.2183536061326544 ] Retry
[Ts = 13.42907448987452 ] [Hs = 0.2183536061326539 ] Pass  [Er = 2.8856e-001]
 [Ix = 1]
```

For zero crossings, the debugger displays information about the iterative search algorithm when the zero crossing occurred. This information includes the time step of the zero crossing, step size of the zero crossing detection algorithm, length of the time interval bracketing the zero crossing, and a flag denoting the rising or falling direction of the zero crossing.

```
[Tz = 3.615333333333301 ] Detected 1 Zero Crossing Event 0[F]
                          Begin iterative search to bracket zero crossing event
[Tz = 3.621111157580072 ] [Hz = 0.005777824246771424 ] [Iz = 4.2222e-003] 0[F]
[Tz = 3.621116982080098 ] [Hz = 0.005783648746797265 ] [Iz = 4.2164e-003] 0[F]
[Tz = 3.621116987943544 ] [Hz = 0.005783654610242994 ] [Iz = 4.2163e-003] 0[F]
[Tz = 3.621116987943544 ] [Hz = 0.005783654610242994 ] [Iz = 1.1804e-011] 0[F]
[Tz = 3.621116987949452 ] [Hz = 0.005783654616151157 ] [Iz = 5.8962e-012] 0[F]
[Tz = 3.621116987949452 ] [Hz = 0.005783654616151157 ] [Iz = 5.1514e-014] 0[F]
                          End iterative search to bracket zero crossing event
```

When a solver resets occur, the debugger displays the time at which the solver was reset.

```
[Tr = 6.246905153573676 ] Process Solver Reset
[Tr = 6.246905153573676 ] Reset Zero Crossing Cache
[Tr = 6.246905153573676 ] Reset Derivative Cache
```

See Also

atrace | etrace | states | trace | zclist

Introduced before R2006a

systems

List nonvirtual systems of model

Syntax

```
systems
sys
```

Description

`systems` displays the nonvirtual subsystems for a model in the MATLAB Command Window.

`sys` is the short form of the command.

Examples

Display the nonvirtual systems for the model `sldemo_enginewc` using the Simulink debugger.

1. In the MATLAB Command Window, enter:

   ```
   sldebug 'sldemo_enginewc'
   ```

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   ```
   systems
   ```

   The MATLAB Command Window displays the nonvirtual subsystems.

   ```
   0   'sldemo_enginewc'
   1   'sldemo_enginewc/Compression'
   2   'sldemo_enginewc/Controller/TmpAtomicSubsysAtSwitchInport3'
   3   'sldemo_enginewc/Controller/TmpAtomicSubsysAtSwitchInport1'
   4   'sldemo_enginewc/Controller'
   5   'sldemo_enginewc/Throttle & Manifold/Throttle/TmpAtomicSubsysAtSwitchInport1'
   6   'sldemo_enginewc/valve timing/positive edge to dual edge conversion'
   ```

See Also

`slist`

Introduced before R2006a

tbreak

Set or clear time breakpoint

Syntax

```
tbreak

tb

tbreak t
```

Description

The `tbreak` command sets a breakpoint at the specified time step. If a breakpoint already exists at the specified time, `tbreak` clears the breakpoint. If you do not specify a time, `tbreak` toggles a breakpoint at the current time step.

Instead of `tbreak`, you can use the short form of `tb`, with or without `t`.

More About

- `ebreak`

See Also

`break` | `bafter` | `xbreak` | `nanbreak` | `zcbreak` | `rbreak`

Introduced before R2006a

trace

Display block's I/O each time block executes

Syntax

```
trace gcb
trace s:b

tr gcb
trace s:b
```

Arguments

`s:b`	The block whose system index is `s` and block index is `b`.
`gcb`	Current block.

Description

The `trace` command registers a block as a trace point. The debugger displays the I/O of each registered block each time the block executes.

See Also

`disp` | `probe` | `untrace` | `slist` | `strace`

Introduced before R2006a

undisp

Remove block from debugger's list of display points

Syntax

```
undisp gcb

und gcb

undisp s:b

und s:b
```

Arguments

`s:b`	The block whose system index is `s` and block index is `b`.
`gcb`	Current block.

Description

The `undisp` command removes the specified block from the debugger's list of display points.

See Also

`disp` | `slist`

Introduced before R2006a

untrace

Remove block from debugger's list of trace points

Syntax

```
untrace gcb
unt gcb
untrace s:b
unt s:b
```

Arguments

`s:b`	The block whose system index is `s` and block index is `b`.
`gcb`	Current block.

Description

The `untrace` command removes the specified block from the debugger's list of trace points.

See Also

`trace` | `slist`

Introduced before R2006a

where

Display current location of simulation in simulation loop

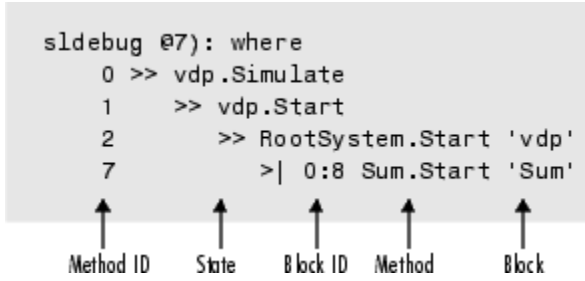
Syntax

```
where [detail]
```

```
w [detail]
```

Description

The `where` command displays the current location of the simulation in the simulation loop, for example,



The display consists of a list of simulation nodes with the last entry being the node that is about to be entered or exited. Each entry contains the following information:

- Method ID

  The method ID identifies a specific invocation of a method.
- A symbol specifying its state:
  - `>>` (active)
  - `>|` (about to be entered)
  - `<|` (about to be exited)
- Name of the method invoked (e.g., RootSystem.Start)

- Name of the block or system on which the method is invoked (e.g., Sum)
- System and block ID (`sysIdx:blkIdx`) of the block on which the method is invoked

  For example, `0:8` indicates that the specified method operates on block `8` of system `0`.

`where detail`, where `detail` is any nonnegative integer, includes inactive nodes in the display.

```
0 >> vdp.Simulate
    1    >> vdp.Start
    2       >> RootSystem.Start 'vdp'
    3          0:4 Scope.Start 'Scope'
    4             0:5 Fcn.Start 'Fcn'
    5             0:6 Product.Start
'Product'
    6             0:7 Gain.Start 'Mu'
    7          >| 0:8 Sum.Start 'Sum'
```

See Also

`step`

Introduced before R2006a

xbreak

Break when debugger encounters step-size-limiting state

Syntax

```
xbreak
```

```
x
```

Description

The `xbreak` command pauses execution of the model when the debugger encounters a state that limits the size of the steps that the solver takes. If `xbreak` mode is already on, `xbreak` turns the mode off.

More About

- `ebreak`

See Also

`break` | `bafter` | `zcbreak` | `tbreak` | `nanbreak` | `rbreak`

Introduced before R2006a

zcbreak

Toggle breaking at nonsampled zero-crossing events

Syntax

```
zcbreak
```

```
zcb
```

Description

The `zcbreak` command causes the debugger to break when a nonsampled zero-crossing event occurs. If zero-crossing break mode is already on, `zcbreak` turns the mode off.

See Also

`break` | `bafter` | `xbreak` | `tbreak` | `nanbreak` | `zclist`

Introduced before R2006a

zclist

List blocks containing nonsampled zero crossings

Syntax

```
zclist
```

```
zcl
```

Description

The `zclist` command displays a list of blocks in which nonsampled zero crossings can occur. The command displays the list in the MATLAB Command Window.

See Also

`zcbreak`

Introduced before R2006a

Simulink Classes

eventData	Provide information about block method execution events
ModelAdvisor.Preferences	Set Model Advisor preferences
Simulink.AliasType	Create alias for signal and parameter data type
Simulink.Annotation	Specify properties of model annotation
Simulink.BlockCompDworkData	Provide postcompilation information about block's DWork vector
Simulink.BlockCompInputPortData	Provide postcompilation information about block input port
Simulink.BlockCompOutputPortData	Provide postcompilation information about block output port
Simulink.BlockData	Provide run-time information about block-related data, such as block parameters
Simulink.BlockPath	Fully specified Simulink block path
Simulink.BlockPortData	Describe block input or output port
Simulink.BlockPreCompInputPortData	Provide precompilation information about block input port
Simulink.BlockPreCompOutputPortData	Provide precompilation information about block output port
Simulink.Bus	Specify properties of signal bus
Simulink.BusElement	Describe element of signal bus

Simulink.CoderInfo	Specify information needed to generate code for signal or parameter
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.MDLInfo	Extract model file information without loading block diagram into memory
getDescription	Extract model file description without loading block diagram into memory
getMetadata	Extract model file metadata without loading block diagram into memory
Simulink.ModelAdvisor	Run Model Advisor from MATLAB file
Simulink.ModelDataLogs	Container for signal data logs of a model
Simulink.SimState.ModelSimState	Access SimState snapshot data
Simulink.ModelWorkspace	Describe model workspace
Simulink.MSFcnRunTimeBlock	Get run-time information about Level-2 MATLAB S-function block
Simulink.NumericType	Specify floating point, integer, or fixed point data type
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.scopes.TimeScopeConfiguration	Configure Scope and Time Scope for programmatic access
Simulink.Signal	Specify attributes of signal

Simulink.SimulationData.BlockPath	Fully specified Simulink block path
Simulink.SimulationData.DataStoreMemory	Container for data store logging information
Simulink.SimulationData.LoggingInfo	Signal logging override settings
Simulink.SimulationData.ModelLoggingInfo	Signal logging override settings for a model
Simulink.SimulationData.SignalLoggingInfo	Signal logging override settings for signal
Simulink.SimulationData.Signal	Container for signal logging information
Simulink.SimulationData.State	State logging element
Simulink.SimulationMetadata	Access metadata of simulation runs
Simulink.SimulationOutput	Access object values of simulation results
Simulink.SubsysDataLogs	Container for subsystem 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.Variant	Specify conditions that control variant selection
Simulink.WorkspaceVar	Contains information about workspace variables and blocks that use them
Simulink.DualScaledParameter	Specify name, value, units, and other properties of Simulink dual-scaled parameter
Simulink.Mask	Control masks programmatically
Simulink.MaskParameter	Control mask parameters programmatically
Simulink.dialog.Control	Create instances of dialog control

Simulink.dialog.Container	Create instances of a container dialog control
Simulink.dialog.Panel	Create an instance of a panel dialog control
Simulink.dialog.Group	Create an instance of a group dialog control
Simulink.dialog.Tab	Create an instance of a tab dialog control
Simulink.dialog.TabContainer	Create an instance of a tab container dialog control
Simulink.dialog.Button	Create a button dialog control
Simulink.dialog.Hyperlink	Create a hyperlink dialog control
Simulink.dialog.Image	Create an image dialog control
Simulink.dialog.Text	Create a text dialog control
Simulink.dialog.parameter.Control	Create a parameter dialog control

eventData

Provide information about block method execution events

Description

Simulink software creates an instance of this class when a block method execution event occurs during simulation and passes it to any listeners registered for the event (see `add_exec_event_listener`). The instance specifies the type of event that occurred and the block whose method execution triggered the event. See "Access Block Data During Simulation" for more information.

Parent

None

Children

None

Property Summary

Name	Description
"Type" on page 5-6	Type of method execution event that occurred.
"Source" on page 5-6	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

Introduced in R2009b

matlab.System class

Package: matlab

Base class for System objects

Description

`matlab.System` is the base class for System objects. In your class definition file, you must subclass your object from this base class (or from another class that derives from this base class). Subclassing allows you to use the implementation and service methods provided by this base class to build your object. Type this syntax as the first line of your class definition file to directly inherit from the `matlab.System` base class, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System
```

Note: You must set `Access = protected` for each `matlab.System` method you use in your code.

Methods

allowModelReferenceDiscreteSampleTimeInheritanceImpl	Model reference sample time inheritance status for discrete sample times
getInputNamesImpl	Names of System block input ports
getOutputNamesImpl	Names of System block output ports
getPropertyGroupsImpl	Property groups for System object display
getSimulateUsingImpl	Specify value for Simulate using parameter
infoImpl	Information about System object
showFiSettingsImpl	Fixed point data type tab visibility for System objects
showSimulateUsingImpl	Simulate Using visibility
stepImpl	System output and state update equations

setupImpl	Initialize System object
resetImpl	Reset System object states
releaseImpl	Release resources
getNumInputsImpl	Number of inputs to step method
getNumOutputsImpl	Number of outputs from `step` method
getDiscreteStateImpl	Discrete state property values
supportsMultipleInstanceImpl	Support System object in Simulink For Each subsystem
validateInputsImpl	Validate inputs to step method
validatePropertiesImpl	Validate property values
processTunedPropertiesImpl	Action when tunable properties change
isInputSizeLockedImpl	Locked input size status
isInactivePropertyImpl	Inactive property status
setProperties	Set property values using name-value pairs
loadObjectImpl	Load System object from MAT file
saveObjectImpl	Save System object in MAT file

Attributes

In addition to the attributes available for MATLAB objects, you can apply the following attributes to any property of a custom System object.

`Nontunable`	After an object is locked (after `step` or `setup` has been called), use `Nontunable` to prevent a user from changing that property value. By default, all properties are tunable. The `Nontunable` attribute is useful to lock a property that has side effects when changed. This attribute is also useful for locking a property value assumed to be constant during processing. You should always specify properties that affect the number of input or output ports as `Nontunable`.
`Logical`	Use `Logical` to limit the property value to a logical, scalar value. Any scalar value that can be converted to a logical is also valid, such as 0 or 1.

`PositiveInteger`	Use `PositiveInteger` to limit the property value to a positive integer value.
`DiscreteState`	Use `DiscreteState` to mark a property so it will display its state value when you use the `getDiscreteState` method.

To learn more about attributes, see "Property Attributes" in the MATLAB Object-Oriented Programming documentation.

Examples

Create a Basic System Object

Create a simple System object, `AddOne`, which subclasses from `matlab.System`. You place this code into a MATLAB file, `AddOne.m`.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access = protected)
       % stepImpl method is called by the step method.
       function y = stepImpl(~,x)
          y = x + 1;
       end
    end
end
```

Use this object by creating an instance of `AddOne`, providing an input, and using the `step` method.

```
hAdder = AddOne;
x = 1;
y = step(hAdder,x)
```

Assign the `Nontunable` attribute to the `InitialValue` property, which you define in your class definition file.

```
properties (Nontunable)
   InitialValue
end
```

See Also

matlab.system.StringSet | matlab.system.mixin.FiniteSource

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes
- "Method Attributes"
- "Define Basic System Objects"
- "Define Property Attributes"

allowModelReferenceDiscreteSampleTimeInheritanceImpl

Class: matlab.System
Package: matlab

Model reference sample time inheritance status for discrete sample times

Syntax

```
flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
```

Description

`flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)` indicates whether a System object in a referenced model is allowed to inherit the sample time of the parent model. Use this method only for System objects that use discrete sample time and are intended for inclusion in Simulink via the `MATLAB System` block.

During model compilation, Simulink sets the model reference sample time inheritance before the System object `setupImpl` method is called.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object handle

Output Arguments

`flag`

Flag indicating whether model reference discrete sample time inheritance is allowed for the `MATLAB System` block containing the System object, returned as a logical value.

The default value for this argument depends on the number of inputs to the System object. To use the default value, you do not need to include the this method in your System object class definition file.

Number of System object Inputs	Default Value and Override Effects
No inputs	**Default:**`false` — Model reference discrete sample time inheritance is not allowed. If your System object uses discrete sample time in its algorithm, override the default by returning `true` from `allowModelReferenceDiscreteSampleTimeInheritanceImpl`.
One or more inputs	**Default:**`true` — If no other Simulink constraint prevents it, model reference sample time inheritance is allowed. If your System object does not use sample time in its algorithm, override the default by returning `false` from `allowModelReferenceDiscreteSampleTimeInheritanceImpl`.

Examples

Set Sample Time Inheritance for System Object

For a System object that has one or more inputs, to disallow model reference discrete sample time inheritance for that object, set the sample time inheritance to `false`. Include this code in your class definition file for the object.

```
methods (Access = protected)
   function flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(~)
      flag = false;
   end
end
```

See Also

matlab.System

How To

- “Set Model Reference Discrete Sample Time Inheritance”
- “Overview of Model Referencing”

- “Inherit Sample Times”

getInputNamesImpl

Class: matlab.System
Package: matlab

Names of System block input ports

Syntax

```
[name1,name2,...] = getInputNamesImpl(obj)
```

Description

`[name1,name2,...] = getInputNamesImpl(obj)` returns the names of the input ports to System object, `obj` implemented in a `MATLAB System` block. The number of returned input names matches the number of inputs returned by the `getNumInputs` method. If you change a property value that changes the number of inputs, the names of those inputs also change.

`getInputNamesImpl` is called by the `getInputNames` method by the MATLAB System block.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

Output Arguments

`name1,name2,...`

Names of the inputs for the specified object, returned as strings

Default: empty string

Examples

Specify Input Port Name

Specify in your class definition file the names of two input ports as `'upper'` and `'lower'`.

```
methods (Access = protected)
   function varargout = getInputNamesImpl(obj)
      numInputs = getNumInputs(obj);
      varargout = cell(1,numInputs);
      varargout{1} = 'upper';
      if numInputs > 1
         varargout{2} = 'lower';
      end
   end
end
```

See Also

getNumInputsImpl | getOutputNamesImpl

How To

- “Validate Property and Input Values”

getOutputNamesImpl

Class: matlab.System
Package: matlab

Names of System block output ports

Syntax

```
[name1,name2,...] = getOutputNamesImpl(obj)
```

Description

`[name1,name2,...] = getOutputNamesImpl(obj)` returns the names of the output ports from System object, `obj` implemented in a `MATLAB System` block. The number of returned output names matches the number of outputs returned by the `getNumOutputs` method. If you change a property value that affects the number of outputs, the names of those outputs also change.

`getOutputNamesImpl` is called by the `getOutputNames` method and by the `MATLAB System` block.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

Output Arguments

`name1,name2,...`

Names of the outputs for the specified object, returned as strings.

Default: empty string

Examples

Specify Output Port Name

Specify the name of an output port as `'count'`.

```
methods (Access = protected)
   function outputName = getOutputNamesImpl(~)
      outputName = 'count';
   end
end
```

See Also

getNumOutputsImpl | getInputNamesImpl

How To

- “Validate Property and Input Values”

getPropertyGroupsImpl

Class: matlab.System
Package: matlab

Property groups for System object display

Syntax

```
group = getPropertyGroupsImpl
```

Description

`group = getPropertyGroupsImpl` returns the groups of properties to display. You define property sections (`matlab.system.display.Section`) and section groups (`matlab.system.display.SectionGroup`) within this method. Sections arrange properties into groups. Section groups arrange sections and properties into groups. If a System object, included through the `MATLAB System` block, has a section, but that section is not in a section group, its properties appear above the block dialog tab panels.

If you do not include a `getPropertyGroupsImpl` method in your code, all public properties are included in the dialog box by default. If you include a `getPropertyGroupsImpl` method but do not list a property, that property does not appear in the dialog box.

When the System object is displayed at the MATLAB command line, the properties are grouped as defined in `getPropertyGroupsImpl`. If your `getPropertyGroupsImpl` defines multiple section groups, only properties from the first section group are displayed at the command line. To display properties in other sections, a link is provided at the end of a System object property display. Group titles are also displayed at the command line. To omit the "Main" title for the first group of properties, set `TitleSource` to `'Auto'` in `matlab.system.display.SectionGroup`.

`getPropertyGroupsImpl` is called by the `MATLAB System` block and when displaying the object at the command line.

Note: You must set `Access = protected` and `Static` for this method.

Output Arguments

`group`

Property group or groups

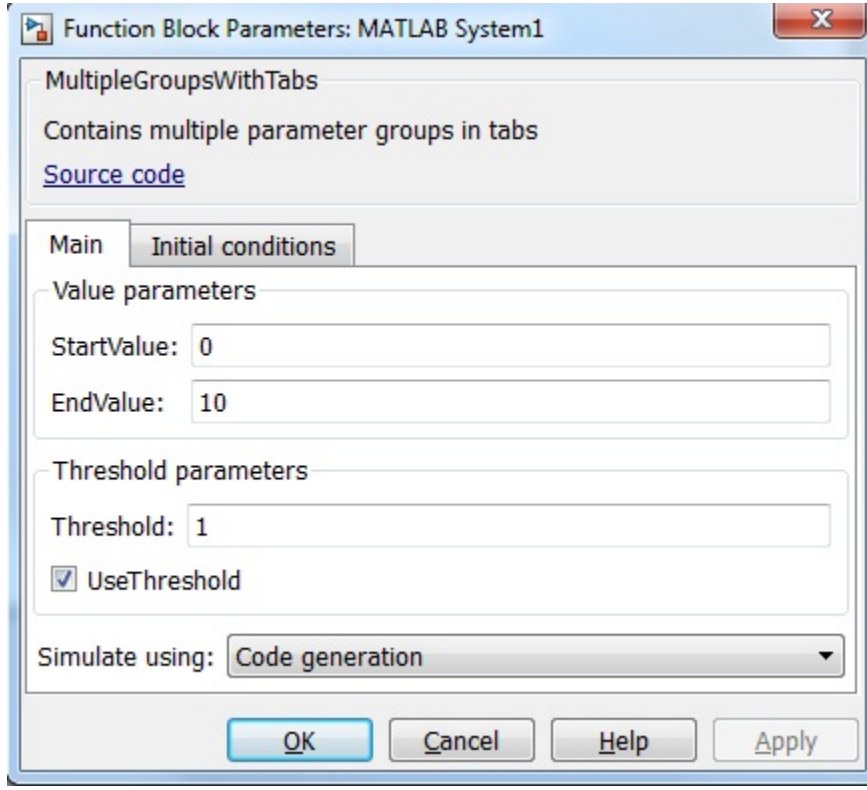
Examples

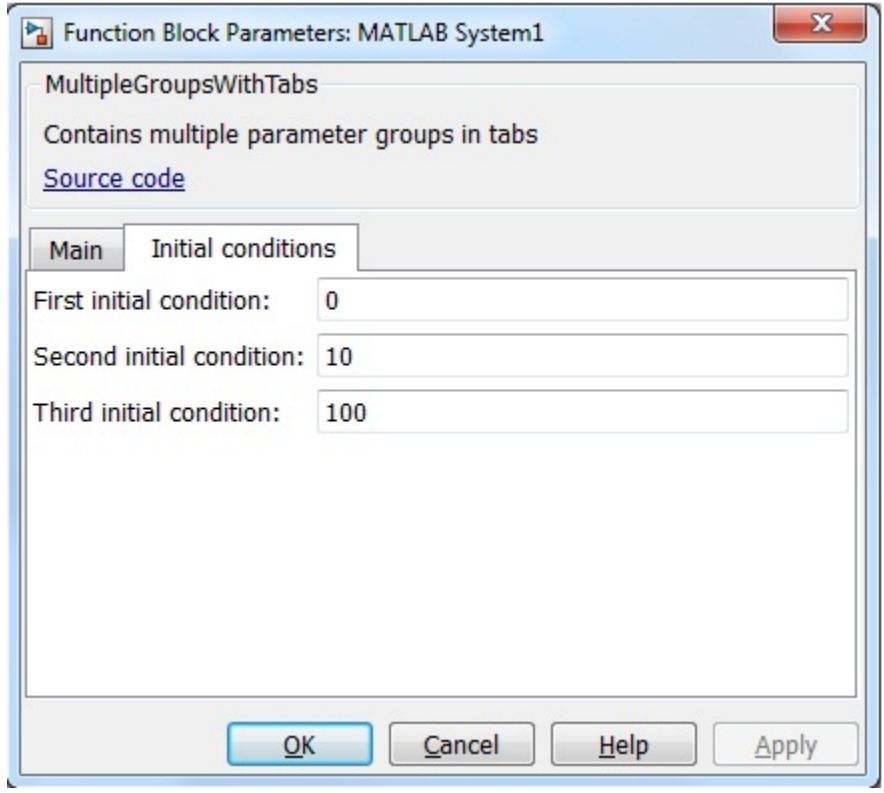
Define Block Dialog Tabs

Define two block dialog tabs, each containing specific properties. For this example, you use the `getPropertyGroupsImpl`, `matlab.system.display.SectionGroup`, and `matlab.system.display.Section` methods in your class definition file.

```
methods (Static, Access = protected)
   function groups = getPropertyGroupsImpl
      valueGroup = matlab.system.display.Section(...
           'Title','Value parameters',...
           'PropertyList',{'StartValue','EndValue'});

      thresholdGroup = matlab.system.display.Section(...
           'Title','Threshold parameters',...
           'PropertyList',{'Threshold','UseThreshold'});

      mainGroup = matlab.system.display.SectionGroup(...
           'Title','Main', ...
           'Sections',[valueGroup,thresholdGroup]);

      initGroup = matlab.system.display.SectionGroup(...
           'Title','Initial conditions', ...
           'PropertyList',{'IC1','IC2','IC3'});

      groups = [mainGroup,initGroup];
   end
end
```

The resulting dialog box appears as follows.


Function Block Parameters: MATLAB System1
MultipleGroupsWithTabs
Contains multiple parameter groups in tabs
Source code
Main
Initial conditions
Value parameters
StartValue:
0
EndValue:
10
Threshold parameters
Threshold:
1
UseThreshold
Simulate using:
Code generation
OK
Cancel
Help
Apply



See Also

matlab.system.display.Header | matlab.system.display.Section | matlab.system.display.SectionGroup

How To

- "Add Property Groups to System Object and MATLAB System Block"

getSimulateUsingImpl

Class: matlab.System
Package: matlab

Specify value for Simulate using parameter

Syntax

```
simmode = getSimulateUsingImpl
```

Description

`simmode = getSimulateUsingImpl` returns a string, `simmode`, that specifies the Simulink simulation mode. The simulation mode restricts your System object to simulation using either code generation or interpreted execution. The associated `showSimulateUsingImpl` method controls whether the **Simulate using** option is displayed on the `MATLAB System` block dialog box.

`getSimulateUsingImpl` is called by the `MATLAB System` block.

Note: You must set `Access = protected` and `Static` for this method.

Output Arguments

`simmode`

Simulation mode, returned as the string `'Code generation'` or `'Interpreted execution'`. If you do not include the `getSimulateUsingImpl` method in your class definition file, the simulation mode is unrestricted. Depending on the value returned by the associated `showSimulateUsingImpl` method, the simulation mode is displayed as either a dropdown list on the dialog box or not at all.

Examples

Specify the Simulation Mode

In the class definition file of your System object, define the simulation mode to display in the `MATLAB System` block. To prevent **Simulate using** from displaying, see `showSimulateUsingImpl`.

```
methods (Static, Access = protected)
   function simMode = getSimulateUsingImplj
      simMode = 'Interpreted execution';
   end
end
```

See Also

showSimulateUsingImpl

How To

- "Control Simulation Type in MATLAB System Block"

infoImpl

Class: matlab.System
Package: matlab

Information about System object

Syntax

```
s = infoImpl(obj,varargin)
```

Description

`s = infoImpl(obj,varargin)` lets you set up information to return about the current configuration of a System object `obj`. This information is returned in a struct from the `info` method. The default `infoImpl` method, which is used if you do not include `infoImpl` in your class definition file, returns an empty struct.

`infoImpl` is called by the `info` method.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

`varargin`

Optional. Allow variable number of inputs

Examples

Specify System object Information

Define the `infoImpl` method to return current count information.

```
methods (Access = protected)
   function s = infoImpl(obj)
      s = struct('Count',obj.pCount);
   end
end
```

How To

- “Define System Object Information”

showFiSettingsImpl

Class: matlab.System
Package: matlab

Fixed point data type tab visibility for System objects

Syntax

```
flag = showFiSettingsImpl
```

Description

`flag = showFiSettingsImpl` specifies whether the Data Types tab appears on the MATLAB System block dialog box. The Data Types tab includes parameters to control processing of fixed point data the MATLAB System block. You cannot specify which parameters appear on the tab. If you implement `showFiSettingsImpl`, the simulation mode is set code generation.

`showFiSettingsImpl` is called by the MATLAB System block.

The parameters that appear on the Data Types tab, which cannot be customized, are

- **Saturate on integer overflow** is a check box to control the action to take on integer overflow for built-in integer types. The default is that the box is checked, which indicates to saturate. This is also the default for when **Same as MATLAB** is selected as the **MATLAB System fimath** option.
- **Treat these inherited Simulink signal types as fi objects** is a pull down that indicates which inherited data types to treat as fi data types. Valid options are `Fixed point` and `Fixed point & integer`. The default value is `Fixed point`.
- **MATLAB System fimath** has two radio button options: **Same as MATLAB** and **Specify Other**. The default, **Same as MATLAB**, uses the current MATLAB fixed-point math settings. **Specify Other** enables the edit box for specifying the desired fixed-point math settings. For information on setting fixed-point math, see `fimath`, in the Fixed-Point Designer documentation.

Note: If you do not want to display the tab, you do not need to implement this method in your class definition file.

You must set `Access = protected` and `Static` for this method.

Output Arguments

`flag`

Flag indicating whether to display the Data Types tab on the `MATLAB System` block mask, returned as a logical scalar value. Returning a `true` value displays the tab. A `false` value does not display the tab.

Default: `false`

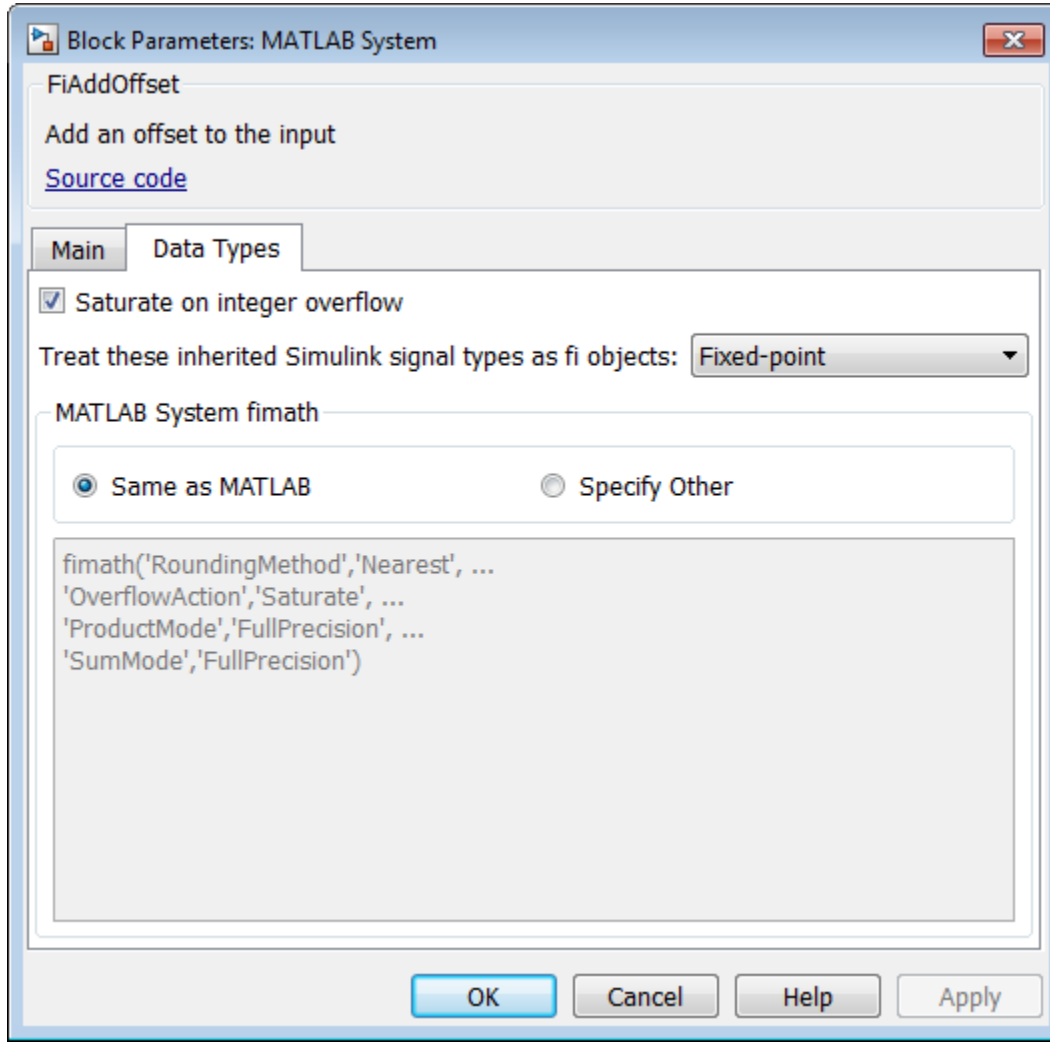
Examples

Show the Data Types Tab

Show the Data Types tab on the `MATLAB System` block dialog box.

```
methods (Static, Access = protected)
   function isVisible = showFiSettingsImpl
     isVisible = true;
   end
end
```

If you set the flag, `isVisible`, to `true`, the tab appears as follows when you add the object to Simulink with the `MATLAB System` block.



How To

- "Add Data Types Tab to MATLAB System Block"

showSimulateUsingImpl

Class: matlab.System
Package: matlab

Simulate Using visibility

Syntax

```
flag = showSimulateUsingImpl
```

Description

`flag = showSimulateUsingImpl` specifies whether the **Simulate using** parameter and dropdown list appear on the `MATLAB System` block dialog box.

`showSimulateUsingImpl` is called by the `MATLAB System` block.

Note: You must set `Access = protected` and `Static` for this method.

Output Arguments

`flag`

Flag indicating whether to display the **Simulate using** parameter and dropdown list on the `MATLAB System` block mask, returned as a logical scalar value. A `true` value displays the parameter and dropdown list. A `false` value hides the parameter and dropdown list.

Default: `true`

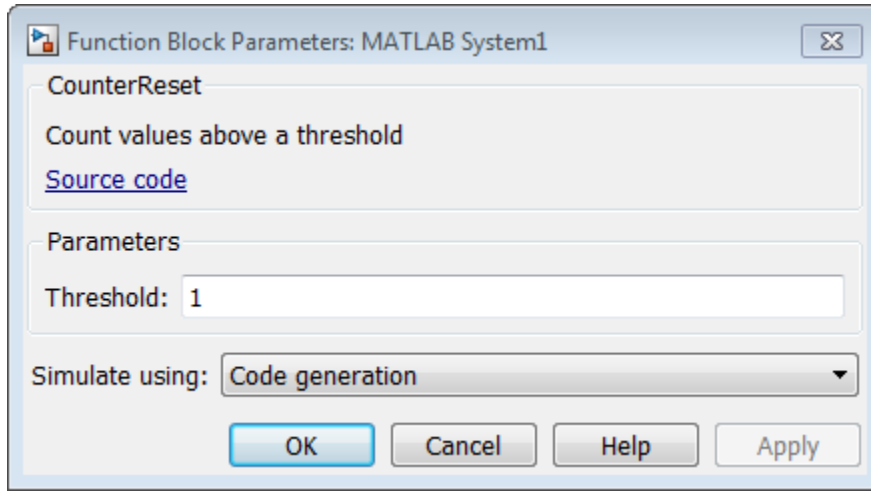
Examples

Hide the Simulate using Parameter

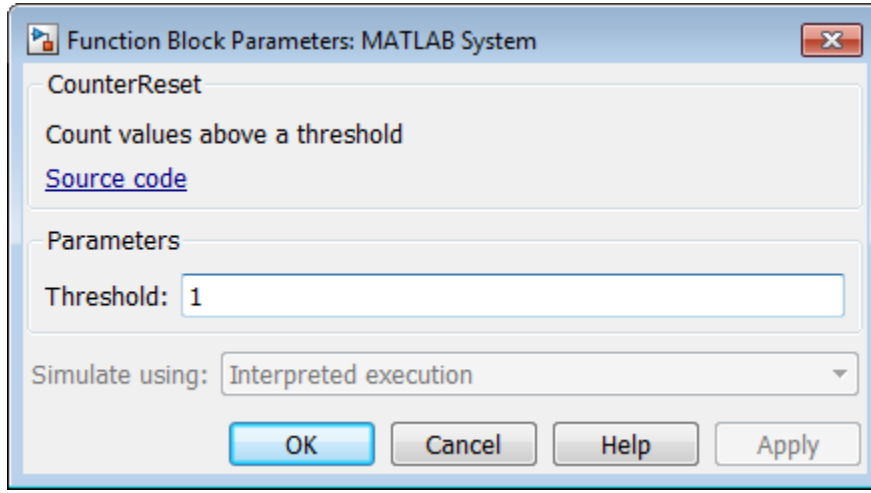
Hide the **Simulate using** parameter on the MATLAB System block dialog box.

```
methods (Static, Access = protected)
  function flag = showSimulateUsingImpl
    flag = false;
  end
end
```

If you set the flag to `true` or omit the `showSimulateUsingImpl` method, which defaults to `true`, the dialog appears as follows when you add the object to Simulink with the `MATLAB System` block.



If you also specify a single value for `getSimulateUsingImpl`, the dialog appears as follows when you add the object to Simulink with the `MATLAB System` block.



See Also

getSimulateUsingImpl

How To

- “Control Simulation Type in MATLAB System Block”

stepImpl

Class: matlab.System
Package: matlab

System output and state update equations

Syntax

```
[output1,output2,...] = stepImpl(obj,input1,input2,...)
```

Description

`[output1,output2,...] = stepImpl(obj,input1,input2,...)` defines the algorithm to execute when you call the `step` method on the specified object `obj`. The `step` method calculates the outputs and updates the object's state values using the inputs, properties, and state update equations.

`stepImpl` is called by the `step` method.

Note: You must set `Access = protected` for this method.

Tips

The number of input arguments and output arguments must match the values returned by the `getNumInputsImpl` and `getNumOutputsImpl` methods, respectively

Input Arguments

`obj`

System object handle

`input1,input2,...`

Inputs to the `step` method

Output Arguments

`output`

Output returned from the `step` method.

Examples

Specify System Object Algorithm

Use the `stepImpl` method to increment two numbers.

```
methods (Access = protected)
   function [y1,y2] = stepImpl(obj,x1,x2)
      y1 = x1 + 1;
      y2 = x2 + 1;
    end
end
```

See Also

getNumInputsImpl | getNumOutputsImpl | validateInputsImpl

How To

- “Define Basic System Objects”
- “Change Number of Step Inputs or Outputs”

setupImpl

Class: matlab.System
Package: matlab

Initialize System object

Syntax

```
setupImpl(obj)
setupImpl(obj,input1,input2,...)
```

Description

`setupImpl(obj)` sets up a System object and implements one-time tasks that do not depend on any inputs to its `stepImpl` method. You typically use `setupImpl` to set private properties so they do not need to be calculated each time `stepImpl` method is called. To acquire resources for a System object, you must use `setupImpl` instead of a constructor.

`setupImpl` executes the first time the `step` method is called on an object after that object has been created. It also executes the next time `step` is called after an object has been released.

`setupImpl(obj,input1,input2,...)` sets up a System object using one or more of the `stepImpl` input specifications. The number and order of inputs must match the number and order of inputs defined in the `stepImpl` method. You pass the inputs into `setupImpl` to use the specifications, such as size and data types in the one-time calculations.

`setupImpl` is called by the `setup` method, which is done automatically as the first subtask of the `step` method on an unlocked System object.

Note: You can omit this method from your class definition file if your System object does not require any setup tasks.

You must set `Access = protected` for this method.

Do not use `setupImpl` to initialize or reset states. For states, use the `resetImpl` method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Tips

To validate properties or inputs use the `validatePropertiesImpl`, `validateInputsImpl`, or `setProperties` methods. Do not include validation in `setupImpl`.

Do not use the `setupImpl` method to set up input values.

Input Arguments

`obj`

System object handle

`input1,input2,...`

Inputs to the `stepImpl` method

Examples

Setup a File for Writing

This example shows how to open a file for writing using the `setupImpl` method in your class definition file.

```
methods (Access = protected)
   function setupImpl(obj)
      obj.pFileID = fopen(obj.Filename,'wb');
      if obj.pFileID < 0
         error('Opening the file failed');
       end
    end
```

```
end
```

Initialize Properties Based on Step Inputs

This example shows how to use `setupImpl` to specify that `step` initialize the properties of an input. In this case, calls to the object's `step` method, which include input `u`, initialize the object states in a matrix of size u.

```
methods (Access = protected)
   function setupImpl(obj, u)
    obj.State = zeros(size(u),'like', u);
  end
end
```

See Also

validatePropertiesImpl | validateInputsImpl | setProperties

How To

- "Initialize Properties and Setup One-Time Calculations"
- "Set Property Values at Construction Time"

resetImpl

Class: matlab.System
Package: matlab

Reset System object states

Syntax

```
resetImpl(obj)
```

Description

`resetImpl(obj)` defines the state reset equations for a System object. Typically you reset the states to a set of initial values, which is useful for initialization at the start of simulation.

`resetImpl` is called by the `reset` method only if the object is locked. The object remains locked after it is reset. `resetImpl` is also called by the `setup` method, after the `setupImpl` method.

Note: You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Input Arguments

`obj`

System object

Examples

Reset Property Value

Use the `reset` method to reset the state of the counter stored in the `pCount` property to zero.

```
methods (Access = protected)
   function resetImpl(obj)
      obj.pCount = 0;
   end
end
```

See Also

releaseImpl

How To

- “Reset Algorithm State”

releaseImpl

Class: matlab.System
Package: matlab

Release resources

Syntax

```
releaseImpl(obj)
```

Description

`releaseImpl(obj)` releases any resources used by the System object, such as file handles. This method also performs any necessary cleanup tasks. To release resources for a System object, you must use `releaseImpl` instead of a destructor.

`releaseImpl` is called by the `release` method. `releaseImpl` is also called when the object is deleted or cleared from memory, or when all references to the object have gone out of scope.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

Examples

Close a File and Release Its Resources

Use the `releaseImpl` method to close a file opened by the System object.

```
methods (Access = protected)
   function releaseImpl(obj)
     fclose(obj.pFileID);
   end
end
```

How To

- “Release System Object Resources”

getNumInputsImpl

Class: matlab.System
Package: matlab

Number of inputs to step method

Syntax

```
num = getNumInputsImpl(obj)
```

Description

`num = getNumInputsImpl(obj)` returns the number of inputs `num` expected by the `step` method. The System object input argument is not included in the count. For example, if your `step` method syntax is `step(h_obj,x1,x2,x3)`, `getNumInputs` returns `3`.

If your `step` method has a variable number of inputs (uses `varargin`), implement the `getNumInputsImpl` method in your class definition file.

If the number of inputs expected by the `step` method is fixed (does not use `varargin`), the default `getNumInputsImpl` determines the required number of inputs directly from the `step` method. In this case, you do not need to include `getNumInputsImpl` in your class definition file.

`getNumInputsImpl` is called by the `getNumInputs` method and by the `setup` method if the number of inputs has not been determined already.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

If you set the return argument, `num`, from an object property, that object property must have the `Nontunable` attribute.

Input Arguments

`obj`

System object

Output Arguments

`num`

Number of inputs expected by the `step` method for the specified object, returned as an integer.

Default: 1

Examples

Set Number of Inputs

Specify the number of inputs (2, in this case) expected by the `step` method.

```
methods (Access = protected)
   function num = getNumInputsImpl(~)
      num = 2;
   end
end
```

Set Number of Inputs to Zero

Specify that the `step` method does not accept any inputs.

```
methods (Access = protected)
   function num = getNumInputsImpl(~)
      num = 0;
   end
end
```

See Also

setupImpl | stepImpl | getNumOutputsImpl

How To

- “Change Number of Step Inputs or Outputs”

getNumOutputsImpl

Class: matlab.System
Package: matlab

Number of outputs from `step` method

Syntax

```
num = getNumOutputsImpl (obj)
```

Description

`num = getNumOutputsImpl (obj)` returns the number of outputs from the `step` method.

If your `step` method has a variable number of outputs (uses `varargout`), implement the `getNumOutputsImpl` method in your class definition file to determine the number of outputs. Use `nargout` in the `stepImpl` method to assign the expected number of outputs.

If the number of outputs expected by the `step` method is fixed (does not use `varargout`), the object determines the required number of outputs from the `step` method. In this case, you do not need to implement the `getNumOutputsImpl` method.

`getNumOutputsImpl` is called by the `getNumOutputs` method, if the number of outputs has not been determined already.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

If you set the return argument, `num`, from an object property, that object property must have the `Nontunable` attribute.

Input Arguments

obj

System object

Output Arguments

num

Number of outputs from the `step` method for the specified object, returned as an integer.

Examples

Set Number of Outputs

Specify the number of outputs (2, in this case) returned from the `step` method.

```
methods (Access = protected)
   function num = getNumOutputsImpl(~)
      num = 2;
   end
end
```

Set Number of Outputs to Zero

Specify that the `step` method does not return any outputs.

```
methods (Access = protected)
   function num = getNumOutputsImpl(~)
      num = 0;
   end
end
```

Use nargout for Variable Number of Outputs

Use `nargout` in the `stepImpl` method when you have a variable number of outputs and will generate code.

```
methods (Access = protected)
```

```
   function varargout = stepImpl(~,varargin)
      for i = 1:nargout
         varargout{i} = varargin{i}+1;
      end
   end
end
```

See Also

stepImpl | getNumInputsImpl | setupImpl

How To

- “Change Number of Step Inputs or Outputs”

getDiscreteStateImpl

Class: matlab.System
Package: matlab

Discrete state property values

Syntax

```
s = getDiscreteStateImpl(obj)
```

Description

`s = getDiscreteStateImpl(obj)` returns a struct `s` of state values. The field names of the struct are the object's `DiscreteState` property names. To restrict or change the values returned by `getDiscreteState` method, you can override this `getDiscreteStateImpl` method.

`getDiscreteStatesImpl` is called by the `getDiscreteState` method, which is called by the `setup` method.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`s`

State values, returned as a struct

Examples

Get Discrete State Values

Use the `getDiscreteStateImpl` method in your class definition file to get the discrete states of the object.

```
methods (Access = protected)
   function s = getDiscreteStateImpl(obj)
   end
end
```

See Also

setupImpl

How To

- "Define Property Attributes"

supportsMultipleInstanceImpl

Class: matlab.System
Package: matlab

Support System object in Simulink For Each subsystem

Syntax

```
flag = supportsMultipleInstanceImpl(obj)
```

Description

`flag = supportsMultipleInstanceImpl(obj)` indicates whether you can use the System object in a Simulink For Each subsystem via the `MATLAB System` block. To enable For Each support, you must include the `supportsMultipleInstanceImpl` in your class definition file and have it return `true`. Do not enable For Each support if your System object allocates exclusive resources that may conflict with other System objects, such as allocating file handles, memory by address, or hardware resources.

During Simulink model compilation and propagation, the `MATLAB System` block calls the `supportMultipleInstance` method, which then calls the `supportsMultipleInstanceImpl` method to determine For Each support.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`flag`

Boolean value indicating whether the System object can be used in a For Each subsystem. The default value, if you do not include the `supportMultipleInstance` method, is `false`.

Examples

Enable For-Each Support for System Object

Specify in your class definition file that the System object can be used in a Simulink For Each subsystem.

```
methods (Access = protected)
   function flag = supportsMultipleInstanceImpl(obj)
       flag = true;
   end
end
```

See Also

matlab.System

How To

- "Enable For Each Subsystem Support"

validateInputsImpl

Class: matlab.System
Package: matlab

Validate inputs to step method

Syntax

```
validateInputsImpl(obj,input1,input2,...)
```

Description

`validateInputsImpl(obj,input1,input2,...)` validates inputs to the `step` method at the beginning of initialization. Validation includes checking data types, complexity, cross-input validation, and validity of inputs controlled by a property value.

`validateInputsImpl` is called by the `setup` method before `setupImpl`. `validateInputsImpl` executes only once.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method. Use the `processTunedPropertiesImpl` method or `setupImpl` method to modify properties.

Input Arguments

`obj`

System object handle

`input1,input2,...`

Inputs to the `setup` method

Examples

Validate Input Type

Validate that the input is numeric.

```
methods (Access = protected)
   function validateInputsImpl(~,x)
      if ~isnumeric(x)
         error('Input must be numeric');
      end
   end
end
```

See Also

validatePropertiesImpl | setupImpl

How To

- “Validate Property and Input Values”

validatePropertiesImpl

Class: matlab.System
Package: matlab

Validate property values

Syntax

```
validatePropertiesImpl(obj)
```

Description

`validatePropertiesImpl(obj)` validates interdependent or interrelated property values at the beginning of object initialization, such as checking that the dependent or related inputs are the same size.

`validatePropertiesImpl` is the first method called by the `setup` method. `validatePropertiesImpl` also is called before the `processTunedPropertiesImpl` method.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method. Use the `processTunedPropertiesImpl` method or `setupImpl` method to modify properties.

Tips

To check if a property has changed since `stepImpl` was last called, use `isChangedProperty(obj,property)` within `validatePropertiesImpl`.

Input Arguments

`obj`

System object handle

Examples

Validate a Property

Validate that the `useIncrement` property is `true` and that the value of the `increment` property is greater than zero.

```
methods (Access = protected)
   function validatePropertiesImpl(obj)
      if obj.useIncrement && obj.increment < 0
         error('The increment value must be positive');
      end
    end
end
```

See Also

processTunedPropertiesImpl | setupImpl | validateInputsImpl

How To

- “Validate Property and Input Values”

processTunedPropertiesImpl

Class: matlab.System
Package: matlab

Action when tunable properties change

Syntax

```
processTunedPropertiesImpl(obj)
```

Description

`processTunedPropertiesImpl(obj)` specifies the actions to perform when one or more tunable property values change. This method is called as part of the next call to the `step` method after a tunable property value changes. A property is tunable only if its `Nontunable` attribute is `false`, which is the default.

`processTunedPropertiesImpl` is called by the `step` method.

Note: You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Tips

Use this method when a tunable property affects the value of a different property.

To check if a property has changed since `stepImpl` was last called, use `isChangedProperty` within `processTunedPropertiesImpl`.

In MATLAB when multiple tunable properties are changed before running the System object, `processTunedPropertiesImpl` is called only once for all the changes. `isChangedProperty` returns `true` for all the changed properties.

In Simulink, when a parameter is changed in a `MATLAB System` block dialog, the next simulation step calls `processTunedPropertiesImpl` before calling `stepImpl`. All tunable parameters are considered changed and `processTunedPropertiesImpl` method is called for each of them. `isChangedProperty` returns `true` for all the dialog properties.

Input Arguments

`obj`

System object

Examples

Specify Action When Tunable Property Changes

Use `processTunedPropertiesImpl` to recalculate the lookup table if the value of either the `NumNotes` or `MiddleC` property changes before the next call to the `step` method. `propChange` indicates if either property has changed.

```
methods (Access = protected)
   function processTunedPropertiesImpl(obj)
      propChange = isChangedProperty(obj,obj.NumNotes) ||...
         isChangedProperty(obj,obj.MiddleC)
      if propChange
         obj.pLookupTable = obj.MiddleC * (1+log(1:obj.NumNotes)/log(12));
      end
    end
end
```

See Also

validatePropertiesImpl | setProperties

How To

- “Validate Property and Input Values”
- “Define Property Attributes”

isInputSizeLockedImpl

Class: matlab.System
Package: matlab

Locked input size status

Syntax

```
flag = isInputSizeLockedImpl(obj,i)
```

Description

`flag = isInputSizeLockedImpl(obj,i)` indicates whether the i^{th} input port to the `step` method has its size locked. If `flag` is `true`, the size is locked and inputs to the System object cannot change size while the object is locked. If `flag` is `false`, the input is variable size and is not locked, In the unlocked case, the size of inputs to the object can change while the object is running and locked.

`isInputSizeLockedImpl` executes once for each input during System object initialization.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

`i`

`step` method input port number

Output Arguments

`flag`

Flag indicating whether the size of inputs to the specified port is locked, returned as a logical scalar value. If the value of `isInputSizeLockedImpl` is `true`, the size of the current input to that port is compared to the first input to that port. If the sizes do not match, an error occurs.

Default: `false`

Examples

Check If Input Size Is Locked

Specify in your class definition file to check whether the size of the System object input is locked.

```
methods (Access = protected)
   function flag = isInputSizeLockedImpl(~,index)
      flag = true;
   end
end
```

See Also

matlab.System

How To

- "Specify Locked Input Size"

isInactivePropertyImpl

Class: matlab.System
Package: matlab

Inactive property status

Syntax

```
flag = isInactivePropertyImpl(obj,prop)
```

Description

`flag = isInactivePropertyImpl(obj,prop)` specifies whether a public, non-state property is inactive for the current object configuration. An *inactive property* is a property that is not relevant to the object, given the values of other properties. Inactive properties are not shown if you use the `disp` method to display object properties. If you attempt to use public access to directly access or use `get` or `set` on an inactive property, a warning occurs.

`isInactiveProperty` is called by the `disp` method and by the `get` and `set` methods.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object handle

`prop`

Public, non-state property name

Output Arguments

`flag`

Inactive status Indicator of the input property `prop` for the current object configuration, returned as a logical scalar value

Examples

Specify When a Property Is Inactive

Display the `InitialValue` property only when the `UseRandomInitialValue` property value is `false`.

```
methods (Access = protected)
  function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
      flag = obj.UseRandomInitialValue;
    else
      flag = false;
    end
  end
end
```

See Also

setProperties

How To

- “Hide Inactive Properties”

setProperties

Class: matlab.System
Package: matlab

Set property values using name-value pairs

Syntax

```
setProperties(obj,numargs,name1,value1,name2,value2,...)
setProperties(obj,numargs,arg1,...,argN,propvalname1,...propvalnameN)
```

Description

`setProperties(obj,numargs,name1,value1,name2,value2,...)` provides the name-value pair inputs to the System object constructor. Use this syntax if every input must specify both name and value.

`setProperties(obj,numargs,arg1,...,argN,propvalname1,...propvalnameN)` provides the value-only inputs, which you can follow with the name-value pair inputs to the System object during object construction. Use this syntax if you want to allow users to specify one or more inputs by their values only.

Input Arguments

`obj`

System object

`numargs`

Number of inputs passed in by the object constructor

`name1,name2,...`

Name of property

`value1,value2,...`

Value of the property

`arg1,...argN`

Value of property (for value-only input to the object constructor)

`propvalname1,...propvalnameN`

Name of the value-only property

Examples

Setup Value-Only Inputs

Set up an object so users can specify value-only inputs for `VProp1`, `VProp2`, and other property values via name-value pairs when constructing the object.

```
methods
   function obj = MyFile(varargin)
      setProperties(obj,nargin,varargin{:},'VProp1','VProp2');
   end
end
```

How To

- “Set Property Values at Construction Time”

loadObjectImpl

Class: matlab.System
Package: matlab

Load System object from MAT file

Syntax

```
loadObjectImpl(obj)
```

Description

`loadObjectImpl(obj)` loads a saved System object, `obj`, from a MAT file. Your `loadObjectImpl` method should correspond to your `saveObjectImpl` method to ensure that all saved properties and data are loaded.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

Examples

Load System object

Load a saved System object. In this example, the object contains a child object, protected and private properties, and a discrete state. It also saves states if the object is locked and calls the `loadObjectImpl` method from the `matlab.System` class.

```
methods (Access = protected)
```

```
  function loadObjectImpl(obj,s,wasLocked)
    obj.child = matlab.System.loadObject(s.child);

    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    if wasLocked
      obj.state = s.state;
    end

    loadObjectImpl@matlab.System(obj,s,wasLocked);
  end
end
```

See Also

saveObjectImpl

How To

- "Load System Object"
- "Save System Object"

saveObjectImpl

Class: matlab.System
Package: matlab

Save System object in MAT file

Syntax

```
saveObjectImpl(obj)
```

Description

`saveObjectImpl(obj)` defines the System object `obj` property and state values to be saved in a MAT file when a user calls `save` on that object. `save` calls `saveObject`, which then calls `saveObjectImpl`. To save a System object in generated code, the object must be unlocked and it cannot contain or be a child object.

If you do not define a `saveObjectImpl` method for your System object class, only public properties and properties with the `DiscreteState` attribute are saved.

To save any private or protected properties or state information, you must define a `saveObjectImpl` in your class definition file.

End users can use `load`, which calls `loadObjectImpl` to load a System object into their workspace.

Tip Save the state of an object only if the object is locked. When the user loads that saved object, it loads in that locked state.

To save child object information, use the associated `saveObject` method within the `saveObjectImpl` method.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

Examples

Define Property and State Values to Save

Define what is saved for the System object. Call the base class version of `saveObjectImpl` to save public properties. Then, save any child System objects and any protected and private properties. Finally, save the state if the object is locked.

```
methods (Access = protected)
  function s = saveObjectImpl(obj)
    s = saveObjectImpl@matlab.System(obj);
    s.child = matlab.System.saveObject(obj.child);
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;
    if isLocked(obj)
      s.state = obj.state;
    end
  end
end
```

See Also

loadObjectImpl

How To

- “Save System Object”
- “Load System Object”

matlab.system.mixin.FiniteSource class

Package: matlab.system.mixin

Finite source mixin class

Description

`matlab.system.mixin.FiniteSource` is a class that defines the `isDone` method, which reports the state of a finite data source, such as an audio file.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System &...
    matlab.system.mixin.FiniteSource
```

Methods

isDoneImpl	End-of-data flag

See Also

matlab.System

Tutorials

- "Define Finite Source Objects"

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes

isDoneImpl

Class: matlab.system.mixin.FiniteSource
Package: matlab.system.mixin

End-of-data flag

Syntax

```
status = isDoneImpl(obj)
```

Description

`status = isDoneImpl(obj)` indicates if an end-of-data condition has occurred. The `isDone` method should return `false` when data from a finite source has been exhausted, typically by having read and output all data from the source. You should also define the result of future reads from an exhausted source in the `isDoneImpl` method.

`isDoneImpl` is called by the `isDone` method.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object handle

Output Arguments

`status`

Logical value, `true` or `false`, that indicates if an end-of-data condition has occurred or not, respectively.

Examples

Check for End-of-Data

Set up the `isDoneImpl` method in your class definition file so the `isDone` method checks whether the object has completed eight iterations.

```
methods (Access = protected)
   function bdone = isDoneImpl(obj)
      bdone = obj.NumIters==8;
   end
end
```

See Also

matlab.system.mixin.FiniteSource

How To

- "Define Finite Source Objects"

matlab.system.StringSet class

Package: matlab.system

Set of valid string values

Description

`matlab.system.StringSet` defines a list of valid string values for a property. This class validates the string in the property and enables tab completion for the property value. A *StringSet* allows only predefined or customized strings as values for the property.

A `StringSet` uses two linked properties, which you must define in the same class. One is a public property that contains the current string value. This public property is displayed to the user. The other property is a hidden property that contains the list of all possible string values. This hidden property should also have the transient attribute so its value is not saved to disk when you save the System object.

The following considerations apply when using StringSets:

- The string property that holds the current string can have any name.
- The property that holds the `StringSet` must use the same name as the string property with the suffix "Set" appended to it. The string set property is an instance of the `matlab.system.StringSet` class.
- Valid strings, defined in the `StringSet`, must be declared using a cell array. The cell array cannot be empty nor can it have any empty strings. Valid strings must be unique and are case-sensitive.
- The string property must be set to a valid `StringSet` value.

Examples

Set String Property Values

Set the string property, `Flavor`, and the `StringSet` property, `FlavorSet` in your class definition file.

```
properties
  Flavor = 'Chocolate';
end

properties (Hidden,Transient)
  FlavorSet = ...
    matlab.system.StringSet({'Vanilla','Chocolate'});
end
```

See Also

matlab.System

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Limit Property Values to Finite String Set”

matlab.system.mixin.CustomIcon class

Package: matlab.system.mixin

Custom icon mixin class

Description

`matlab.system.mixin.CustomIcon` is a class that defines the `getIcon` method. This method customizes the name of the icon used for the System object implemented through a `MATLAB System` block.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system &...
    matlab.system.mixin.CustomIcon
```

Methods

getIconImpl	Name to display as block icon

See Also

matlab.System

Tutorials

- “Define MATLAB System Block Icon”

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes

getIconImpl

Class: matlab.system.mixin.CustomIcon
Package: matlab.system.mixin

Name to display as block icon

Syntax

```
icon = getIconImpl(obj)
```

Description

`icon = getIconImpl(obj)` returns the string or cell array of strings to display on the block icon of the System object implemented through the `MATLAB System` block. If you do not specify the `getIconImpl` method, the block displays the class name of the System object as the block icon. For example, if you specify `pkg.MyObject` in the `MATLAB System` block, the default icon is labeled `My Object`

`getIconImpl` is called by the `getIcon` method, which is used by the MATLAB System block during Simulink model compilation.

Note: You must set `Access = protected` for this method.

Input Arguments

`obj`

System object handle

Output Arguments

`icon`

String or cell array of strings to display as the block icon. Each cell is displayed as a separate line.

Examples

Add System Block Icon Name

Specify in your class definition file the name of the block icon as `'Enhanced Counter'` using two lines.

```
methods (Access = protected)
   function icon = getIconImpl(~)
      icon = {'Enhanced','Counter'};
   end
end
```

See Also

matlab.system.mixin.CustomIcon

How To

- “Define MATLAB System Block Icon”

matlab.system.display.Header class

Package: matlab.system.display

Header for System objects properties

Syntax

```
matlab.system.display.Header(N1,V1,...Nn,Vn)
matlab.system.display.Header(Obj,...)
```

Description

`matlab.system.display.Header(N1,V1,...Nn,Vn)` defines a header for the System object, with the header properties defined in Name-Value (`N`,`V`) pairs. You use `matlab.system.display.Header` within the `getHeaderImpl` method. The available header properties are

- `Title` — Header title string. The default value is an empty string.
- `Text` — Header description text string. The default value is an empty string.
- `ShowSourceLink` — Show link to source code for the object.

`matlab.system.display.Header(Obj,...)` creates a header for the specified System object (`Obj`) and sets the following property values:

- `Title` — Set to the `Obj` class name.
- `Text` — Set to help summary for `Obj`.
- `ShowSourceLink` — Set to `true` if `Obj` is MATLAB code. In this case, the **Source Code** link is displayed. If `Obj` is P-coded and the source code is not available, set this property to `false`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

Methods

getHeaderImpl	Header for System object display

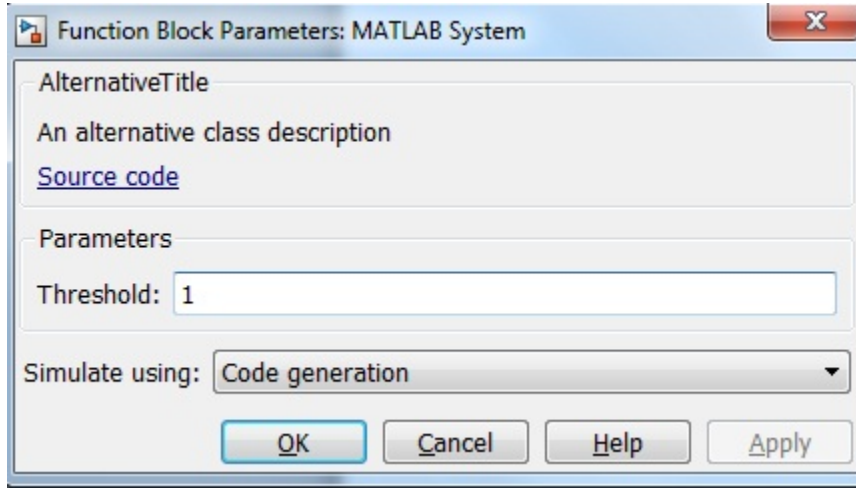
Examples

Define System Block Header

Define a header in your class definition file.

```
methods (Static, Access = protected)
   function header = getHeaderImpl
      header = matlab.system.display.Header(mfilename('class'), ...
          'Title','AlternativeTitle',...
          'Text','An alternative class description');
   end
end
```

The resulting output appears as follows. In this case, **Source code** appears because the `ShowSourceLink` property was set to `true`.



See Also

matlab.system.display.Section | matlab.system.display.SectionGroup

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Add Header to MATLAB System Block”

getHeaderImpl

Class: matlab.system.display.Header
Package: matlab.system.display

Header for System object display

Syntax

```
header = getHeaderImpl
```

Description

`header = getHeaderImpl` returns the header to display for the System object. If you do not specify the `getHeaderImpl` method, no title or text appears for the header in the block dialog box.

`getHeaderImpl` is called by the `MATLAB System` block

Note: You must set `Access = protected` and `Static` for this method.

Output Arguments

`header`

Header text

Examples

Define Header for System Block Dialog Box

Define a header in your class definition file for the `EnhancedCounter` System object.

```
methods (Static, Access = protected)
```

```
    function header = getHeaderImpl
       header = matlab.system.display.Header('EnhancedCounter',...
           'Title','Enhanced Counter');
    end
end
```

See Also

getPropertyGroupsImpl

How To

- “Add Header to MATLAB System Block”

matlab.system.display.Section class

Package: matlab.system.display

Property group section for System objects

Syntax

```
matlab.system.display.Section(N1,V1,...Nn,Vn)
matlab.system.display.Section(Obj,...)
```

Description

`matlab.system.display.Section(N1,V1,...Nn,Vn)` creates a property group section for displaying System object properties, which you define using property Name-Value pairs (`N`,`V`). You use `matlab.system.display.Section` to define property groups using the `getPropertyGroupsImpl` method. The available Section properties are

- `Title` — Section title string. The default value is an empty string.
- `TitleSource` — Source of section title string. Valid values are `'Property'` and `'Auto'`. The default value is `'Property'`, which uses the string from the `Title` property. If the `Obj` name is given, the default value is `Auto`, which uses the `Obj` name.
- `Description` — Section description string. The default value is an empty string.
- `PropertyList` — Section property list as a cell array of property names. The default value is an empty array. If the `Obj` name is given, the default value is all eligible display properties.

Note: Certain properties are not eligible for display either in a dialog box or in the System object summary on the command-line. Property types that cannot be displayed are: hidden, abstract, private or protected access, discrete state, and continuous state. Dependent properties do not display in a dialog box, but do display in the command-line summary.

`matlab.system.display.Section(Obj,...)` creates a property group section for the specified System object (`Obj`) and sets the following property values:

- `TitleSource` — Set to `'Auto'`, which uses the `Obj` name.
- `PropertyList` — Set to all publically-available properties in the `Obj`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

Methods

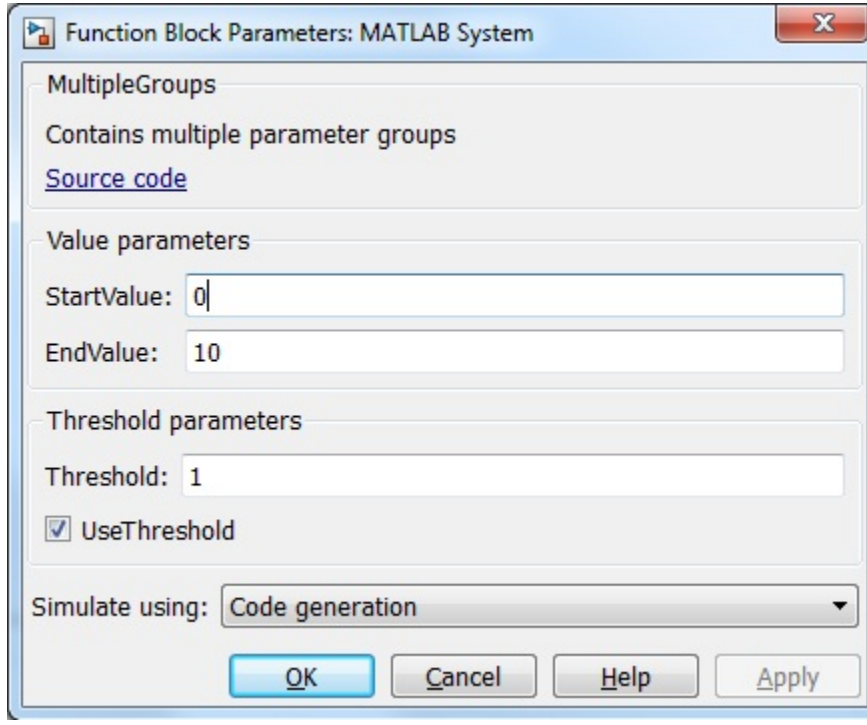
Examples

Define Property Groups

Define two property groups in your class definition file by specifying their titles and property lists.

```
methods (Static, Access = protected)
   function groups = getPropertyGroupsImpl
      valueGroup = matlab.system.display.Section(...
           'Title','Value parameters',...
           'PropertyList',{'StartValue','EndValue'});

      thresholdGroup = matlab.system.display.Section(...
           'Title','Threshold parameters',...
           'PropertyList',{'Threshold','UseThreshold'});
      groups = [valueGroup,thresholdGroup];
   end
end
```

When you specify the System object in the `MATLAB System` block, the resulting dialog box appears as follows.



See Also

matlab.system.display.Header | matlab.system.display.SectionGroup | getPropertyGroupsImpl

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes
- "Add Property Groups to System Object and MATLAB System Block"

matlab.system.display.Action class

Package: matlab.system.display

Custom button

Syntax

```
matlab.system.display.Action(action)
matlab.system.display.Action(action,Name,Value)
```

Description

`matlab.system.display.Action(action)` defines a button to display on the `MATLAB System` block. This button executes a function by launching a System object method or invoking any MATLAB function or code.

A typical button function launches a figure. The launched figure is decoupled from the block dialog box. Changes to the block are not synced to the displayed figure.

You define `matlab.system.display.Action` within the `getPropertyGroupsImpl` method in your class definition file. You can define multiple buttons using separate instances of `matlab.system.display.Action` in your class definition file.

`matlab.system.display.Action(action,Name,Value)` includes `Name,Value` pair arguments, which you can use to specify any properties.

Input Arguments

`action`

Action taken when the user presses the specified button on the `MATLAB System` block dialog. The action is defined as a function handle or as a MATLAB command string. If you define the action as a function handle, the function definition must define two inputs. These inputs are a `matlab.system.display.ActionData` object and a System object instance, which can be used to invoke a method.

A `matlab.system.display.ActionData` object is the callback object for a display action. You use the `UserData` property of `matlab.system.display.ActionData` to store persistent data, such as a figure handle.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

You specify these properties as part of the input using `Name,Value` pair arguments. Optionally, you can define them using `object.property` syntax.

- `ActionCalledFcn` — Action to take when the button is pressed. You cannot specify this property using a Name-Value pair argument.
- `Label` — Text string to display on the button. The default value is an empty string.
- `Description` — Text string for the button tooltip. The default value is an empty string.
- `Placement` — Text string indicating where on a separate row in the property group to place the button. Valid values are `'first'`, `'last'`, or a property name. If you specify a property name, the button is placed above that property. The default value is `'last'`.
- `Alignment` — Text string indicating how to align the button. Valid values are `'left'` and `'right'`. The default value is `'left'`.

Examples

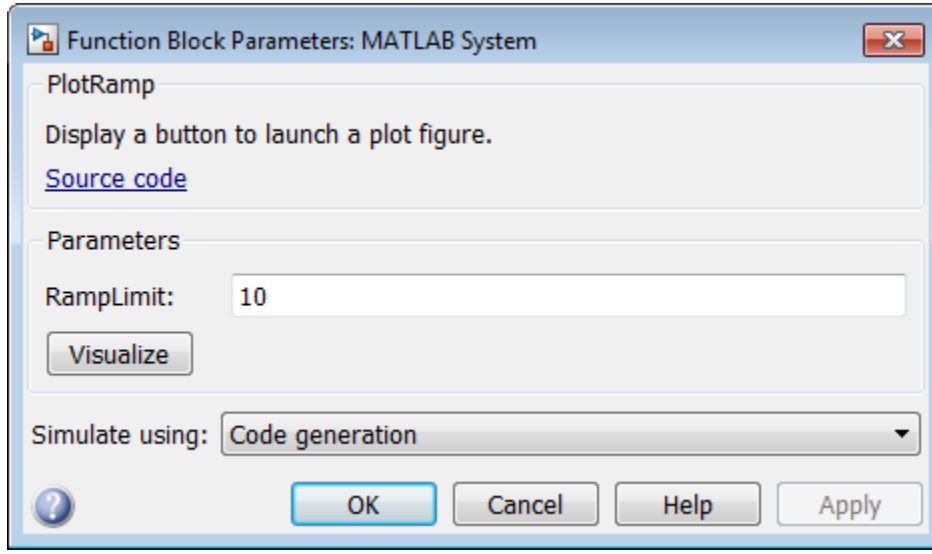
Define Button on MATLAB System Block

Define a **Visualize** button and its associated function to open a figure that plots a ramp using the parameter values in the block dialog.

```
methods(Static,Access = protected)
```

```
  function group = getPropertyGroupsImpl
  group = matlab.system.display.Section(mfilename('class'));
  group.Actions = matlab.system.display.Action(@(~,obj)...
      visualize(obj),'Label','Visualize');
  end
end

methods
  function obj = PlotRamp(varargin)
    setProperties(obj,nargin,varargin{:});
  end

  function visualize(obj)
    figure;
    d = 1:obj.RampLimit;
    plot(d);
  end
end
```

When you specify the System object in the `MATLAB System` block, the resulting block dialog box appears as follows.



To open the same figure, rather than multiple figures, when the button is pressed more than once, use this code instead.

```
methods(Static,Access = protected)
  function group = getPropertyGroupsImpl
    group = matlab.system.display.Section(mfilename('class'));
    group.Actions = matlab.system.display.Action(@(actionData,obj)...
       visualize(obj,actionData),'Label','Visualize');
  end
end

methods
  function obj = ActionDemo(varargin)
    setProperties(obj,nargin,varargin{:});
  end

  function visualize(obj,actionData)
    f = actionData.UserData;
    if isempty(f) || ~ishandle(f)
      f = figure;
      actionData.UserData = f;
    else
        figure(f); % Make figure current
    end

    d = 1:obj.RampLimit;
    plot(d);
  end
end
```

See Also

matlab.System.getPropertyGroupsImpl | matlab.system.display.Section | matlab.system.display.SectionGroup

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Add Button to MATLAB System Block”

matlab.system.display.SectionGroup class

Package: matlab.system.display

Section group for System objects

Syntax

```
matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)
matlab.system.display.SectionGroup(Obj,...)
```

Description

`matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)` creates a group for displaying System object properties and display sections created with `matlab.system.display.Section`. You define such sections or properties using property Name-Value pairs (`N`,`V`). A section group can contain both properties and sections. You use `matlab.system.display.SectionGroup` to define section groups using the `getPropertyGroupsImpl` method. Section groups display as separate tabs in the `MATLAB System` block. The available Section properties are

- `Title` — Group title string. The default value is an empty string.
- `TitleSource` — Source of group title string. Valid values are `'Property'` and `'Auto'`. The default value is `'Property'`, which uses the string from the `Title` property. If the `Obj` name is given, the default value is `Auto`, which uses the `Obj` name. In the System object property display at the MATLAB command line, you can omit the default "Main" title for the first group of properties by setting `TitleSource` to `'Auto'`.
- `Description` — Group or tab description that appears above any properties or panels. The default value is an empty string.
- `PropertyList` — Group or tab property list as a cell array of property names. The default value is an empty array. If the `Obj` name is given, the default value is all eligible display properties.
- `Sections` — Group sections as an array of section objects. If the `Obj` name is given, the default value is the default section for the `Obj`.

`matlab.system.display.SectionGroup(Obj,...)` creates a section group for the specified System object (`Obj`) and sets the following property values:

- `TitleSource` — Set to `'Auto'`.
- `Sections` — Set to `matlab.system.display.Section` object for `Obj`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

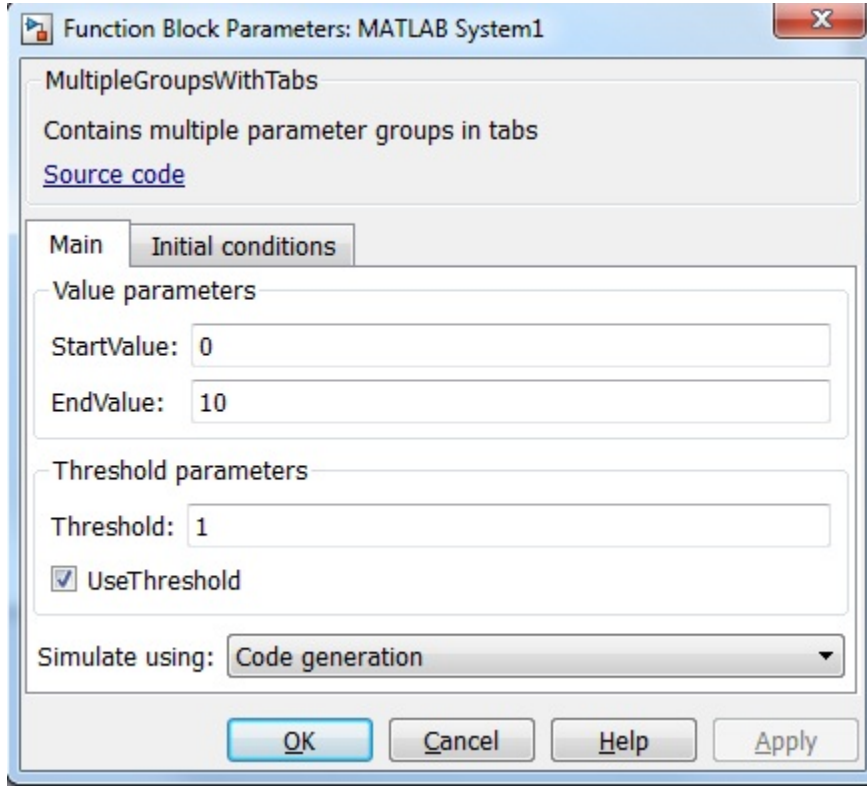
Examples

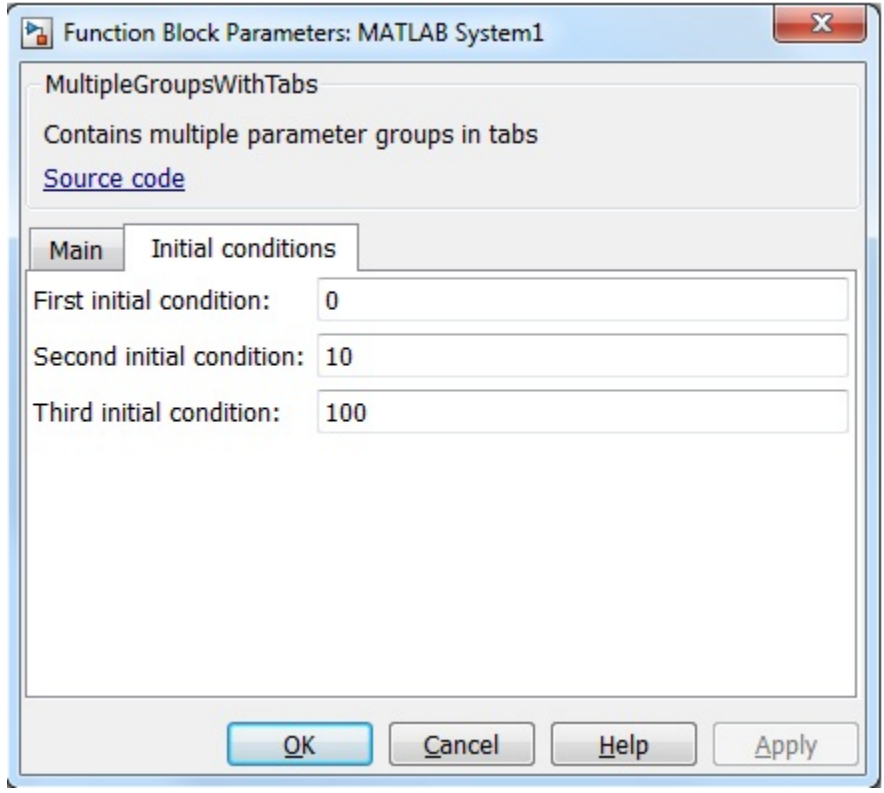
Define Block Dialog Tabs

Define in your class definition file two tabs, each containing specific properties. For this example, you use the `matlab.system.display.SectionGroup`, `matlab.system.display.Section`, and `getPropertyGroupsImpl` methods.

```
methods (Static, Access = protected)
   function groups = getPropertyGroupsImpl
      valueGroup = matlab.system.display.Section(...
           'Title','Value parameters',...
           'PropertyList',{'StartValue','EndValue'});

      thresholdGroup = matlab.system.display.Section(...
           'Title','Threshold parameters',...
           'PropertyList',{'Threshold','UseThreshold'});

      mainGroup = matlab.system.display.SectionGroup(...
           'Title','Main', ...
           'Sections',[valueGroup,thresholdGroup]);

      initGroup = matlab.system.display.SectionGroup(...
           'Title','Initial conditions', ...
           'PropertyList',{'IC1','IC2','IC3'});

      groups = [mainGroup,initGroup];
   end
end
```

The resulting dialog appears as follows when you add the object to Simulink with the `MATLAB System` block.


Function Block Parameters: MATLAB System1
MultipleGroupsWithTabs
Contains multiple parameter groups in tabs
Source code
Main
Initial conditions
Value parameters
StartValue:
0
EndValue:
10
Threshold parameters
Threshold:
1
UseThreshold
Simulate using:
Code generation
OK
Cancel
Help
Apply



See Also

matlab.system.display.Header | matlab.system.display.Section | getPropertyGroupsImpl

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes
- "Add Property Groups to System Object and MATLAB System Block"

matlab.system.mixin.Propagates class

Package: matlab.system.mixin

Signal characteristics propagation mixin class

Description

`matlab.system.mixin.Propagates` defines the output size, data type, and complexity of a System object. Use this mixin class and its methods when you will include your System object in Simulink via the `MATLAB System` block. This mixin is called by the `MATLAB System` block during Simulink model compilation.

Implement the methods of this class when Simulink cannot infer the output specifications directly from the inputs or when you want bus support. If you do not include this mixin, Simulink cannot propagate the output or bus data type, an error occurs.

To use this mixin, subclass from this `matlab.system.mixin.Propagates` in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file. `ObjectName` is the name of your System object.

```
classdef ObjectName < matlab.System &...
    matlab.system.mixin.Propagates
```

Methods

getDiscreteStateSpecificationImpl	Discrete state size, data type, and complexity
getOutputDataTypeImpl	Data types of output ports
getOutputSizeImpl	Sizes of output ports
isOutputComplexImpl	Complexity of output ports
isOutputFixedSizeImpl	Fixed- or variable-size output ports
propagatedInputComplexity	Complexity of input during Simulink propagation

propagatedInputDataType	Data type of input during Simulink propagation
propagatedInputFixedSize	Fixed-size status of input during Simulink propagation
propagatedInputSize	Size of input during Simulink propagation

Note: If your System object has exactly one input and one output and no discrete property states, or if you do not need bus support, you do not have to implement any of these methods. The `matlab.system.mixin.Propagates` provides default values in these cases.

See Also

matlab.System

Tutorials

- "Set Output Data Type"
- "Set Output Size"
- "Set Output Complexity"
- "Specify Whether Output Is Fixed- or Variable-Size"
- "Specify Discrete State Output Specification"

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes

getDiscreteStateSpecificationImpl

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Discrete state size, data type, and complexity

Syntax

```
[sz,dt,cp] = getDiscreteStateSpecificationImpl(obj,name)
```

Description

`[sz,dt,cp] = getDiscreteStateSpecificationImpl(obj,name)` returns the size, data type, and complexity of the property, `name`. This property must be a discrete state property. You must define this method if your System object has discrete state properties and is used in the `MATLAB System` block. If you define this method for a property that is not discrete state, an error occurs during model compilation.

You always set the `getDiscreteStateSpecificationImpl` method access to `protected` because it is an internal method that users do not directly call or run.

`getDiscreteStateSpecificationImpl` is called by the `MATLAB System` block during Simulink model compilation.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

`name`

Name of discrete state property of the System object

Output Arguments

`sz`

Vector containing the length of each dimension of the property.

Default: `[1 1]`

`dt`

Data type of the property. For built-in data types, `dt` is a string. For fixed-point data types, `dt` is a numerictype object.

Default: `double`

`cp`

Complexity of the property as a scalar, logical value, where `true` = complex and `false` = real.

Default: `false`

Examples

Specify Discrete State Property Size, Data Type, and Complexity

Specify in your class definition file the size, data type, and complexity of a discrete state property.

```
methods (Access = protected)
   function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
      sz = [1 1];
      dt = 'double';
      cp = false;
   end
```

```
end
```

See Also

matlab.system.mixin.Propagates

How To

- “Specify Discrete State Output Specification”

getOutputDataTypeImpl

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Data types of output ports

Syntax

```
[dt_1,dt_2,...,dt_n] = getOutputDataTypeImpl(obj)
```

Description

`[dt_1,dt_2,...,dt_n] = getOutputDataTypeImpl(obj)` returns the data types of each output port. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `step` method.

For System objects with one input and one output and where you want the input and output data types to be the same, you do not need to implement this method. In this case `getOutputDataTypeImpl` assumes the input and output data types are the same and returns the data type of the input.

If your System object has more than one input or output or you need the output and input data types to be different, you must implement the `getOutputDataTypeImpl` method to define the output data type. For bus output, you must also specify the name of the output bus. You use `propagatedInputDataType` within the `getOutputDataTypeImpl` method to obtain the input type, if necessary.

During Simulink model compilation and propagation, the `MATLAB System` block calls the `getOutputDataType` method, which then calls the `getOutputDataTypeImpl` method to determine the output data type.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object

Output Arguments

`dt_1,dt_2,...`

Data type of the property. For built-in data types, `dt` is a string. For fixed-point data types, `dt` is a numerictype object.

Examples

Specify Output Data Type

Specify, in your class definition file, the data type of a System object with one output.

```
methods (Access = protected)
   function dt_1 = getOutputDataTypeImpl(~)
      dt_1 = 'double';
   end
end
```

Specify Bus Output

Specify, in your class definition file, that the System object data type is a bus. You must also include a property to specify the bus name.

```
properties(Nontunable)
   OutputBusName = 'myBus';
end

methods (Access = protected)
   function out = getOutputDataTypeImpl(obj)
      out = obj.OutputBusName;
   end
```

```
end
```

See Also

matlab.system.mixin.Propagates | propagatedInputDataType

getOutputSizeImpl

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Sizes of output ports

Syntax

```
[sz_1,sz_2,...,sz_n] = getOutputSizeImpl(obj)
```

Description

`[sz_1,sz_2,...,sz_n] = getOutputSizeImpl(obj)` returns the sizes of each output port. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `step` method.

For System objects with one input and one output and where you want the input and output sizes to be the same, you do not need to implement this method. In this case `getOutputSizeImpl` assumes the input and output sizes are the same and returns the size of the input. For variable-size inputs in MATLAB, the size varies each time you run `step` on your object. For variable-size inputs in Simulink, the output size is the maximum input size.

If your System object has more than one input or output or you need the output and input sizes to be different, you must implement the `getOutputSizeImpl` method to define the output size. You also must use the `propagatedInputSize` method if the output size differs from the input size.

During Simulink model compilation and propagation, the `MATLAB System` block calls the `getOutputSize` method, which then calls the `getOutputSizeImpl` method to determine the output size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`sz_1,sz_2,...`

Vector containing the size of each output port.

Examples

Specify Output Size

Specify in your class definition file the size of a System object output.

```
methods (Access = protected)
   function sz_1 = getOutputSizeImpl(obj)
      sz_1 = [1 1];
   end
end
```

Specify Multiple Output Ports

Specify in your class definition file the sizes of multiple System object outputs.

```
methods (Access = protected)
   function [sz_1,sz_2] = getOutputSizeImpl(obj)
      sz_1 = propagatedInputSize(obj,1);
      sz_2 = [1 1];
   end
 end
```

Specify Output When Using Propagated Input Size

Specify in your class definition file the size of System object output when it is dependant on the propagated input size.

```
methods (Access = protected)
   function varargout = getOutputSizeImpl(obj)
      varargout{1} = propagatedInputSize(obj,1);
      if obj.HasSecondOutput
         varargout{2} = [1 1];
      end
   end
end
```

See Also

matlab.system.mixin.Propagates | propagatedInputSize

How To

- “Set Output Size”

isOutputComplexImpl

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Complexity of output ports

Syntax

```
[cp_1,cp_2,...,cp_n] = isOutputComplexImpl(obj)
```

Description

`[cp_1,cp_2,...,cp_n] = isOutputComplexImpl(obj)` returns whether each output port has complex data. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `step` method.

For System objects with one input and one output and where you want the input and output complexities to be the same, you do not need to implement this method. In this case `isOutputComplexImpl` assumes the input and output complexities are the same and returns the complexity of the input.

If your System object has more than one input or output or you need the output and input complexities to be different, you must implement the `isOutputComplexImpl` method to define the output complexity. You also must use the `propagatedInputComplexity` method if the output complexity differs from the input complexity.

During Simulink model compilation and propagation, the `MATLAB System` block calls the `isOutputComplex` method, which then calls the `isOutputComplexImpl` method to determine the output complexity.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`cp_1,cp_2,...`

Logical, scalar value indicating whether the specific output port is complex (`true`) or real (`false`).

Examples

Specify Output as Real-Valued

Specify in your class definition file that the output from a System object is a real value.

```
methods (Access = protected)
   function c1 = isOutputComplexImpl(obj)
       c1 = false;
   end
end
```

See Also

matlab.system.mixin.Propagates | propagatedInputComplexity

How To

- “Set Output Complexity”

isOutputFixedSizeImpl

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Fixed- or variable-size output ports

Syntax

```
[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj)
```

Description

`[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj)` indicates whether each output port is fixed size. The number of outputs must match the value returned from the `getNumOutputs` method, which is the number of output arguments listed in the `step` method.

For System objects with one input and one output and where you want the input and output fixed sizes to be the same, you do not need to implement this method. In this case `isOutputFixedSizeImpl` assumes the input and output fixed sizes are the same and returns the fixed size of the input.

If your System object has more than one input or output or you need the output and input fixed sizes to be different, you must implement the `isOutputFixedSizeImpl` method to define the output fixed size. You also must use the `propagatedInputFixedSize` method if the output fixed size status differs from the input fixed size status.

During Simulink model compilation and propagation, the `MATLAB System` block calls the `isOutputFixedSize` method, which then calls the `isOutputFixedSizeImpl` method to determine the output fixed size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.

Note: You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`flag_1,flag2,...`

Logical, scalar value indicating whether the specific output port is fixed size (`true`) or variable size (`false`).

Examples

Specify Output as Fixed Size

Specify in your class definition file that the output from a System object is of fixed size.

```
methods (Access = protected)
   function c1 = isOutputFixedSizeImpl(obj)
      c1 = true;
   end
end
```

See Also

matlab.system.mixin.Propagates | propagatedInputFixedSize

How To

- “Specify Whether Output Is Fixed- or Variable-Size”

propagatedInputComplexity

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Complexity of input during Simulink propagation

Syntax

```
flag = propagatedInputComplexity(obj,index)
```

Description

`flag = propagatedInputComplexity(obj,index)` returns `true` or `false` to indicate whether the input argument for the indicated System object is complex. `index` specifies the `step` method input for which to return the complexity flag.

You can use `propagatedInputComplexity` only from within the `isOutputComplexImpl` method in your class definition file. Use `isOutputComplexImpl` when:

- Your System object has more than one input or output.
- The input complexity determines the output complexity.
- The output complexity must differ from the input complexity.

Input Arguments

`obj`

System object

`index`

Index of the specified `step` method input. Do not count the `obj` in the `index`. The first step input is always `obj`.

Output Arguments

`flag`

Complexity of the specified input, returned as `true` or `false`

Examples

Match Input and Output Complexity

Get the complexity of the second input to the `step` method and set the output to match it. Assume that the first input of the `step` method has no impact on the output complexity.

```
methods (Access = protected)
   function outcomplx = isOutputComplexImpl(obj)
      outcomplx = propagatedInputComplexity(obj,2);
   end
end
```

See Also

matlab.system.mixin.Propagates | isOutputComplexImpl

How To

- “Set Output Complexity”

propagatedInputDataType

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Data type of input during Simulink propagation

Syntax

```
dt = propagatedInputDataType(obj,index)
```

Description

`dt = propagatedInputDataType(obj,index)` returns the data type of an input argument for a System object. `index` specifies the `step` method input for which to return the data type.

You can use `propagatedInputDataType` only from within `getOutputDataTypeImpl`. Use `getOutputDataTypeImpl` when:

- Your System object has more than one input or output.
- The input data type status determines the output data type.
- The output data type must differ from the input data type.

Input Arguments

`obj`

System object

`index`

Index of the specified `step` method input. Do not count the `obj` in the `index`. The first step input is always `obj`.

Output Arguments

`dt`

Data type of the specified `step` method input, returned as a string for floating-point input or as a numerictype for fixed-point input.

Examples

Match Input and Output Data Type

Get the data type of the second input to the `step` method. If the second input data type is `double`, then the output data type is `int32`. For all other cases, the output data type matches the second input data type. Assume that the first input to the `step` method has no impact on the output.

```
methods (Access = protected)
   function dt = getOutputDataTypeImpl(obj)
       if strcmpi(propagatedInputDataType(obj,2),'double')
           dt = 'int32';
       else
           dt = propagatedInputDataType(obj,2);
       end
   end
end
```

See Also

“Data Type Propagation” | matlab.system.mixin.Propagates | getOutputDataTypeImpl

How To

- “Set Output Data Type”

propagatedInputFixedSize

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Fixed-size status of input during Simulink propagation

Syntax

```
flag = propagatedInputFixedSize(obj,index)
```

Description

`flag = propagatedInputFixedSize(obj,index)` returns `true` or `false` to indicate whether an input argument of a System object is fixed size. `index` specifies the `step` method input for which to return the fixed-size flag.

You can use `propagatedInputFixedSize` only from within `isOutputFixedSizeImpl`. Use `isOutputFixedSizeImpl` when:

- Your System object has more than one input or output.
- The input fixed-size status determines the output fixed-size status.
- The output fixed-size status must differ from the input fixed-size status.

Input Arguments

`obj`

System object

`index`

Index of the specified `step` method input. Do not count the `obj` in the `index`. The first step input is always `obj`.

Output Arguments

`flag`

Fixed-size status of the specified `step` method input, returned as `true` or `false`.

Examples

Match Fixed-Size Status of Input and Output

Get the fixed-size status of the third input to the `step` method and set the output to match it. Assume that the first and second inputs to the `step` method have no impact on the output.

```
methods (Access = protected)
   function outtype = isOutputFixedSizeImpl(obj)
      outtype = propagatedInputFixedSize(obj,3)
   end
end
```

See Also

matlab.system.mixin.Propagates | isOutputFixedSizeImpl

How To

- “Specify Whether Output Is Fixed- or Variable-Size”

propagatedInputSize

Class: matlab.system.mixin.Propagates
Package: matlab.system.mixin

Size of input during Simulink propagation

Syntax

```
sz = propagatedInputSize(obj,index)
```

Description

`sz = propagatedInputSize(obj,index)` returns, as a vector, the input size of the specified System object. The`index` specifies the input to the `step` method for which to return the size information. (Do not count the `obj` in the `index`. The first input is always `obj`.)

You can use `propagatedInputSize` only from within the `getOutputSizeImpl` method in your class definition file. Use `getOutputSizeImpl` when:

- Your System object has more than one input or output.
- The input size determines the output size.
- The output size must differ from the input size.

Note: For variable-size inputs, the propagated input size from `propagatedInputSizeImpl` differs depending on the environment.

- MATLAB — When you first run step on an object, it uses the actual sizes of the inputs.

- Simulink — The maximum of all the input sizes is set before the model runs and does not change during the run.

Input Arguments

`obj`

System object

`index`

Index of the specified `step` method input

Output Arguments

`sz`

Size of the specified input, returned as a vector

Examples

Match Size of Input and Output

Get the size of the second input to the `step` method. If the first dimension of the second input to the `step` method has a size greater than `1`, then set the output size to a 1 x 2 vector. For all other cases, the output is a 2 x 1 matrix. Assume that the first input to the `step` method has no impact on the output size.

```
methods (Access = protected)
   function outsz = getOutputSizeImpl(obj)
       sz = propagatedInputSize(obj,2);
       if sz(1) == 1
           outsz = [1,2];
       else
           outsz = [2,1];
       end
   end
end
```

See Also

matlab.system.mixin.Propagates | getOutputSizeImpl

How To

- “Set Output Size”

matlab.system.mixin.Nondirect class

Package: matlab.system.mixin

Nondirect feedthrough mixin class

Description

`matlab.system.mixin.Nondirect` is a class that uses the `output` and `update` methods to process nondirect feedthrough data through a System object.

For System objects that use direct feedthrough, the object's input is needed to generate the output at that time. For these direct feedthrough objects, the `step` method calculates the output and updates the state values. For nondirect feedthrough, however, the object's output depends only on the internal states at that time. The inputs are used to update the object states. For these objects, calculating the output with `outputImpl` is separated from updating the state values with `updateImpl`. If you use the `matlab.system.mixin.Nondirect` mixin and include the `stepImpl` method in your class definition file, an error occurs. In this case, you must include the `updateImpl` and `outputImpl` methods instead.

The following cases describe when System objects in Simulink use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the System object code.
- System object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
- System object does not support code generation — Default `isInputDirectFeedthrough` method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

Use the `Nondirect` mixin to allow a System object to be used in a Simulink feedback loop. A delay object is an example of a nondirect feedthrough object.

To use this mixin, you must subclass from this class in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system &...
    matlab.system.mixin.Nondirect
```

Methods

isInputDirectFeedthroughImpl	Direct feedthrough status of input
outputImpl	Output calculation from input or internal state of System object
updateImpl	Update object states based on inputs

See Also

matlab.system

Tutorials

- “Use Update and Output for Nondirect Feedthrough”

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes

isInputDirectFeedthroughImpl

Class: matlab.system.mixin.Nondirect
Package: matlab.system.mixin

Direct feedthrough status of input

Syntax

```
[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,u1,u2,...,uN)
```

Description

`[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,u1,u2,...,uN)` indicates whether each input is a direct feedthrough input. If direct feedthrough is `true`, the output depends on the input at each time instant.

Note: You must set `Access = protected` for this method.

You cannot modify any properties or implement or access tunable properties in this method.

If you do not include the `isInputDirectFeedthroughImpl` method in your System object class definition file, all inputs are assumed to be direct feedthrough.

The following cases describe when System objects in Simulink code generation use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the System object code.
- System object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
- System object does not support code generation — Default `isInputDirectFeedthrough` method returns false, indicating that direct

feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

`isInputDirectFeedthroughImpl` is called by the `isInputDirectFeedthrough` method.

Input Arguments

`obj`

System object handle

`u1,u2,...,uN`

Specifications of the inputs to the algorithm or `step` method.

Output Arguments

`flag1,...,flagN`

Logical value or either `true` or `false`. This value indicates whether the corresponding input is direct feedthrough or not, respectively. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

Examples

Specify Input as Nondirect Feedthrough

Use `isInputDirectFeedthroughImpl` in your class definition file to mark the inputs as nondirect feedthrough.

```
methods (Access = protected)
   function flag = isInputDirectFeedthroughImpl(~,~)
      flag = false;
   end
end
```

See Also

matlab.system.mixin.Nondirect

How To

- "Use Update and Output for Nondirect Feedthrough"

outputImpl

Class: matlab.system.mixin.Nondirect
Package: matlab.system.mixin

Output calculation from input or internal state of System object

Syntax

```
[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)
```

Description

`[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)` implements the output equations for the System object. The output values are calculated from the states and property values. Any inputs that you set to nondirect feedthrough are ignored during output calculation.

`outputImpl` is called by the `output` method. It is also called before the `updateImpl` method in the `step` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Note: You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Input Arguments

`obj`

System object handle

`u1,u2,...uN`

Inputs from the algorithm or `step` method. The number of inputs must match the number of inputs returned by the `getNumInputs` method. Nondirect feedthrough inputs

are ignored during normal execution of the System object. However, for code generation, you must provide these inputs even if they are empty.

Output Arguments

`y1,y2,...yN`

Outputs calculated from the specified algorithm. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

Examples

Set Up Output that Does Not Depend on Input

Specify in your class definition file that the output does not directly depend on the current input with the `outputImpl` method. `PreviousInput` is a property of the `obj`.

```
methods (Access = protected)
   function [y] = outputImpl(obj, ~)
      y = obj.PreviousInput(end);
   end
end
```

See Also

matlab.system.mixin.Nondirect

How To

- "Use Update and Output for Nondirect Feedthrough"

updateImpl

Class: matlab.system.mixin.Nondirect
Package: matlab.system.mixin

Update object states based on inputs

Syntax

```
updateImpl(obj,u1,u2,...,uN)
```

Description

`updateImpl(obj,u1,u2,...,uN)` implements the state update equations for the system. You use this method when your algorithm outputs depend only on the object's internal state and internal properties. Do not use this method to update the outputs from the inputs.

`updateImpl` is called by the `update` method and after the `outputImpl` method in the `step` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Note: You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Input Arguments

`obj`

System object handle

`u1,u2,...uN`

Inputs to the algorithm or `step` method. The number of inputs must match the number of inputs returned by the `getNumInputs` method.

Examples

Set Up Output that Does Not Depend on Current Input

Update the object with previous inputs. Use `updateImpl` in your class definition file. This example saves the `u` input and shifts the previous inputs.

```
methods (Access = protected)
   function updateImpl(obj,u)
      obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
   end
end
```

See Also

matlab.system.mixin.Nondirect

How To

- "Use Update and Output for Nondirect Feedthrough"

ModelAdvisor.Preferences

Set Model Advisor preferences

Description

Use instances of this class to set Model Advisor preferences.

Property Summary

Property	Description	Values
`DeselectByProduct`	String specifying the selection of the **By Product** folder in the Model Advisor window. The default value is `true`.	`{true} | false`
`ShowByProduct`	String specifying the display of the **By Product** folder in the Model Advisor window. The default value is `true`.	`{true} | false`
`ShowByTask`	String specifying the display of the **By Task** folder in the Model Advisor window. The default value is `true`.	`{true} | false`
`ShowSourceTab`	String specifying the display of the **Source** tab in the Model Advisor window. The default value is `false`. When you click the **Source** tab, the Model Advisor window displays the check Title, TitleID, and location of the MATLAB source code for the check.	`true | {false}`
`ShowExclusionTab`	String specifying the display of the **Exclusions** tab in the Model Advisor window. The default	`true | {false}`

Property	Description	Values
	value is `false`. When you click the **Exclusions** tab, the Model Advisor window displays checks that are excluded form the Model Advisor analysis.	
`ShowAccordian`	String specifying the display of the **Code Generation Advisor**, **Upgrade Advisor**, and **Performance Advisor** in the Model Advisor window. The default value is `false`. You can use these advisors to help configure your model for code generation, upgrade your model for the current release, or improve performance.	`true | {false}`
`ShowExclusionsInR`	String specifying to include exclusions in the Model Advisor report. The default value is `true`.	`{true} | false`

Methods

Name	Description
`load`	Load Model Advisor preferences.
`save`	Save Model Advisor preferences.

Examples

This example shows how to not display the **By Product** folder in the Model Advisor window:

```
mp = ModelAdvisor.Preferences;
mp.load;
mp.ShowByProduct = false;
mp.save
```

Alternatives

You can set the Model Advisor preferences by using the Model Advisor Preferences dialog box:

- On the Model Advisor menu, select **Settings > Preferences**.
- From the Model Editor, select **Analysis > Model Advisor > Preferences**.

See Also

"Run Model Checks"

Introduced in R2014b

Simulink.AliasType

Create alias for signal and parameter data type

Description

This class allows you to designate MATLAB variables as aliases for signal and parameter data types. You designate variables as aliases by creating instances of this class and assigning them to variables in the MATLAB base workspace or a data dictionary (see "Creating a Data Type Alias" on page 5-128). The MATLAB variable to which a `Simulink.AliasType` object is assigned is called a data type alias. The data type to which an alias refers is called its base type. Simulink software allows you to set the `BaseType` property of the object that the variable references, thereby designating the data type for which it is an alias.

Simulink software lets you use aliases instead of actual type names in dialog boxes and `set_param` commands to specify the data types of Simulink block outputs and parameters. Using aliases to specify signal and parameter data types can greatly simplify global changes to the signal and parameter data types that a model specifies. In particular, changing the data type of all signals and parameters whose data type is specified by an alias requires changing only the base type of the alias. By contrast, changing the data types of signals and parameters whose data types are specified by an actual type name requires respecifying the data type of each signal and parameter individually.

You can use objects of this class to create an alias for Simulink built-in data types, enumerated data types, `Simulink.NumericType` objects, and other `Simulink.AliasType` objects.

To define and name your own fixed-point data type, use an object of the class `Simulink.NumericType`. Objects of the class `Simulink.AliasType` can create an alias for a `Simulink.NumericType` object that defines a fixed-point data type, but cannot define a fixed-point data type.

Creating a Data Type Alias

You can use either the Model Explorer or MATLAB commands to create a data type alias. See "MATLAB Commands for Creating Data Type Aliases" on page 5-129.

To use the Model Explorer to create an alias:

1. On the Model Explorer **Model Hierarchy** pane, select `Base Workspace`.

   You must create data type aliases in the MATLAB workspace or in a data dictionary. If you attempt to create an alias in a model workspace, Simulink software displays an error.
2. From the Model Explorer **Add** menu, select **Simulink.AliasType**.

   Simulink software creates an instance of a `Simulink.AliasType` object and assigns it to a variable named `Alias` in the MATLAB workspace.
3. Rename the variable to a more appropriate name, for example, a name that reflects its intended usage.

   To change the name, edit the name displayed in the **Name** field on the Model Explorer **Contents** pane.
4. On the Model Explorer **Dialog** pane, in the **Base type** field, enter the name of the data type that this alias represents.

   You can specify the name of any existing standard or user-defined data type in this field. Skip this step if the base type is `double` (the default).
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 command prompt or in a MATLAB script.

```
ALIAS = Simulink.AliasType;
```

`ALIAS` is the name of the variable that you want to serve as the alias. For example, the following line creates an alias named `MyFloat`.

```
MyFloat = Simulink.AliasType;
```

The following notations get and set the properties of a data type alias, respectively:

```
PROPVALUE = ALIAS.PROPNAME;
ALIAS.PROPNAME = PROPVALUE;
```

`ALIAS` is the name of the alias object, `PROPNAME` is the name of the alias object properties, and `PROPVALUE` is the property value. For example, the following code saves the current value of `MyFloat`'s `BaseType` property and assigns it a new value.

```
old = MyFloat.BaseType;
MyFloat.BaseType = 'single';
```

For information on the names, permitted values, and usage of the properties of data type alias objects, see “Properties” on page 5-134.

Create Alias for Enumerated Data Type

You can use a `Simulink.AliasType` object to create an alias for an enumerated data type. For example, you can create an alias for an enumerated type called `SlDemoSign`.

```
myEnumAlias = Simulink.AliasType;
myEnumAlias.BaseType = 'Enum: SlDemoSign';
```

You can use the expression `myEnumAlias` to specify the data type of signals or parameters as the enumerated type `SlDemoSign`.

Data Type Aliases in Generated Code

If you have a Simulink Coder license, you can cause data type aliases to appear in the code generated for a model using any of the following methods:

- Specifying the signal data type of a block in the model as a `Simulink.AliasType` via the **Block Parameters** dialog box.
- Creating a `Simulink.Signal` object that uses the `Simulink.AliasType` as its data type. Use this signal object as the name of a signal in the model and specify that the signal name must resolve to an object in the MATLAB workspace. See “Control Signals and States in Code by Applying Storage Classes” in the Simulink Coder User's Guide.
- Creating a `Simulink.Parameter` object that uses the `Simulink.AliasType` as its data type. Use this parameter object as a block parameter in the model. See “Control Parameter Data Types in the Generated Code” in the Simulink Coder User's Guide.

Notes

- If you assign a data type in a block dialog box and if you use a `Simulink.Signal` object on the signal feeding into the block, the code is always generated using the data type in the dialog box.

- The Simulink Coder code generator tries to preserve the names of alias types in the generated code. However, in some cases, an alias type name can revert to its underlying equivalent built-in data type. If you have a Embedded Coder license, you can specify that the code generator use 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.
- You can specify the data type of a complex signal using the `Simulink.AliasType` class. In this case, if the **DataScope** property of the `Simulink.AliasType` class is set to `Imported` (or `Auto` with a header file specified), provide a definition for the complex type. As shown in the following example, the alias type definition, `IAT_int32`, must contain the name of the complex type prefixed by `c`.

```
#ifndef myAliasTypes_H_
#define myAliasTypes_H_

#include "rtwtypes.h"

typedef int32_T IAT_int32;
typedef cint32_T cIAT_int32;

#endif
```

  In the preceding example, while you must define `IAT_int32` in the base workspace, you do not need to define `cIAT_int32` in the base workspace.
- If you define two nested alias types, Simulink Coder generates an error if the **DataScope** property of the alias type is set to `Imported` or if either of the alias types specifies a header file.

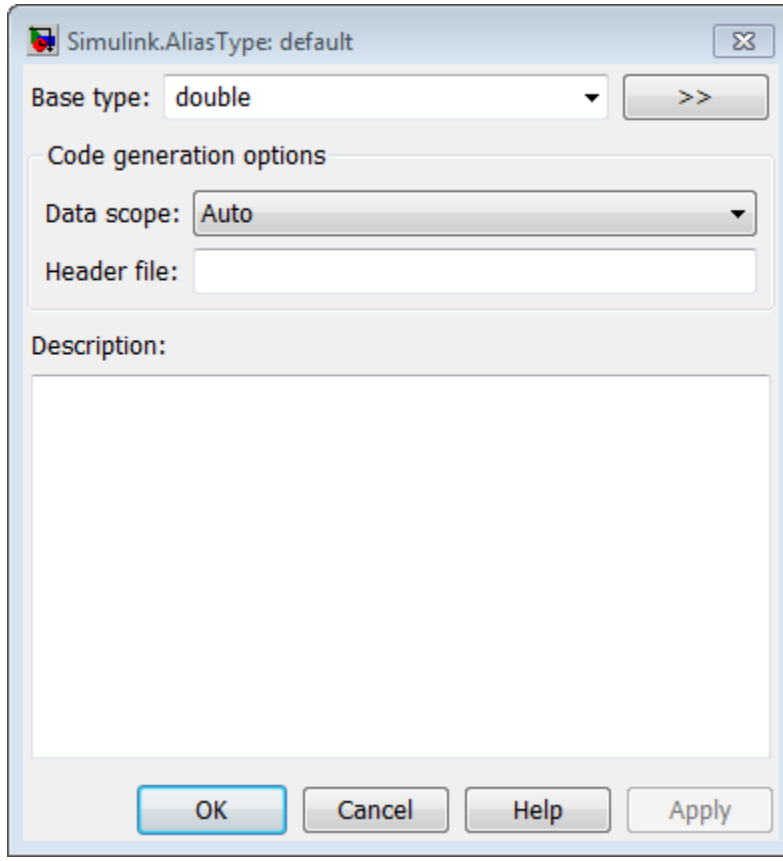
Parent

None

Children

None

Property Dialog Box



Base type

The data type to which this alias refers. The default is `double`. To specify another data type, select the data type from the adjacent drop–down list of standard data types or enter the data type name in the edit field. You can, with one exception, specify a nonstandard data type, e.g., a data type defined by a `Simulink.NumericType` object, by entering the data type name in the edit field. The exception is a `Simulink.NumericType` whose `Category` is `Fixed-point: unspecified scaling`.

Note: `Fixed-point: unspecified scaling` is a partially specified type whose definition is completed by the block that uses the `Simulink.NumericType`. Forbidding its use in alias types avoids creating aliases that have different base types depending on where they are used.

Data scope

Specifies whether the data type definition is imported from, or exported to, a header file during code generation. The possible values are:

Value	Action
`Auto` (default)	If no value is specified for **Header file**, export the type definition to *`model`*`_types.h`, where *`model`* is the model name. If you have an Embedded Coder license, and you have specified a data type replacement, then export the type definition to `rtwtypes.h`. If a value is specified for **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`. *`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`. *`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`.

By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify **Header file** as `<myTypes.h>`.

Description

Describes the usage of the data type referenced by this alias.

Properties

Name	Description
`BaseType`	A string specifying the name of a standard or custom data type. (**Base type**)
`DataScope`	A string specifying whether the data type definition is imported from, or exported to, a header file during code generation. (**Data scope**)
`Description`	A string that describes the usage of the data type. Can 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**)

More About

- “About Data Types in Simulink”

See Also

`Simulink.NumericType`

Related Examples

- “Control Signal Data Types”
- “Create Data Type Alias in the Generated Code”
- “Data Type Replacement”

Introduced before R2006a

Simulink.Annotation

Specify properties of model annotation

Description

Instances of this class specify the properties of annotations. You can use `getCallbackAnnotation` in an annotation callback function to get the `Simulink.Annotation` instance for the annotation associated with the callback function. You can use `find_system` and `get_param` to get the `Simulink.Annotation` instance associated with any annotation in a model. For example, 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 font. The default value, `'auto'`, specifies use of the model preference for the font angle.	`'normal' | 'italic' | 'oblique' | {'auto'}`
`FontName`	String specifying name of annotation's font. The default value, `'auto'`, specifies use of the model preference for the font.	string
`FontSize`	Integer specifying size of annotation's font in points. The default value, `-1`, specifies use of the model preference for the font size.	real `{'-1'}`
`FontWeight`	String specifying the weight of the annotation font. The default value, `'auto'`, specifies use of the model preference for 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
`Position`	Array specifying the location of the annotation	1x4 array `[left top right bottom]`. The maximum value for a coordinate is 32767.

Property	Description	Values
`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 (for example, `'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. Annotation background color can also be `'black'`, `'white'`, `'red'`, `'green'`, `'blue'`, `'cyan'`, `'magenta'`, `'yellow'`, `'gray'`, `'lightBlue'`, `'orange'`, `'darkGreen'`.
`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. Annotation 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'}`

Property	Description	Values
`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 MATLAB code to be executed when the model containing this annotation is loaded. See “Annotation Callback Functions”.	string
`DeleteFcn`	String specifying MATLAB code to be executed before deleting this annotation. See “Annotation Callback Functions”.	string
`RequirementInfo`	For internal use.	string
`Tag`	User-specified text that is assigned to the annotation `Tag` parameter and saved with the annotation.	string

Property	Description	Values
`UseDisplayText-AsClickCallback`	String specifying whether to use the contents of the `Text` property as the click function for this annotation. 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 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" for more information.	`'on' | {'off'}`
`UserData`	Any data that you want to associate with this annotation.	vector
`Interpreter`	Type of annotation	`'rich' | 'tex' | {'off'}`
`IsImage`	Only `'on'` if the annotation is an image annotation	`'on' | {'off'}`
`InternalMargins`	Array specifying the space from the bounding box of text to the borders of the annotation.	1x4 array `[left top right bottom]`. The default is `[1 1 1 1]`. The maximum value for a coordinate is 32767.
`PlainText`	Read-only display of the text in the annotation, without formatting	vector
`FixedHeight`	String specifying whether the bottom border of the annotation resizes as you add content	`'on' | {'off'}`, where `'off'` means that the bottom border resizes as you add content

Property	Description	Values
`FixedWidth`	String specifying whether or not to use wordwrap or to have the width of the annotation expand to accommodate text	`'on' | {'off'}`, where `'off'` means to use wordwrap

Method Summary

Method	Description
`delete`	Delete this annotation from the Simulink model.
`dialog`	Display the Annotation properties dialog box.
`disp`	Display the property names and their settings for this Annotation object.
`fitToView`	Zoom in on this annotation and highlight it in the model.
`get`	Return the specified property settings for this annotation.
`help`	Display a list of properties for this Annotation object with short descriptions.
`insertImage`	Insert image from clipboard or image file into an annotation.
`methods`	Display all nonglobal methods of this Annotation object.
`set`	Set the specified property of this Annotation object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Annotation object.
`view`	Display this annotation in the Simulink Editor with this annotation highlighted.

Introduced before R2006a

Simulink.BlockCompDworkData

Provide postcompilation information about block's DWork vector

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the "Dwork" on page 5-314 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 5-141	Usage type of this DWork vector.
"UsedAsDiscState" on page 5-142	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.

Introduced before R2006a

Simulink.BlockCompInputPortData

Provide postcompilation information about block input port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the "InputPort" on page 5-315 method of a block's run-time object after the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

None

Property Summary

Name	Description
"DirectFeedthrough" on page 5-143	True if this port has direct feedthrough.
"Overwritable" on page 5-144	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.

Introduced before R2006a

Simulink.BlockCompOutputPortData

Provide postcompilation information about block output port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the "OutputPort" on page 5-316 method of a block's run-time object after the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

None

Property Summary

Name	Description
"Reusable" on page 5-169	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.

Introduced before R2006a

Simulink.BlockData

Provide run-time information about block-related data, such as block parameters

Description

This class defines properties that are common to objects that provide run-time information about a block's ports and work vectors.

Parent

None

Children

`Simulink.BlockPortData`, `Simulink.BlockCompDworkData`

Property Summary

Name	Description
"AliasedThroughDataType" on page 5-148	Fundamental base data type.
"AliasedThroughDataTypeID" on page 5-149	Fundamental base data type ID.
"Complexity" on page 5-149	Numeric type (real or complex) of the block data.
"Data" on page 5-149	The block data.
"DataAsDouble" on page 5-150	The block data in `double` form.
"Datatype" on page 5-150	Data type of the block data.
"DatatypeID" on page 5-150	Index of the data type of the block data.

Name	Description
"Dimensions" on page 5-151	Dimensions of the block data.
"Name" on page 5-151	Name of the block data.
"Type" on page 5-152	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" on page 5-150 or "DatatypeID" on page 5-150 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`

Data Type	Value
`'double'`	0
`'single'`	1
`'int8'`	2
`'uint8'`	3
`'int16'`	4
`'uint16'`	5
`'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

Introduced before R2006a

Simulink.BlockPath

Fully specified Simulink block path

Description

A `Simulink.BlockPath` object represents a fully specified block path that uniquely identifies a block within a model hierarchy, including model reference hierarchies that involve multiple instances of a referenced model. Simulink uses block path objects in a variety of contexts. For example, when you specify Normal mode visibility, Simulink uses block path objects to identify the models with Normal mode visibility. For details, see "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.

Name	Description
`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)
```

More About

- “Specify the Instance That Has Normal Mode Visibility”

See Also

`Simulink.SimulationData.BlockPath` | Simulink.SimulationData.Dataset

Simulink.BlockPortData

Describe block input or output port

Description

This class defines properties that are common to objects that provide run-time information about a block's ports.

Parent

`Simulink.BlockData`

Children

`Simulink.BlockPreCompInputPortData`, `Simulink.BlockPreCompOutputPortData`, `Simulink.BlockCompInputPortData`, `Simulink.BlockCompOutputPortData`

Property Summary

Name	Description
"IsBus" on page 5-164	True if this port is connected to a bus.
"IsSampleHit" on page 5-164	True if this port produces output or accepts input at the current simulation time step.
"SampleTime" on page 5-164	Sample time of this port.
"SampleTimeIndex" on page 5-165	Sample time index of this port.
"SamplingMode" on page 5-165	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

Introduced before R2006a

Simulink.BlockPreCompInputPortData

Provide precompilation information about block input port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the "InputPort" on page 5-315 method of a block's run-time object before the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

None

Property Summary

Name	Description
"DirectFeedthrough" on page 5-167	True if this port has direct feedthrough.
"Overwritable" on page 5-168	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

Introduced before R2006a

Simulink.BlockPreCompOutputPortData

Provide precompilation information about block output port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the "OutputPort" on page 5-316 method of a block's run-time object before the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

none

Property Summary

Name	Description
"Reusable" on page 5-169	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

Introduced before R2006a

Simulink.Bus

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`
- `From File`
- `From Workspace`
- `Inport`
- `Outport`
- `Signal Specification`

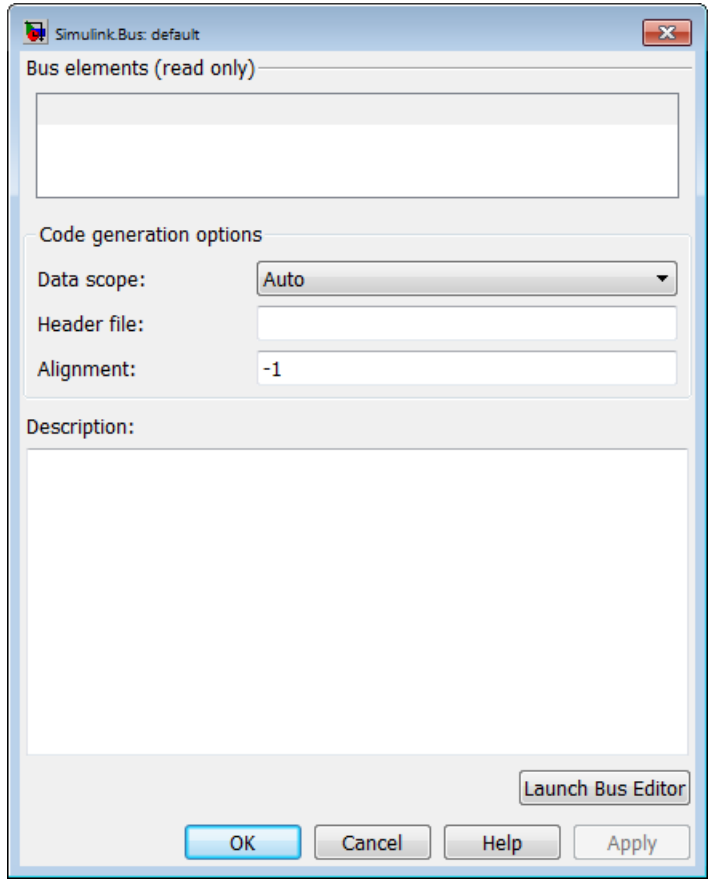
You can use the Model Explorer **Add > Simulink Bus** command (see "Create Data Objects from Built-In Data Class Package Simulink"), the Simulink Bus editor (see "Manage Bus Objects with the Bus Editor"), or MATLAB commands (see "Data Objects") to create bus objects in the base MATLAB workspace. You 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



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. Simulink software ignores this field.

By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify **Header file** as `<myTypes.h>`.

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 “Data Alignment for Code Replacement”. 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)
`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. (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. (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`

Introduced before R2006a

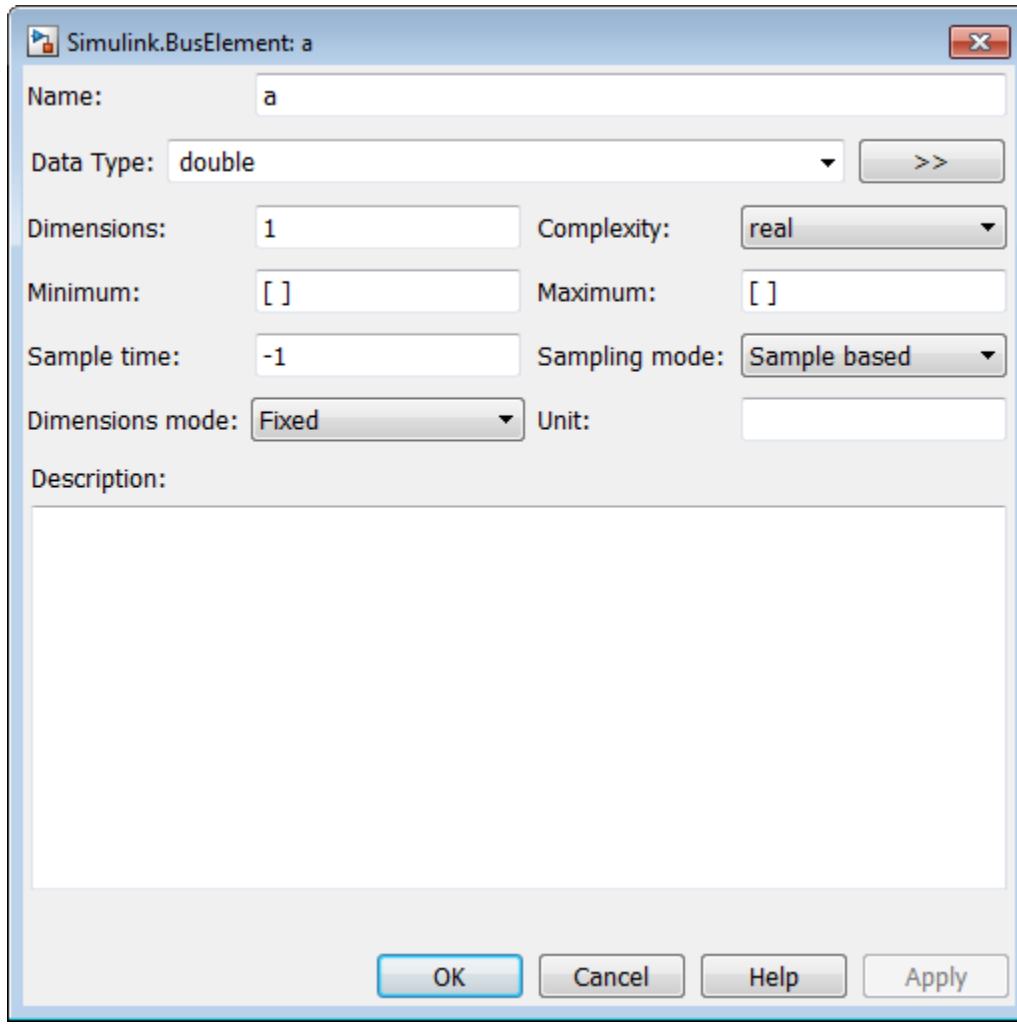
Simulink.BusElement

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



Name

Name of this element. See “Signal Names and Labels” 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

Unit

Physical unit in which this value is expressed (for example, inches). To specify a unit, begin typing in the text box. As you type, the parameter displays potential unit string matches. For more information, see "Unit Specification in Simulink Models".

`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`

Simulink.CoderInfo

Specify information needed to generate code for signal or parameter

Description

Simulink software creates an instance of this class for each instance of a `Simulink.Signal` object or `Simulink.Parameter` object that it creates. Simulink software uses the `Simulink.CoderInfo` object to store information needed to generate code for the signal or parameter specified by the `Simulink.Signal` object or `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.Signal` object or `Simulink.Parameter` object that uses it. For example, the following MATLAB expression sets the `StorageClass` property of a `Simulink.CoderInfo` object used by a signal object named `mysignal`.

```
mysignal.CoderInfo.StorageClass = 'ExportedGlobal';
```

Property Dialog Box

Use the Code Generation Options section of the `Simulink.Signal` or `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 signal or parameter.
`Alignment`	Data alignment boundary for this signal or parameter. See "Data Alignment for Code Replacement" in the Embedded Coder documentation for more information.
`CustomAttributes`	Custom storage class attributes of this signal or parameter. You must set the property `StorageClass` to `'Custom'` to enable this property. See "Introduction

Name	Description
	to Custom Storage Classes" in the Embedded Coder documentation for more information.
`CustomStorageClass`	Custom storage class of this signal or parameter. You must set the property `StorageClass` to `'Custom'` to enable this property.
`StorageClass`	Storage class of this signal or parameter. For more information, see "Control Signals and States in Code by Applying Storage Classes" or "Control Parameter Representation and Declare Tunable Parameters in the Generated Code" in the Simulink Coder documentation.

More About

- "Data Objects"
- "Introduction to Custom Storage Classes"

See Also

`Simulink.Parameter` | `Simulink.Signal`

Related Examples

- "Control Signals and States in Code by Applying Storage Classes"
- "Control Parameter Representation and Declare Tunable Parameters in the Generated Code"

Introduced in R2015a

Simulink.ConfigSet

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” on page 5-185	Components of the configuration set.
“Description” on page 5-186	Description of the configuration set.
“Name” on page 5-186	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” on page 5-186	Attach a component to a configuration set.
“copy” on page 5-187	Create a copy of a configuration set.
“getComponent” on page 5-188	Get a component of a configuration set.
“getFullName” on page 5-188	Get the full path of a configuration set.

Name	Description
"getModel" on page 5-189	Get the handle of the model that owns a configuration set.
"get_param" on page 5-189	Get the value of a configuration set parameter.
"isActive" on page 5-190	Determine whether a configuration set is the active set of the model that owns it.
"isValidParam" on page 5-190	Determine whether a specified parameter is a valid parameter of a configuration set.
"saveAs" on page 5-191	Save a configuration set to a MATLAB file.
"setPropEnabled" on page 5-193	Prevent or allow a user to change a parameter.
"set_param" on page 5-194	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. • `'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 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. The file groups parameters by category. 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"

Introduced before R2006a

Simulink.ConfigSetRef

Link model to configuration set stored independently of any model

Description

Instances of this handle class allow a model to reference configuration sets that exist outside any model. See “Manage a Configuration Set”, “Overview”, and “Manage a Configuration Reference” for more information.

Property Summary

Name	Description
“Description” on page 5-197	Description of the configuration reference.
“Name” on page 5-197	Name of the configuration reference.
“SourceName” on page 5-198	Name of the variable in the workspace or the data dictionary that contains the referenced configuration set.

Note: You can use the **Configuration Reference** dialog box to set the `Name`, `Description`, and `SourceName` properties of a configuration reference. See “Create and Attach a Configuration Reference” for details.

Method Summary

Name	Description
“copy” on page 5-198	Create a copy of a configuration reference.
“getFullName” on page 5-199	Get the full pathname of a configuration reference.
“getModel” on page 5-199	Get the handle of the model that owns a configuration reference.

Name	Description
"get_param" on page 5-199	Get the value of a configuration set parameter indirectly through a configuration reference.
"getRefConfigSet" on page 5-200	Get the configuration set specified by a configuration reference.
"isActive" on page 5-201	Determine whether a configuration reference is the active configuration object of the model.
"refresh" on page 5-201	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

SourceName

Description

Name of the variable in the workspace or the data dictionary 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 `SourceName` to retrieve the configuration set, then retrieves the value of *`paramName`* from that configuration set. Specifying *`paramName`* as `'ObjectParameters'` returns the names of all valid parameters in the configuration set. If a valid configuration set is not attached to the configuration reference, the method returns unreliable values.

The inverse method, `set_param`, is not defined for configuration references. To obtain a parameter value through a configuration reference, you must first use the `getRefConfigSet` method to retrieve the configuration set from the reference, then use `set_param` directly on the configuration set itself.

You can also use the `get_param` model construction command to get the values of parameters of a model's active configuration set, e.g., `get_param(bdroot, 'SolverName')` gets the solver name of the currently selected model.

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 `SourceName` 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 `SourceName`. If you omit executing `refresh` after any such change, the configuration reference handle will be stale, and using it will give incorrect results.

Introduced in R2007a

Simulink.GlobalDataTransfer class

Package: Simulink

Configure concurrent execution data transfers

Description

The `Simulink.GlobalDataTransfer` object contains the data transfer information for the concurrent execution of a model. To access the properties of this class, use the `get_param` function to get the handle for this class, and then use dot notation to access the properties. For example:

```
dt=get_param(gcs,'DataTransfer');
dt.DefaultTransitionBetweenContTasks

ans =

Ensure deterministic transfer (minimum delay)
```

Properties

`DefaultTransitionBetweenSyncTasks`

Global setting for data transfer handling option when the source and destination of a signal are in two different and periodic tasks.

Data Type: Enumeration. Can be one of:

- `'Ensure data integrity only'`
- `'Ensure deterministic transfer (maximum delay)'`
- `'Ensure deterministic transfer (minimum delay)'`

Access: Read/write

`DefaultTransitionBetweenContTasks`

Global setting for the data transfer handling option for signals that have a continuous sample time.

Data Type: Enumeration. Can be one of:

- `'Ensure data integrity only'`
- `'Ensure deterministic transfer (maximum delay)'`
- `'Ensure deterministic transfer (minimum delay)'`

Access: Read/write

`DefaultExtrapolationMethodBetweenContTasks`

Global setting for the data transfer extrapolation method for signals that have a continuous sample time.

Data Type: Enumeration. Can be one of:

- `'None'`
- `'Zero Order Hold'`
- `'Linear'`
- `'Quadratic'`

Access: Read/write

`AutoInsertRateTranBlk`

Setting for whether or not Simulink software automatically inserts hidden `Rate Transition` blocks between blocks that have different sample rates to ensure the integrity of data transfers between tasks; and optional determinism of data transfers for periodic tasks.

Data Type: Boolean. Can be one of:

- `0`
- `1`

Access: Read/write

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Access the properties of this class.

```
dt=get_param(gcs,'DataTransfer');
dt.DefaultTransitionBetweenContTasks

ans =

Ensure deterministic transfer (minimum delay)
```

See Also

`Simulink.architecture.get_param` | `Simulink.architecture.add` | `Simulink.architecture.delete` | `Simulink.architecture.find_system` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.register` | `Simulink.architecture.set_param`

How To

- "Configure Data Transfer Settings Between Concurrent Tasks"

Simulink.HMI.InstrumentedSignals class

Package: Simulink.HMI

Access streamed signals in model

Description

Instances of `Simulink.HMI.InstrumentedSignals` contain a list of streamed signals in a single model from all subsystems, library instances, and Stateflow charts. This list does not include signals from reference models — each reference model includes its own list of streamed signals.

The `Simulink.HMI.InstrumentedSignals` object contains `Simulink.HMI.SignalSpecification` objects within it that specify the block path and port index of a single signal.

Construction

`Obj = get_param(model,'InstrumentedSignals')` returns a `Simulink.HMI.InstrumentedSignals` object with the streamed signal list from the model.

Properties

`Count` — Number of streamed signals

integer

Number of signals in the model that are marked for steaming, returned as an integer.

`Model` — Model name

string

Name of the model with streamed signals, returned as a string.

Example: `'sldemo_fuelsys'`

Examples

Find Block Path of Streamed Signal

```
% Open the model and stream logged signals to Simulation Data Inspector
load_system('sldemo_absbrake');
Simulink.sdi.changeLoggedToStreamed('sldemo_absbrake');

% Simulate the model
sim('sldemo_absbrake');

% Get the streamed signals
sigs = get_param('sldemo_absbrake','InstrumentedSignals');

% Get the block path of the streamed signal
sig1 = get(sigs,1);
blkpath = sig1.BlockPath
```

```
blkpath =

  Simulink.BlockPath
  Package: Simulink

  Block Path:
    sldemo_absbrake/Bus Creator

  Use the getBlock method to access block path strings from this object.
```

Save, Remove, and Restore Streamed Signals

You can save the set of streamed signals to a file and then restore them to your model at a later time.

```
% Save signals that are currently streamed in a model
sigs = get_param(bdroot,'InstrumentedSignals');
save my_instrumented_signals.mat sigs

% Remove all streamed signals from the model
set_param(bdroot,'InstrumentedSignals',[]);

% Restore streamed signals to the model
load my_instrumented_signals.mat
set_param(bdroot,'InstrumentedSignals',sigs);
```

- “Stream Data to the Simulation Data Inspector”

See Also

`Simulink.sdi.changeLoggedToStreamed` | `Simulink.sdi.markSignalForStreaming`

Introduced in R2015b

Simulink.HMI.SignalSpecification class

Package: Simulink.HMI

Information required to stream single signal

Description

Instances of `Simulink.HMI.SignalSpecification` contain information required to stream a single signal in a model.

Construction

`Obj = get_param(model,'InstrumentedSignals')` returns a `Simulink.HMI.InstrumentedSignals` object with the streamed signal data from the model. Use the `get` method to retrieve the signal specification.

```
% Get the streamed signals
sigs = get_param(bdroot,'InstrumentedSignals');

% Get the signal specification for the first signal
sig1 = get(sigs,1);
```

Properties

`BlockPath` — Signal block path

`Simulink.BlockPath` object

Source path of the signal, specified as a `Simulink.BlockPath` object.

`OutportPortIndex` — Signal port index

integer

Source port index of the signal, specified as an integer. For Stateflow, the default is `1`.

Examples

Find Block Path of Streamed Signal

```
% Open the model and stream logged signals to Simulation Data Inspector
load_system('sldemo_absbrake');
Simulink.sdi.changeLoggedToStreamed('sldemo_absbrake');

% Simulate the model
sim('sldemo_absbrake');

% Get the streamed signals
sigs = get_param('sldemo_absbrake','InstrumentedSignals');

% Get the block path of the streamed signal
sig1 = get(sigs,1);
blkpath = sig1.BlockPath
```

```
blkpath =

  Simulink.BlockPath
  Package: Simulink

  Block Path:
    sldemo_absbrake/Bus Creator

  Use the getBlock method to access block path strings from this object.
```

- “Stream Data to the Simulation Data Inspector”

See Also

`Simulink.sdi.changeLoggedToStreamed` | `Simulink.sdi.markSignalForStreaming`

Introduced in R2015b

Simulink.MDLInfo class

Package: Simulink

Extract model file information without loading block diagram into memory

Description

The class Simulink.MDLInfo extracts information from a model file without loading the block diagram into memory.

You can create an `MdlInfo` object containing all the model information properties, or you can use the static methods for convenient access to individual properties without creating the class first. For example, to get the description only:

```
description = Simulink.MDLInfo.getDescription('mymodel')
```

To get the metadata only:

```
metadata = Simulink.MDLInfo.getMetadata('mymodel')
```

All model information properties are read only.

Construction

`info = Simulink.MDLInfo('`*`mymodel`*`')` creates an instance of the `MdlInfo` class `info` and populates the properties with the information from the model file '*`mymodel`*'.

`mymodel` can be:

- A block diagram name (for example, `vdp`)
- The file name for a file on the MATLAB path (for example, `mymodel.slx`)
- A file name relative to the current folder (for example, `mydir/mymodel.slx`)
- A fully qualified file name (for example, `C:\mydir\mymodel.slx`)

`Simulink.MDLInfo` resolves the supplied name by looking at files on the MATLAB path, and ignores any block diagrams in memory. This may cause unexpected results if you supply the name of a loaded model, but its file is shadowed by another file on the

MATLAB path. If a file is shadowed, you see a warning in the command window. To avoid any confusion, supply a fully-qualified file name to `Simulink.MDLInfo`.

Properties

`BlockDiagramName`

Name of block diagram.

`Description`

Description of model.

`FileName`

Name of model file.

`Interface`

Names and attributes of the block diagram's root inports, outports, model references, etc., describing the graphical interface if you created a Model Reference block from this model.

Structure.

`IsLibrary`

Whether the block diagram is a library.

`LastSavedArchitecture`

Platform architecture when saved, for example, `'glnxa64'`.

`Metadata`

Names and attributes of arbitrary data associated with the model.

Structure. The structure fields can be 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.

`ReleaseName`

Name of release, for example, `'R2016a'`.

`SavedCharacterEncoding`

Character encoding when saved, for example, `'UTF-8'`.

`SimulinkVersion`

Version number of Simulink software that was used to save the model file.

Methods

getDescription	Extract model file description without loading block diagram into memory
getMetadata	Extract model file metadata without loading block diagram into memory

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Construct and view a model information object:

```
info = Simulink.MDLInfo('mymodel')
% Get the Version when the model was saved
simulink_version = info.SimulinkVersion;
% Get model metadata
metadata = info.metadata
```

To add metadata to a model, create a metadata structure containing the information you require and use `set_param` to attach it to the model. For example:

```
  metadata.TestStatus = 'untested';
```

```
  metadata.ExpectedCompletionDate
     = '01/01/2011';
  load_system(mymodelname);
  set_param(mymodelname,'Metadata',...
  metadata)  % must be a struct
  save_system(mymodelname);
  close_system(mymodelname);
```

Construct a model information object for a model named `mpowertrain`, in order to find the names of referenced models without loading the model into memory:

```
info = Simulink.MDLInfo('mpowertrain')
% Get the Interface property
info.Interface
```

Output:

```
ans =
                    Inports: [0x1 struct]
                   Outports: [0x1 struct]
                  Trigports: [0x1 struct]
                  Connports: [0x1 struct]
               ModelVersion: '1.122'
           ModelReferences: {2x1 cell}
    ParameterArgumentNames: ''
        TestPointedSignals: [0x1 struct]
```

Get the referenced models:

```
 info.Interface.ModelReferences
```

Output is in the form *`model name / block path | referenced model name`*:

```
ans =
    'mpowertrain/Model Variants|manual_transmission'
    'mpowertrain/engine model|menginemodel'
```

See Also

`Simulink.MDLInfo.getDescription`; `Simulink.MDLInfo.getMetadata`

Simulink.MDLInfo.getDescription

Class: Simulink.MDLInfo
Package: Simulink

Extract model file description without loading block diagram into memory

Syntax

```
description = Simulink.MDLInfo.getDescription('mymodel')
description = info.getDescription
```

Description

`description = Simulink.MDLInfo.getDescription('mymodel')` returns the `description` associated with the file *`mymodel`*, without loading the model.

`mymodel` can be:

- A block diagram name (for example, `vdp`)
- The file name for a file on the MATLAB path (for example, `mymodel.slx`)
- A file name relative to the current folder (for example, `mydir/mymodel.slx`)
- A fully qualified file name (for example, `C:\mydir\mymodel.slx`)

`description = info.getDescription` returns the `description` property of the `Simulink.MDLInfo` object `info`.

Examples

Get the description without loading the model or creating a `Simulink.MDLInfo` object:

```
description = Simulink.MDLInfo.getDescription('mymodel')
```

Create a `Simulink.MDLInfo` object containing all the model information properties, and get the description property:

```
info = Simulink.MDLInfo('mymodel')
description = info.getDescription
```

See Also

`Simulink.MDLInfo`; `Simulink.MDLInfo.getMetadata`

Simulink.MDLInfo.getMetadata

Class: Simulink.MDLInfo
Package: Simulink

Extract model file metadata without loading block diagram into memory

Syntax

```
metadata = Simulink.MDLInfo.getMetadata('mymodel')
metadata = info.getMetadata
```

Description

`metadata = Simulink.MDLInfo.getMetadata('mymodel')` extracts the structure `metadata` associated with the file *`mymodel`*, without loading the model.

`mymodel` can be:

- A block diagram name (for example, `vdp`)
- The file name for a file on the MATLAB path (for example, `mymodel.slx`)
- A file name relative to the current folder (for example, `mydir/mymodel.slx`)
- A fully qualified file name (for example, `C:\mydir\mymodel.slx`)

`metadata = info.getMetadata` returns the `metadata` property of the `Simulink.MDLInfo` object `info`.

`metadata` is a structure containing the names and attributes of arbitrary data associated with the model. The structure fields can be 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`

Simulink.ModelAdvisor

Run Model Advisor from MATLAB file

Description

Use instances of this class in MATLAB programs to run the Model Advisor, for example, to perform a standard set of checks. MATLAB software creates an instance of this object for each model that you open in the current MATLAB session. To get a handle to a model's Model Advisor object, execute the following command

```
ma = Simulink.ModelAdvisor.getModelAdvisor(model);
```

where `model` is the name of the model or subsystem that you want to check. Your program can then use the Model Advisor object's methods to initialize and run the Model Advisor's checks.

About IDs

Many `Simulink.ModelAdvisor` object methods require or return IDs. An *ID* is a unique string identifier for a Model Advisor check, task, or group. ID must remain constant. A `Simulink.ModelAdvisor` object includes methods that enable you to retrieve the ID or IDs for all checks, tasks, and groups, checks belonging to groups and tasks, the active check, and selected checks, tasks and groups.

You find check IDs in the Model Advisor, using check context menus.

To Find	Do This
Check Title, ID, or location of the MATLAB source code	1 On the model window toolbar, select **Settings** > **Preferences**. 2 In the Model Advisor Preferences dialog box, select **Show Source Tab**. 3 In the right pane of the Model Advisor window, click the **Source** tab. The Model Advisor window displays the check Title, TitleId, and location of the MATLAB source code for the check.
Check ID	1 In the left pane of the Model Advisor, select the check. 2 Right-click the check name and select **Send Check ID to Workspace**. The ID is displayed in the Command Window and sent to the base workspace.

To Find	Do This
Check IDs for selected checks in a folder	1 In the left pane of the Model Advisor, select the checks for which you want IDs. Clear the other checks in the folder. 2 Right-click the folder and select **Send Check ID to Workspace**. An array of the selected check IDs are sent to the base workspace.

Syntax

```
ma = Simulink.ModelAdvisor
```

Arguments

ma

A variable representing the `Simulink.ModelAdvisor` object you create.

Properties

`EmitInputParametersToReport`

The `EmitInputParametersToReport` property specifies the display of check input parameters in the Model Advisor report.

Value	Description
`'true'` (default)	Display check input parameters in the Model Advisor report.
`'false'`	Do not display check input parameters in the Model Advisor report.

Method Summary

Name	Description
“closeReport” on page 5-223	Close Model Advisor report.
“deselectCheck” on page 5-223	Clear checks.
“deselectCheckAll” on page 5-224	Clear all checks.
“deselectCheckForGroup” on page 5-225	Clear a group of checks.

Name	Description
"deselectCheckForTask" on page 5-225	Clear checks that belong to a specified task or set of tasks.
"deselectTask" on page 5-226	Clear tasks.
"deselectTaskAll" on page 5-226	Clear all tasks.
"displayReport" on page 5-227	Display Model Advisor report.
"exportReport" on page 5-227	Copy report to a specified location.
"filterResultWithExclusion" on page 5-228	Filter objects that have been excluded by user-defined exclusions.
"getBaselineMode" on page 5-229	Get baseline mode setting for the Model Advisor.
"getCheckAll" on page 5-230	Get the IDs of the checks performed by the Model Advisor.
"getCheckForGroup" on page 5-230	Get checks belonging to a check group.
"getCheckForTask" on page 5-231	Get checks belonging to a task.
"getCheckResult" on page 5-231	Get check results.
"getCheckResultData" on page 5-232	Get check result data.
"getCheckResultStatus" on page 5-233	Get pass/fail status of a check or set of checks.
"getGroupAll" on page 5-234	Get the IDs of the groups of tasks performed by the Model Advisor.
"getInputParameters" on page 5-234	Get input parameters of a check.
"getListViewParameters" on page 5-235	Get list view parameters of a check.
"getModelAdvisor" on page 5-236	Get the Model Advisor for a model or subsystem.
"getSelectedCheck" on page 5-237	Get selected checks.
"getSelectedSystem" on page 5-237	Get path of system currently targeted by the Model Advisor.
"getSelectedTask" on page 5-238	Get selected tasks.

Name	Description
"getTaskAll" on page 5-238	Get the IDs of the tasks performed by the Model Advisor.
"Simulink.ModelAdvisor.openConfigUI" on page 5-239	Start the Model Advisor Configuration editor.
"Simulink.ModelAdvisor.reportexists" on page 5-240	Determine whether a report exists for a system or subsystem.
"runCheck" on page 5-240	Run selected checks.
"runTask" on page 5-241	Run checks for selected tasks.
"selectCheck" on page 5-242	Select checks.
"selectCheckAll" on page 5-242	Select all checks.
"selectCheckForGroup" on page 5-243	Select a group of checks.
"selectCheckForTask" on page 5-243	Select checks that belong to a specified task.
"selectTask" on page 5-244	Select tasks.
"selectTaskAll" on page 5-245	Select all tasks.
"setActionEnable" on page 5-245	Set enable/disable status for a check action.
"setBaselineMode" on page 5-246	Set baseline mode for the Model Advisor.
"setCheckErrorSeverity" on page 5-247	Set severity of a check failure.
"setCheckResult" on page 5-248	Set result for the currently running check.
"setCheckResultData" on page 5-248	Set result data for the currently running check.
"setCheckResultStatus" on page 5-249	Set pass/fail status for the currently running check.
"setListViewParameters" on page 5-250	Set list view parameters for a check.
"verifyCheckRan" on page 5-251	Verify that checks have run.
"verifyCheckResult" on page 5-252	Generate a baseline set of check results or compare the current set of results to the baseline results.

Name	Description
"verifyCheckResultStatus" on page 5-253	Verify that a model has passed or failed a set of checks.
"verifyHTML" on page 5-254	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" on page 5-227

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" on page 5-230, "deselectCheckForGroup" on page 5-225, "selectCheck" on page 5-242

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" on page 5-242

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" on page 5-243

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" on page 5-238, "selectCheckForTask" on page 5-243

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" on page 5-244, "getTaskAll" on page 5-238

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" on page 5-245

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" on page 5-240

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" on page 5-240

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 5-236

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" on page 5-254, to generate baseline data.

See Also

"setBaselineMode" on page 5-246, "verifyHTML" on page 5-254, "verifyCheckResult" on page 5-252, "verifyCheckResultStatus" on page 5-253

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" on page 5-238, "getGroupAll" on page 5-234

getCheckForGroup

Purpose

Get checks that belong to check group

Syntax

```
IDs = getCheckForGroup(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" on page 5-231

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" on page 5-230

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 "Define Custom Checks" in the Simulink Verification and Validation documentation.

See Also

"getCheckResultData" on page 5-232, "getCheckResultStatus" on page 5-233

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 "Define Custom Checks" in the Simulink Verification and Validation documentation.

See Also

"getCheckResult" on page 5-231, "getCheckResultStatus" on page 5-233

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” on page 5-231, “getCheckResultData” on page 5-232

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” on page 5-230, “getTaskAll” on page 5-238

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 "Define Custom Checks" in 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, see "Define Custom Checks" in the Simulink Verification and Validation documentation.

See Also

"setListViewParameters" on page 5-250, 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" on page 5-238

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" on page 5-236

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" on page 5-237

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" on page 5-230, "getGroupAll" on page 5-234

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 `slprj` folder in the code generation folder to store reports and other information. If the `slprj` folder does not exist in the

code generation folder, the Model Advisor Configuration editor creates it. For more information, see "Model Reference Simulation Targets".

Simulink.ModelAdvisor.reportexists

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" on page 5-227

runCheck

Purpose

Run currently selected checks

Syntax

```
success = runCheck(ID)
```

Arguments

ID

ID or cell array of IDs of checks to run.

`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" on page 5-242

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" on page 5-244

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" on page 5-242, "selectCheckForGroup" on page 5-243, "deselectCheck" on page 5-223

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" on page 5-242, "selectCheckForGroup" on page 5-243, "deselectCheck" on page 5-223

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" on page 5-225

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" on page 5-225

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" on page 5-226

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" on page 5-226

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, see “Define Custom Checks” in the 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" on page 5-229, "verifyCheckResult" on page 5-252, "verifyHTML" on page 5-254

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, see "Define Custom Checks" in the Simulink Verification and Validation documentation.

See Also

"setCheckResultStatus" on page 5-249

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, see “Define Custom Checks” in the Simulink Verification and Validation documentation.

See Also

“getCheckResult” on page 5-231, “setCheckResultData” on page 5-248, “setCheckResultStatus” on page 5-249

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, see "Define Custom Checks" in the Simulink Verification and Validation documentation.

See Also

"getCheckResultData" on page 5-232, "setCheckResult" on page 5-248, "setCheckResultStatus" on page 5-249

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 "Define Custom Checks" in the Simulink Verification and Validation documentation.

See Also

"getCheckResultStatus" on page 5-233, "setCheckResult" on page 5-248, "setCheckResultData" on page 5-248, "setCheckErrorSeverity" on page 5-247

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, see "Define Custom Checks" in the Simulink Verification and Validation documentation.

See Also

"getListViewParameters" on page 5-235, 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 did not run.

`additionalChecks`

Cell array of IDs for unspecified checks that ran.

Description

The output variable `success` returns `true` if both:

- All the checks specified by *`IDs`* ran.
- Only the checks specified by *`IDs`* ran.

The `missingChecks` argument provides the specified checks that did not run. The `additionalChecks` argument lists unspecified checks that ran.

See Also

"verifyCheckResultStatus" on page 5-253

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" on page 5-246), this method stores the most recent results of running the checks specified by `checkIDs` in

a MAT-file at the location specified by `baseline`. If the method is unable to store the check results at the specified location, it returns `false` in the output variable `success` and the reason for the failure in the output variable `message`. If the Model Advisor is not in baseline mode, this method compares the most recent results of running the checks specified by `checkIDs` with the report specified by `baseline`. If the current results match the baseline results, this method returns`true` as the value of the `success` output variable.

Note: You must run the checks specified by `checkIDs` (see "runCheck" on page 5-240) before invoking `verifyCheckResult`.

This method enables you to compare the most recent check results generated by the Model Advisor with a baseline set of check results. You can use the method to generate the baseline report as well as perform current-to-baseline result comparisons. To generate a baseline report, put the Model Advisor in baseline mode, using "setBaselineMode" on page 5-246. Then invoke this method with the baseline argument set to the location where you want to store the baseline results. To perform a current-to-baseline report comparison, first ensure that the Model Advisor is not in baseline mode (see "getBaselineMode" on page 5-229). Then invoke this method with the path of the baseline report as the value of the `baseline` input argument.

See Also

"setBaselineMode" on page 5-246, "getBaselineMode" on page 5-229, "runCheck" on page 5-240, "verifyCheckResultStatus" on page 5-253

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" on page 5-240) before invoking verifyCheckResultStatus.

See Also

"runCheck" on page 5-240

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" on page 5-246), this method stores the report most recently generated by the Model Advisor at the location specified by *`baseline`*. If the method is unable to store a copy of the report at the specified location, it returns `false` in the output variable `success` and the reason for the failure in the output variable `message`. If the Model Advisor is not in baseline mode, this method compares the report most recently generated by the Model Advisor with the report specified by *`baseline`*. If the current report has exactly the same content as the baseline report, this method returns `true` as the value of the `success` output variable.

This method enables you to compare a report generated by the Model Advisor with a baseline report to determine if they differ. You can use the method to generate the baseline report as well as perform current-to-baseline report comparisons. To generate a baseline report, put the Model Advisor in baseline mode. Then invoke this method with the baseline argument set to the location where you want to store the baseline report. To perform a current-to-baseline report comparison, first ensure that the Model Advisor is not in baseline mode (see "getBaselineMode" on page 5-229). The invoke this method with the path of the baseline report as the value of the *`baseline`* input argument.

See Also

"setBaselineMode" on page 5-246, "getBaselineMode" on page 5-229, "verifyCheckResult" on page 5-252

Introduced in R2006a

Simulink.ModelDataLogs

Container for signal data logs of a model

Description

Note: The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

In releases before R2016a, if you set **Configuration Parameters > Data Import/Export > Signal logging format** to `ModelDataLogs`, Simulink software created instances of the `Simulink.ModelDataLogs` class to contain signal logs while simulating a model. Logging created an instance of this class for a top model and for each model referenced by the top model that contains signals to be logged. Logging assigned the `ModelDataLogs` object for the top model to a variable in the base workspace. The name of the variable is the name specified in the **Configuration Parameters > Data Import/export > Signal logging name** parameter. The default value is `logsout`.

A `ModelDataLogs` object has a variable number of properties. The first property, named `Name`, specifies the name of the model whose signal data the object contains or, if the model is a referenced model, the name of the Model block that references the model. The remaining properties reference objects that contain signal data logged during simulation of the model. The objects may be instances of any of the following types of objects:

- `Simulink.ModelDataLogs`

  Container for the data logs of a model

- `Simulink.SubsysDataLogs`

  Container for the data logs of a subsystem
- `Simulink.Timeseries`

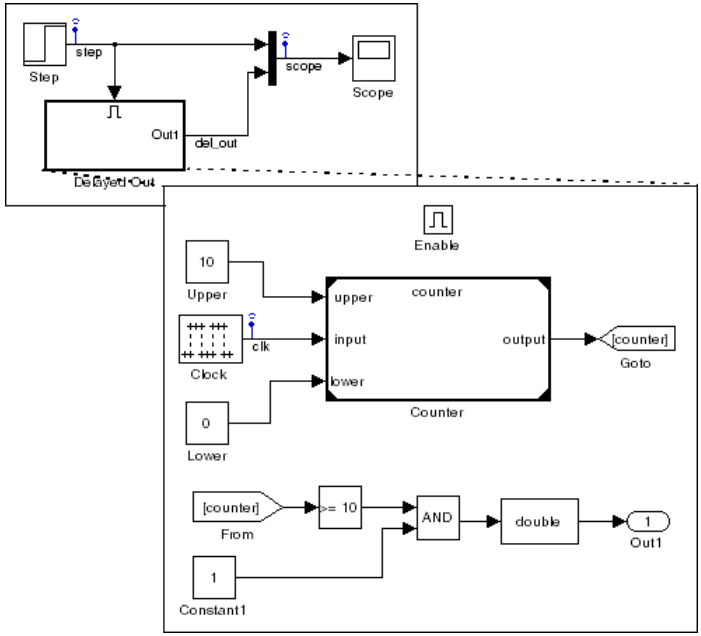
  Data log for any signal except a mux or bus signal
- `Simulink.TsArray`

  Data log for a mux or bus signal

The names of the properties identify the data being logged as follows:

- For signal data logs, the name of the signal
- For a subsystem or model log container, the name of the subsystem or model, respectively

Consider, for example, the following model.



As indicated by the testpoint icons, this model specifies that Simulink software should log the signals named `step` and `scope` in the root system and the signal named `clk` in

the subsystem named Delayed Out. After you simulate this model in a release earlier than R2016a, the MATLAB workspace contains the following variable:

```
Simulink.ModelDataLogs (siglgex):
  Name                    elements  Simulink Class

  scope                      2      TsArray
  step                       1      Timeseries
  ('Delayed Out')            2      SubsysDataLogs
```

You can use fully qualified object names or the Simulink `unpack` command to access the signal data. For example, to access the amplitudes of the `clk` signal in the Delayed Out subsystem in a `logsout` object, enter

```
data = logsout.('Delayed Out').clk.Data;
```

or

```
>> logsout.unpack('all');
>> data = clk.Data;
```

Access Logged Signal Data Saved in `ModelDataLogs` Format

The `Simulink.ModelDataLogs` object contains signal data objects to capture signal logging information for specific model elements.

Model Element	Signal Data Object
Top-level or referenced model	`Simulink.ModelDataLogs`
Subsystem in a model	`Simulink.SubsysDataLogs`
Signal other than a bus or Mux signal	`Simulink.Timeseries`
Bus signal or Mux signal	`Simulink.TsArray`

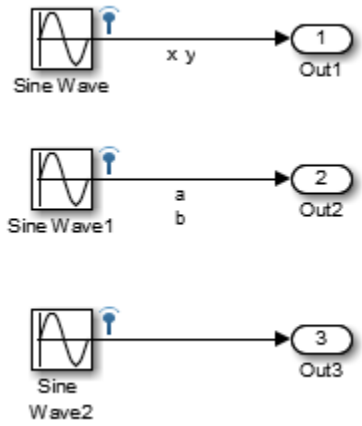
Handling Spaces and Newlines in Logged Names

Signal names in data logs can have spaces or newlines in their names when the signal:

- Is named and the name includes a space or newline character.
- Is unnamed and originates in a block whose name includes a space or newline character.

- Exists in a subsystem or referenced model, and the name of the subsystem, Model block, or of any superior block includes a space or newline character.

The following model shows a signal whose name contains a space, a signal whose name contains a newline, and an unnamed signal that originates in a block whose name contains a newline:



The following example shows how to handle spaces or new lines in logged names, if a model uses `ModelDataLogs` for the signal logging format.

```
logsout
```

```
logsout =

Simulink.ModelDataLogs (model_name):
  Name                    Elements  Simulink Class

  ('x y')                    1      Timeseries
  ('a
b')                    1      Timeseries
  ('SL_Sine
Wave1')          1      Timeseries
```

You cannot access any of the `Simulink.Timeseries` objects in this log using TAB name completion or by typing the name to MATLAB. This syntax is not recognized because the space or newline in each name appears to the MATLAB parser as a separator between identifiers. For example:

```
logsout.x y

??? logsout.x y
              |
Error: Unexpected MATLAB expression.
```

To reference a `Simulink.Timeseries` object whose name contains a space, enclose the element containing the space in single quotes:

```
logsout.('x y')

       Name: 'x y'
     BlockPath: 'model_name/Sine'
     PortIndex: 1
    SignalName: 'x y'
    ParentName: 'x y'
      TimeInfo: [1x1 Simulink.TimeInfo]
          Time: [51x1 double]
          Data: [51x1 double]
```

To reference a `Simulink.Timeseries` object whose name contains a newline, concatenate to construct the element containing the newline:

```
cr=sprintf('\n')
logsout.(['a' cr 'b'])
```

The same techniques work when a space or newline in a data log derives from the name of:

- An unnamed logged signal's originating block
- A subsystem or Model block that contains any logged signal
- Any block that is superior to such a block in the model hierarchy

This code can reference logged data for the signal:

```
logsout.(['SL_Sine' cr 'Wave1'])
```

For names with multiple spaces, newlines, or both, repeat and combine the two techniques as needed to specify the intended name to MATLAB.

Bus Signals

`ModelDataLogs` format stores each logged bus signal data in a separate `Simulink.TsArray` object.

The hierarchy of a bus signal is preserved in the logged signal data. The logged name of a signal in a virtual bus derives from the name of the source signal. The logged name of a signal in a nonvirtual bus derives from the applicable bus object, and can differ from the name of the source signal. See "Composite Signals" for information about those capabilities.

See Also

"Convert Logged Data to Dataset Format", "Migrate Scripts That Use ModelDataLogs API", `Simulink.SubsysDataLogs`, `Simulink.Timeseries`, `Simulink.TsArray`, `who`, `whos`, `unpack`

Introduced before R2006a

Simulink.SimState.ModelSimState class

Package: Simulink.SimState

Access SimState snapshot data

Description

The `Simulink.SimState.ModelSimState` class contains all of the information associated with a "snapshot" of a simulation, including the logged states, the time of the snapshot, and the start time of the simulation. To access these data for a block, use the `getBlockSimState` method or the `loggedStates` property.

Properties

`description`

Specify a description. By default, Simulink generates a 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.

If `loggedStates` is in `Dataset` format, you cannot assign a structure or a `Simulink.SimulationData.Dataset` object with a different number of elements than that of the `Dataset` object used for `loggedStates`.

If the `loggedStates` is in `Structure` format, you cannot assign a `Dataset` object.

Attributes:

`dependent`	`loggedStates` is obtained from the saved states of the block. `loggedStates` depends on the full state being saved in the SimState object, unlike, properties like `description`, which are independent of the save states.

`snapshotTime`

Time at which Simulink takes a “snapshot” of the complete simulation states. This data is read only.

`startTime`

Time at which the simulation starts. This data is read only.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Simulink.ModelWorkspace

Describe model workspace

Description

Instances of this class describe model workspaces. Simulink software creates an instance of this class for each model that you open during a Simulink session. See “Model Workspaces” for more information.

Property Summary

Name	Access	Description
`DataSource`	`RW`	Specifies the source used to initialize this workspace. Valid values are • `'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” on page 5-265	Assign a value to a variable in the model's workspace.

Name	Description
“clear” on page 5-266	Clear the model's workspace.
“evalin” on page 5-266	Evaluate an expression in the model's workspace.
“reload” on page 5-268	Reload the model workspace from the workspace's data source.
“save” on page 5-268	Save the model's workspace to a specified MAT-file.
“saveToSource” on page 5-269	Save the workspace to the MAT-file that the workspace designates as its data source.
“whos” on page 5-270	List the variables in the model workspace.
“getVariable” on page 5-267	Get value of variable from workspace.
“hasVariable” on page 5-267	Determine if variable exists in 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” on page 5-266

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” on page 5-265

getVariable

Purpose

Get value of variable from workspace.

Syntax

```
variableValue = getVariable(workspaceHandle,variableName)
```

Arguments

`workspaceHandle`
Handle to the workspace.

`variableName`
Name of the variable.

Description

This method gets the value of a variable from a workspace.

hasVariable

Purpose

Determine if variable exists in workspace.

Syntax

```
variableExists = hasVariable(workspaceHandle,variableName)
```

Arguments

`workspaceHandle`
Handle to the workspace.

`variableName`
Name of the variable.

Description

This method determines whether a variable exists in a workspace.

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” on page 5-269

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" on page 5-268, "saveToSource" on page 5-269

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" on page 5-268, "reload" on page 5-268

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
```

More About

- "Variables"

Related Examples

- "Model Workspaces"

Introduced before R2006a

Simulink.MSFcnRunTimeBlock

Get run-time information about Level-2 MATLAB S-function block

Description

This class allows a Level-2 MATLAB S-function or other MATLAB program to obtain information from Simulink software and provide information to Simulink software about a Level-2 MATLAB S-Function block. Simulink software creates an instance of this class for each Level-2 MATLAB S-Function block in a model. Simulink software passes the object to the callback methods of Level-2 MATLAB S-functions when it updates or simulates a model, allowing the callback methods to get and provide block-related information to Simulink software. See "Write Level-2 MATLAB S-Functions" for more information.

You can also use instances of this class in MATLAB programs to obtain information about Level-2 MATLAB S-Function blocks during a simulation. See "Access Block Data During Simulation" for more information.

The Level-2 MATLAB S-function template *`matlabroot`*`/toolbox/simulink/blocks/msfuntmpl.m` shows how to use a number of the following methods.

Parent Class

`Simulink.RunTimeBlock`

Derived Classes

None

Property Summary

Name	Description
"AllowSignalsWithMoreThan2D" on page 5-273	enable Level-2 MATLAB S-function to use multidimensional signals.

Name	Description
"DialogPrmsTunable" on page 5-274	Specifies which of the S-function's dialog parameters are tunable.
"NextTimeHit" on page 5-274	Time of the next sample hit for variable sample time S-functions.

Method Summary

Name	Description
"AutoRegRuntimePrms" on page 5-275	Register this block's dialog parameters as run-time parameters.
"AutoUpdateRuntimePrms" on page 5-275	Update this block's run-time parameters.
"IsDoingConstantOutput" on page 5-275	Determine whether the current simulation stage is the constant sample time stage.
"IsMajorTimeStep" on page 5-276	Determine whether the current simulation time step is a major time step.
"IsSampleHit" on page 5-277	Determine whether the current simulation time is one at which a task handled by this block is active.
"IsSpecialSampleHit" on page 5-277	Determine whether the current simulation time is one at which multiple tasks handled by this block are active.
"RegBlockMethod" on page 5-278	Register a callback method for this block.
"RegisterDataTypeFxpBinaryPoint" on page 5-279	Register fixed-point data type with binary point-only scaling.

Name	Description
"RegisterDataTypeFxpFSlopeFixexpBias" on page 5-280	Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias.
"RegisterDataTypeFxpSlopeBias" on page 5-281	Register data type with [Slope Bias] scaling.
"SetAccelRunOnTLC" on page 5-282	Specify whether to use this block's TLC file to generate the simulation target for the model that uses it.
"SetPreCompInpPortInfoToDynamic" on page 5-283	Set precompiled attributes of this block's input ports to be inherited.
"SetPreCompOutPortInfoToDynamic" on page 5-283	Set precompiled attributes of this block's output ports to be inherited.
"SetPreCompPortInfoToDefaults" on page 5-284	Set precompiled attributes of this block's ports to the default values.
"SetSimViewingDevice" on page 5-284	Specify whether block is a viewer.
"SupportsMultipleExecInstances" on page 5-285	
"WriteRTWParam" on page 5-285	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" on page 5-275 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

```
AutoUpdateRuntimePrms;
```

Description

Automatically update the values of the run-time parameters during a call to `ProcessParameters`.

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

IsDoingConstantOutput

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).

When using the `IsSpecialSampleHit` macro, the slower sample time must be an integer multiple of the faster sample time.

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 `Double`, `Single`, `ScaledDouble`, or the fixed-point data type specified by the other arguments of the function.

`false` indicates that the **Data Type Override** setting is to be ignored.

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 5-313.

Use this function if you want to register a fixed-point data type with binary point-only scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use "RegisterDataTypeFxpFSlopeFixexpBias" on page 5-280 to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use "RegisterDataTypeFxpSlopeBias" on page 5-281 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" on page 5-313.

Use this function if you want to register a fixed-point data type by specifying the word length, fractional slope, fixed exponent, and bias. Alternatively, you can use one of the other fixed-point registration functions:

- Use "RegisterDataTypeFxpBinaryPoint" on page 5-279 to register a data type with binary point-only scaling.
- Use "RegisterDataTypeFxpSlopeBias" on page 5-281 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 5-313.

Use this function if you want to register a fixed-point data type with [Slope Bias] scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use "RegisterDataTypeFxpBinaryPoint" on page 5-279 to register a data type with binary point-only scaling.
- Use "RegisterDataTypeFxpFSlopeFixexpBias" on page 5-280 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/simfeatures/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/simfeatures/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/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

Introduced before R2006a

Simulink.NumericType

Specify floating point, integer, or fixed point data type

Description

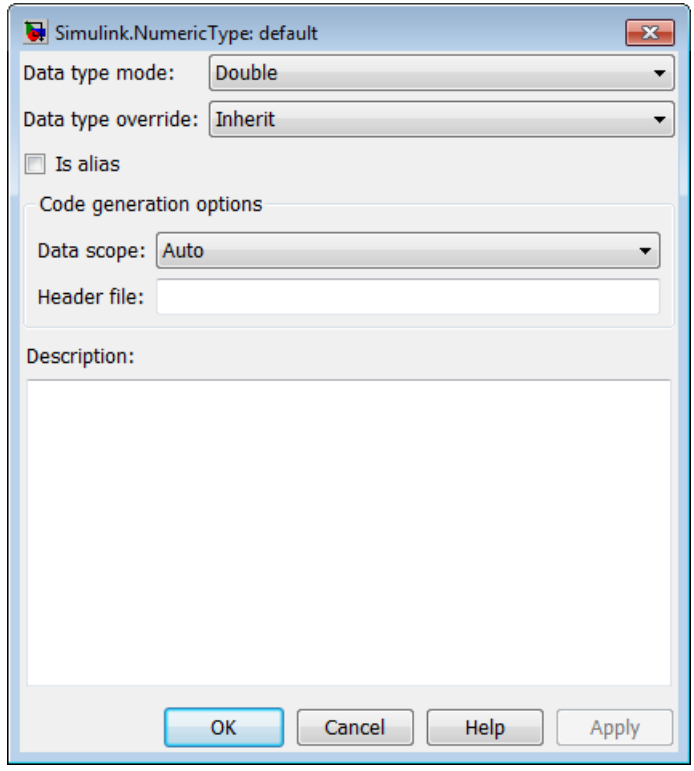
This class allows you to specify a numeric data type as follows:

1. Create an instance of this class in the MATLAB base workspace, a model workspace, or a data dictionary. To create a numeric type in a model workspace, you must disable the **Is alias** option.
2. Set the properties of the object to create a custom floating point, integer, or fixed point data type.
3. Assign the data type to all signals and parameters of your model that you want to conform to the data type.

Assigning a data type in this way allows you to change the data types of the signals and parameters in your model by changing the properties of the object that describe them. You do not have to change the model itself.

You can use objects of this class, instead of the class `Simulink.AliasType`, to define and name your own fixed-point data types. `Simulink.AliasType` objects can create an alias for a fixed-point data type, but cannot define one.

Property Dialog Box



Data type mode

Data type of this numeric type. The options are listed in this table.

Option	Description
`Double`	Same as the MATLAB `double` type.
`Single`	Same as the MATLAB `single` type.
`Boolean`	Same as the MATLAB `boolean` type.
`Fixed-point: unspecified scaling`	A fixed-point data type with unspecified scaling.

Option	Description
`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 that apply to the mode and to disable other controls that do not apply. Selecting a fixed-point data type mode can, depending on the other dialog box options that you select, cause the model to run only on systems that have a Fixed-Point Designer option installed.

Data type override

Data type override setting for this numeric type. The options are listed in this table.

Option	Description
Inherit (default)	Data type override setting for the context in which this numeric type is used (block, signal, Stateflow chart in Simulink) applies to this numeric type.
Off	Data type override setting does not affect this numeric type.

Is alias

If you select this option for a workspace object of this type, Simulink software uses the name of the object as the data type for all objects that specify the object as its data type. Otherwise, Simulink software uses the data type mode of the data type as its name, or, if the data type mode is a fixed-point mode, Simulink software generates a name that encodes the type properties, using the encoding specified by Fixed-Point Designer.

Data scope

Specifies whether the data type definition is imported from, or exported to, a header file during code generation. The possible values are listed in this table.

Value	Action
`Auto` (default)	If no value is specified for **Header file**, export the type definition to *`model`*`_types.h`. *`model`* is the model name.

Value	Action
	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`. *`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`. *`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`.

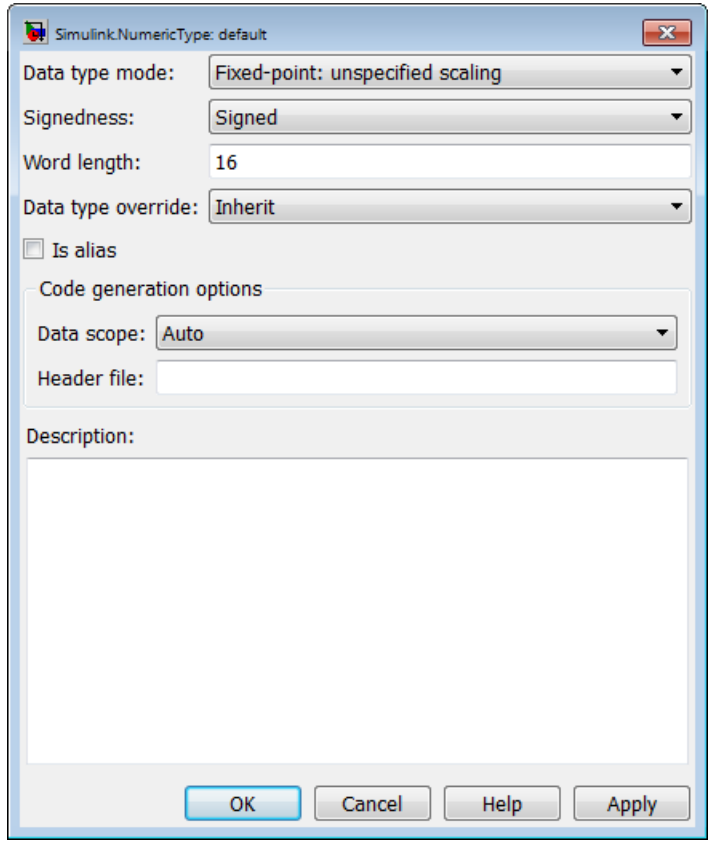
By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify **Header file** as `<myTypes.h>`.

Description

Description of this data type. This field is intended for use in documenting this data type. Simulink software ignores it.

Signedness

Specifies whether the data type is signed or unsigned, or inherits its signedness. Set the option to `Signed`, `Unsigned`, or `Auto`. This option is enabled only for fixed-point data type modes as shown.

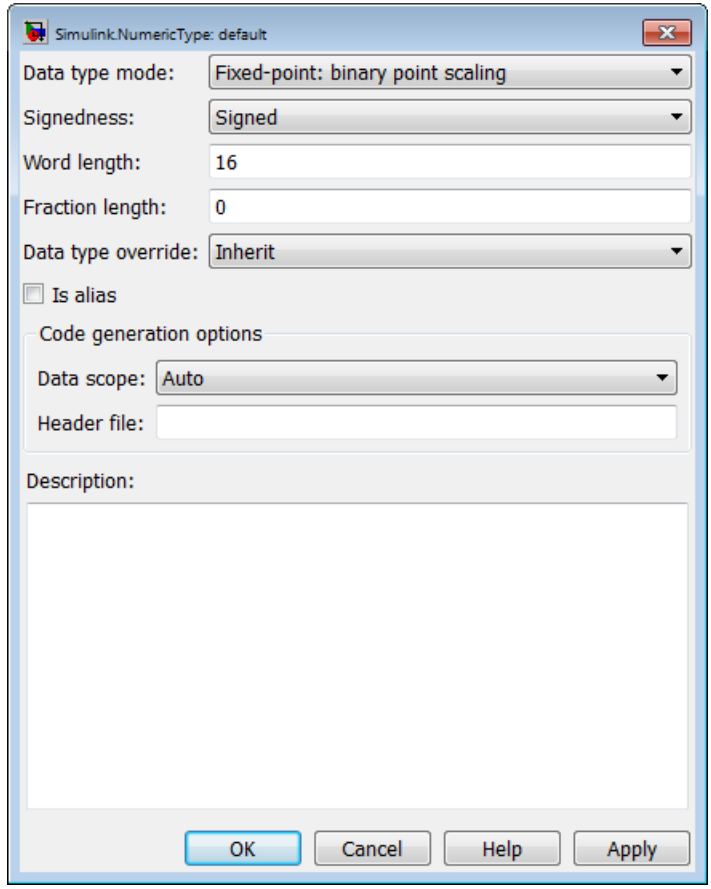


Word length

Word length in bits of the fixed-point data type. This option is enabled only for fixed-point data type modes.

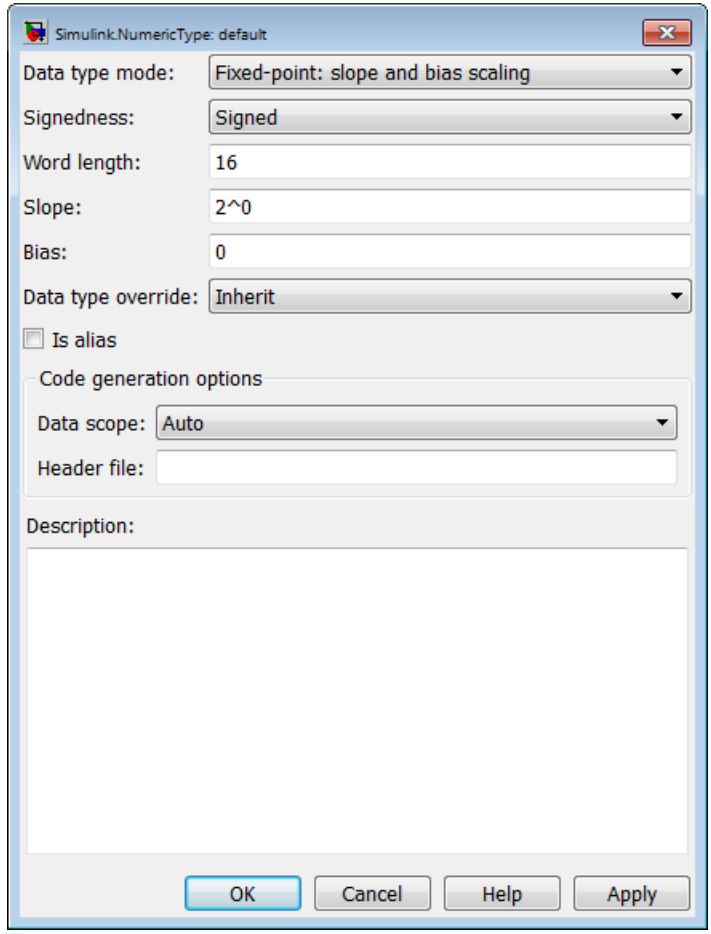
Fraction length

Number of bits to the right of the binary point. This option is enabled only if the data type mode is `Fixed-point: binary point scaling`.



Slope

Slope for slope and bias scaling. This option is enabled only if the data type mode is `Fixed-point: slope and bias scaling`.



Bias

Bias for slope and bias scaling. This option is enabled only if the data type mode is `Fixed-point: slope and bias scaling`. See the preceding figure.

Properties

Name	Access	Description
`Bias`	`RW`	Bias used for slope and bias scaling of a fixed-point data type. This field is intended for use by Fixed-Point Designer. (Bias)
`DataScope`	`RW`	A string specifying whether the data type definition is 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)
`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 in the object Property dialog box, but can be accessed at the 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

Name	Access	Description
		intended for use by Fixed-Point Designer. (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)
`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 in the object Property dialog box, but can be accessed at the command prompt.

Name	Access	Description
`WordLength`	`RW`	Integer that specifies the word size of this data type. This field is intended for use by Fixed-Point Designer. This property appears only if `DataTypeMode` is `Fixed-point`. (Word Length)

Methods

Name	Description
`isboolean`	Determine whether data type is Boolean. Returns `1` when the `DataTypeMode` is `'Boolean'`, `0` otherwise.
`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.
`isscalingbinarypoint`	Determine whether data type has binary point scaling.

Name	Description
	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.
`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.
`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.

More About

- “Data Types Supported by Simulink”
- “About Data Types in Simulink”

See Also

`Simulink.AliasType`

Related Examples

- “Validate a Floating-Point Embedded Model”

- “Control Signal Data Types”
- “Create a Named Fixed-Point Data Type in the Generated Code”
- “Create and Apply User-Defined Data Types”

Introduced before R2006a

Simulink.Parameter

Specify value, value range, data type, and other properties of block parameter

Description

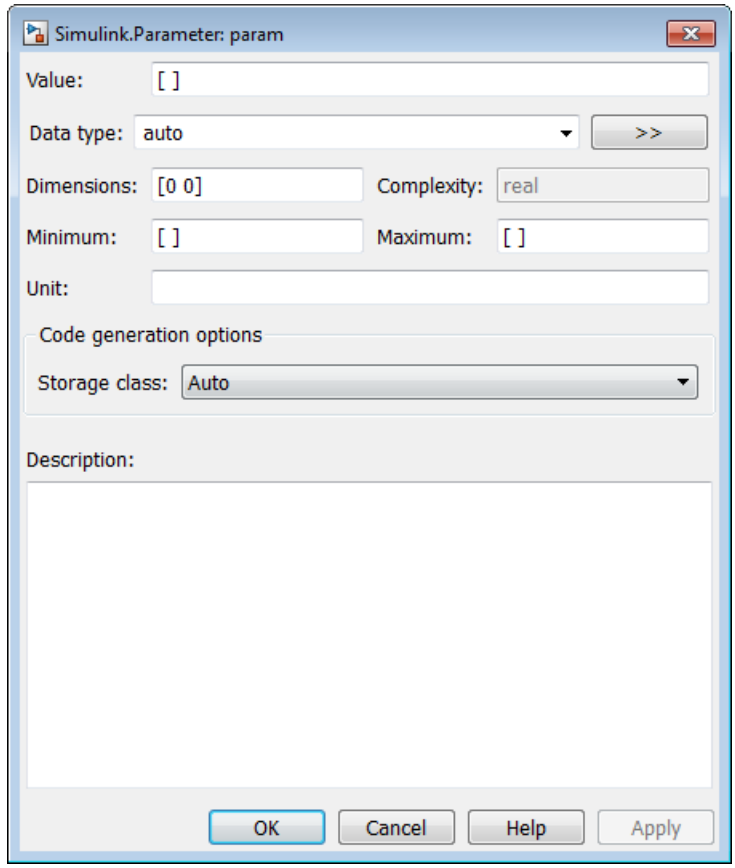
This class enables you to create workspace objects that you can then use as the values of block parameters — for example, the value of the `Gain` parameter of a Gain block. You can create a `Simulink.Parameter` object in the base MATLAB workspace or a model workspace. However, to create the object in 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 "Tune and Experiment with Block Parameter Values" 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



Value

Ideal real-world value of the parameter. You can use MATLAB expressions to specify the dimensions and complexity of the parameter. The table shows examples of valid ways to specify a value.

Expression	Description
`15.23`	Specifies a scalar value
`[3 4; 9 8]`	Specifies a matrix
`3+2i`	Specifies a complex value

Expression	Description
`struct('A',20,'B',5)`	Specifies a structure for initialization of a bus with two signal elements, `A` and `B`, with double-precision values `20` and `5`. 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".

To use a `Simulink.Parameter` object to represent a parameter of a particular numeric data type, specify the ideal value using the **Value** property, and specify the type using the **Data type** property.

For example, to use a parameter object to represent the number `single(32.5)`, specify the **Value** property as `32.5` and the **Data type** property as `single`.

If you set the **Value** property by using a typed expression such as `single(32.5)`, the **Data type** property changes to reflect the new type. A best practice is to use an expression that is not typed. This best practice helps you to avoid accumulating numerical error through repeated quantizations or premature data type saturation, especially for fixed-point data types.

Data type

Data type of the parameter. When you simulate or generate code, Simulink casts the ideal parameter value to the specified data type.

You can either select a data type from the drop-down list, or specify the name of a type using a string. If you specify a string, it must evaluate to one of these entities:

- A built-in data type that Simulink supports. For a list of supported data types, see "Data Types Supported by Simulink".
- A `Simulink.NumericType` object. You can use this technique to specify a fixed-point data type.
- A `Simulink.AliasType` object.
- A `Simulink.Bus` object.

  You can use the Bus Editor to define or edit a `Simulink.Parameter` object that uses a bus object as the data type. In the Bus Editor, use one of these approaches:

  - Select **File** > **Create/Edit a Simulink.Parameter object**.

- On the toolbar, click **Create/Edit a Simulink.Parameter object**  .

  To customize the new object, edit the code in the MATLAB Editor.

- The name of an enumerated data type. Prefix the name with `Enum:`. For example, to use an enumerated data type called `myEnumType`, specify `'Enum: myEnumType'`.

If you select `auto`, which is the default setting, the target block determines the data type. For example, if you use the object to specify the **Constant value** parameter of a `Constant` block whose output data type is `int8`, the block casts the parameter value to `int8` when you simulate or generate code.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Data type** parameter. For more information, see "Specify Data Types Using Data Type Assistant".

Unit

Physical unit in which this value is expressed (for example, inches). To specify a unit, begin typing in the text box. As you type, the parameter displays potential unit string matches. For more information, see "Unit Specification in Simulink Models".

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" in the Simulink Coder documentation and "Simulink Package Custom Storage Classes" in the Embedded Coder documentation.

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 "Data Alignment for Code Replacement"). Simulink software ignores this setting.

Description

Description of this parameter. This field is intended for use in documenting this parameter.

If you have an Embedded Coder license, you can add the signal description as a comment for the variable declaration in generated code.

- Specify a storage class for the signal object other than `Auto`.
- On the **Code Generation > Comments** pane of the Model Configuration Parameters dialog box, select the model configuration parameter **Simulink data object descriptions**. For more information, see "Simulink data object descriptions".

Properties

Name	Access	Description
`Value`	`RW`	Value of this parameter. (Value)
`CoderInfo`	`R`	Information used by Simulink Coder software for generating code for this parameter. The value of this property is an object of `Simulink.CoderInfo` 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)
`Unit`	`RW`	Physical unit in which this parameter's value is expressed. (Unit)
`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)

More About

- "Data Objects"

- “Data Types Supported by Simulink”
- “MPT Data Object Properties”

See Also

`AUTOSAR.Parameter` | `Simulink.CoderInfo` | `Simulink.Signal`

Related Examples

- “Determine Where to Store Data for Simulink Models”
- “Set Block Parameter Values”
- “Define Data Classes”
- “Control Parameter Representation and Declare Tunable Parameters in the Generated Code”

Introduced before R2006a

Simulink.RunTimeBlock

Allow Level-2 MATLAB S-function and other MATLAB programs to get information about block while simulation is running

Description

This class allows a Level-2 MATLAB S-function or other MATLAB program to obtain information about a block. Simulink software creates an instance of this class or a derived class for each block in a model. Simulink software passes the object to the callback methods of Level-2 MATLAB S-functions when it updates or simulates a model, allowing the callback methods to get block-related information from and provide such information to Simulink software. See "Write Level-2 MATLAB S-Functions" in Writing S-Functions for more information. You can also use instances of this class in MATLAB programs to obtain information about blocks during a simulation. See "Access Block Data During Simulation" for more information.

Note `Simulink.RunTimeBlock` objects do not support MATLAB sparse matrices. For example, the following line of code attempts to assign a sparse identity matrix to the run-time object's output port data. This line of code in a Level-2 MATLAB S-function produces an error:

```
block.Outport(1).Data = speye(10);
```

Parent Class

None

Derived Classes

`Simulink.MSFcnRunTimeBlock`

Property Summary

Name	Description
"BlockHandle" on page 5-308	Block's handle.
"CurrentTime" on page 5-308	Current simulation time.
"NumDworks" on page 5-309	Number of discrete work vectors used by the block.
"NumOutputPorts" on page 5-309	Number of block output ports.
"NumContStates" on page 5-309	Number of block's continuous states.
"NumDworkDiscStates" on page 5-310	Number of block's discrete states
"NumDialogPrms" on page 5-310	Number of parameters that can be entered on S-function block's dialog box.
"NumInputPorts" on page 5-310	Number of block's input ports.
"NumRuntimePrms" on page 5-311	Number of run-time parameters used by block.
"SampleTimes" on page 5-311	Sample times at which block produces outputs.

Method Summary

Name	Description
"ContStates" on page 5-311	Get a block's continuous states.
"DataTypeIsFixedPoint" on page 5-312	Determine whether a data type is fixed point.
"DatatypeName" on page 5-312	Get name of a data type supported by this block.

Name	Description
“DatatypeSize” on page 5-313	Get size of a data type supported by this block.
“Derivatives” on page 5-313	Get a block's continuous state derivatives.
“DialogPrm” on page 5-314	Get a parameter entered on an S-function block's dialog box.
“Dwork” on page 5-314	Get one of a block's DWork vectors.
“FixedPointNumericType” on page 5-315	Determine the properties of a fixed-point data type.
“InputPort” on page 5-315	Get one of a block's input ports.
“OutputPort” on page 5-316	Get one of a block's output ports.
“RuntimePrm” on page 5-317	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" on page 5-313

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" on page 5-312

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" on page 5-317

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" on page 5-316

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`. This run-time parameter is a `Simulink.BlockData` object of type Simulink.BlockRunTimePrmData.

See Also

`ssGetRunTimeParamInfo`

Introduced before R2006a

Simulink.SampleTime class

Package: Simulink

Object containing sample time information

Description

The `SampleTime` class represents the sample time information associated with an individual sample time.

Use the methods `Simulink.Block.getSampleTimes` and `Simulink.BlockDiagram.getSampleTimes` to retrieve the values of the `SampleTime` properties for a block and for a block diagram, respectively.

Properties

`Value`

A two-element array of doubles that contains the period and offset of the sample time

`Description`

A 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.scopes.TimeScopeConfiguration class

Package: Simulink.scopes

Configure Scope and Time Scope for programmatic access

Description

The `Simulink.scopes.TimeScopeConfiguration` object contains the scope configuration information for the Scope and Time Scope block.

Construction

Call the `get_param` function, specifying a Scope and Time Scope block.

`htsc = get_param(gcbh,'ScopeConfiguration')` constructs a new 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 property to control which display has its axes colors, line properties, marker properties, and visibility changed. 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 automatically scales 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** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - 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. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable.

`AxesScalingNumUpdates` — Number of updates before scaling y-axes

`'10'` (default) | positive integer string

Number of updates before scaling y-axes. Specified as a positive integer string.

Dependency: Activate this property by setting `AxesScaling` to `'After N Updates'`.

Block Configuration Property: **Number of updates**

`BufferLength`

Number of data points in buffer

Specify the size of the buffer that the scope holds in its memory cache. Memory is limited by available memory on your system. If your signal has *M* rows of data and *N* data points in each row, *M* x *N* is the number of data points per time step. Multiply this result by the number of time steps for your model to obtain the required buffer length. For example, if you have 10 rows of data with each row having 100 data points and your run will be 10 time steps, you should enter 10,000 (which is 10 x 100 x 10) as the buffer length.

Default: `5000`

DataLogging

Specify whether to log data to the workspace

Set this property to `true` to log data to the workspace. Data is logged as a dataset object. See Simulink.SimulationData.Dataset for information. Set this property to `false` to prevent the scope from logging data.

This property does not apply to Floating Scopes and Scope Viewers.

Default: `false`

DataLoggingVariableName

Name of variable for logged data

Specify as a string the name of the variable in the MATLAB workspace to which the scope logs data. Any existing variable is overwritten.

This property does not apply to Floating Scopes and Scope Viewers.

Default: `ScopeData`

DataLoggingLimitDataPoints

Specify whether to limit number of logged data points

Set this property to `true` to limit the number of data points at the end of the simulation data that the scope logs. Set this property to `false` to log all data points.

Default: `false`

DataLoggingMaxPoints

Maximum number of data points to log

Specify the maximum number of data points at the end of the data to log to the workspace.

Default: `5000`

DataLoggingDecimateData

Specify whether to decimate logged data

Set this property to `true` to decimate logged data. Set this property to `false` to log all data points.

This property does not apply to Floating Scopes and Scope Viewers.

Default: `false`

`DataLoggingDecimation`

Logged data decimation rate

Specify the rate at which to log decimation data. The scope logs every `N`th data point, where `N` is the decimation factor you specify.

This property does not apply to Floating Scopes and Scope Viewers.

Default: `2`

`DataLoggingSaveFormat`

Logged data format

Specify the format in which to save logged data. Unless otherwise noted, you can save logged data for single- and multi-port data, sample-based and frame-based data, variable-size data, MAT-file logging, and external mode archiving. Valid values are:

- `Structure With Time` — Save logged data as a structure with associated time information to the MATLAB workspace. Structure With Time format does not support multirate data.
- `Structure` — Save logged data as a structure to the MATLAB workspace. Structure format does not support multirate data.
- `Array` — Save logged data as an array with associated time information to the MATLAB workspace. Array format does not support multiport sample-based data, single port or multiport frame-based data, variable-size data, or multirate data.
- `Dataset` — Save logged data as a dataset object to the MATLAB workspace. Dataset format does not support variable-size data, MAT-file logging, or external mode archiving. See Simulink.SimulationData.Dataset for information.

This property does not apply to Floating Scopes and Scope Viewers.

Default: `Dataset`

`DisplayFullPath` — Full path display control

`false` | `true`

Display the full block path on the scope title bar, specified as one of these values:

- `false` — Full path is not displayed.
- `true` — Full path is displayed.

Block Configuration Property: **Display the full path**

Default: `false`

`FrameBasedProcessing` — Frame-based processing of signals.

`false` (default for Time Scope block) | `true` (default for Scope block)

Frame-based processing of signals, specified as one of these values:

- `false` — Process signal values in a channel at each time interval (sample based).
- `true` — Process signal values in a channel as a group of values from multiple time intervals (frame based). Frame-based processing is available only with discrete input signals.

Block Configuration Property: **Input processing**

`LayoutDimensions`

Layout grid dimensions

Specify the layout grid dimensions as a 2-element vector: `[numberOfRows, numberOfColumns]`. You can use no more than sixteen rows and sixteen 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'`

`MinimizeControls`

Minimize menus and toolbar

Set this property to `true` to hide the menus and toolbar. Set this property to `false` to display the menus and toolbar. This property is not active if the scope is docked. This property is Tunable

Default: `false`

`Name`

Caption to display on the Scope or Time Scope window

Specify as a string the caption to display on the scope window. This property is Tunable.

Default: `'Scope'` for Simulink Scope and `'Time Scope'` for DSP System Toolbox.

`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.

This property does not apply to Floating Scopes and Scope Viewers.

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`

Scope or 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, a scope window appears in the center of your screen with a width of 410 pixels and height of 300 pixels.

`ReduceUpdates`

Specify how often a Scope window updates signal data. Reducing updates can improve simulation performance.

- `0` (false) — Update displays at each time step.

- 1 (true) — Record data at each time step, but update displays periodically at a rate not exceeding 20 hertz.

You can also set this property from the Scope window menu. Select **Simulation** > **Reduce Updates to Improve Performance**.

Default: 1

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`

SampleTime

Specify the sampling time in seconds. To use the sample time of the input signal, enter `-1`.

This property does not apply to Floating Scopes and Scope Viewers.

Default: `-1`

ShowLegend

When you set this property to `true`, the scope displays a legend with input channel string names. When you set this property to `false`, the scope does not display a legend. This property is Tunable

See FrameBasedProcessing for information on input channels.

When set, `ActiveDisplay` controls which display is updated.

Default: `false`

ShowTimeAxisLabel

When you set this property to `true`, the scope displays the *time*-axis label. When you set this property to `false`, the scope does not display the *time*-axis label, but still displays

tick marks and other *time*-axis items. This property applies only if the `TimeAxisLabels` property is `All` or `Bottom`. This property is Tunable

`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 property 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 units 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. Enter `%<SignalLabel>` to use the signal labels in the Simulink Model as the axes titles. This property is Tunable.

When set, `ActiveDisplay` controls which display is updated.

Default: `''`

`Visible`

Specify whether the scope is visible

When you set this property to `true`, the scope is visible. When you set this property to `false` or `0`, the scope is hidden. This property is Tunable.

Default: `1`for Simulink Scope and `0` for DSP System Toolbox Time Scope.

`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. 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 Scope Configuration Object

Create a new Simulink model.

```
mdl='scopemdl';
new_system(mdl);
```

Add a new Scope block to the model.

```
add_block('Simulink/Sinks/Scope', [mdl '/myScope']);)
```

Call the `get_param` function to retrieve the default Scope block properties.

```
htsc = get_param([mdl '/myScope'],'ScopeConfiguration')
```

```
htsc =

  Scope configuration with properties:

                        Name: 'myScope
                    Position: [680 390 560 420]
                     Visible: 0
               ReduceUpdates: 1
      OpenAtSimulationStart: 0
            DisplayFullPath: 0
              NumInputPorts: '1'
          LayoutDimensions: [1 1]
```

```
                SampleTime: '-1'
          FrameBasedProcessing: 0
                 MaximizeAxes: 'Off'
             MinimizeControls: 0
                  AxesScaling: 'Manual'
       AxesScalingNumUpdates: '10'
                    TimeSpan: 'Auto'
       TimeSpanOverrunAction: 'Wrap'
                  TimeUnits: 'none'
          TimeDisplayOffset: '0'
             TimeAxisLabels: 'Bottom'
          ShowTimeAxisLabel: 0
              ActiveDisplay: 1
                      Title: ''
                 ShowLegend: 0
                   ShowGrid: 1
        PlotAsMagnitudePhase: 0
                    YLimits: [-10 10]
                     YLabel: ''
                DataLogging: 0
     DataLoggingVariableName: 'ScopeData'
  DataLoggingLimitDataPoints: 0
        DataLoggingMaxPoints: '5000'
     DataLoggingDecimateData: 0
       DataLoggingDecimation: '2'
       DataLoggingSaveFormat: 'Dataset'
```

- “Control Scopes Programmatically”

Simulink.sdi.Dataset class

Package: Simulink.sdi
Superclasses: Simulink.SimulationData.Dataset

Create `Simulink.sdi.Dataset` object

Description

Simulink creates `Simulink.sdi.Dataset` objects to store data elements when you stream signals to the Simulation Data Inspector and also stream them to the base workspace.

The `Simulink.sdi.Dataset` class is a subclass of Simulink.SimulationData.Dataset.

Construction

Simulink creates `Simulink.sdi.Dataset` objects when you stream signals to the Simulation Data Inspector and also stream them to the base workspace. To turn on streaming to the base workspace, use the parameter setting command:

```
set_param('sldemo_absbrake','StreamToWorkspace','on');
```

Properties

`Name` — Name of the data set

same as the streamed data run (default) | string

Name of the data set, specified as a string.

Example: `'Run 1: sldemo_absbrake'`

`Total Elements` — Total number of elements

double

Total number of elements in data set, specified as a double. This property is read only.

Methods

See the methods of the superclass Simulink.SimulationData.Dataset, plus the following method.

getRun	Get run data from data set

Examples

Create Data Set from Streamed Data

```
% Open the model
load_system('sldemo_absbrake');

% Streamed logged signals directly to the Data Inspector and automatically
% export them at the end of simulation
Simulink.sdi.changeLoggedToStreamed('sldemo_absbrake');
set_param('sldemo_absbrake','StreamToWorkspace','on');

% Simulate the model and stream data to base workspace
out = sim('sldemo_absbrake','ReturnWorkspaceOutputs','on');

% Get streamed data set
streamed = out.get('streamout')
```

```
streamed =

  Simulink.sdi.Dataset
  Package: Simulink.sdi

  Characteristics:
              Name: 'Run 18: sldemo_absbrake'
    Total Elements: 2

  Elements:
    1 : 'yout'
    2 : 'slp'
```

- “Stream Data to the Simulation Data Inspector”

See Also

Simulink.SimulationData.Dataset

Introduced in R2015b

getRun

Class: Simulink.sdi.Dataset
Package: Simulink.sdi

Get run data from data set

Syntax

```
run = getRun(streamout)
```

Description

`run = getRun(streamout)` returns a Simulink.sdi.Run object with meta data that belongs to the streamed data set.

Input Arguments

`streamout` — Streamed data set

`Simulink.sdi.Dataset` object

Streamed data set from the base workspace, specified as a Simulink.sdi.Dataset object.

Output Arguments

`run` — Simulation Data Inspector run

`Simulink.sdi.Run` object

Simulation Data Inspector run associated with the streamed data, returned as a Simulink.sdi.Run object.

Examples

Get Run Meta Data

```
% Open the model
```

```
load_system('sldemo_absbrake');

% Streamed logged signals directly to the Data Inspector and automatically
% export them at the end of simulation
Simulink.sdi.changeLoggedToStreamed('sldemo_absbrake');
set_param('sldemo_absbrake','StreamToWorkspace','on');

% Simulate the model and stream data to base workspace
out = sim('sldemo_absbrake','ReturnWorkspaceOutputs','on');

% Get streamed data set
streamed = out.get('streamout');

% Get the run meta data
run = getRun(streamed);
```

- “Stream Data to the Simulation Data Inspector”

See Also

Simulink.sdi.Run

Introduced in R2015b

Simulink.sdi.DiffRunResult class

Package: Simulink.sdi

Results from comparing two simulation runs

Description

The `Simulink.sdi.DiffRunResult` class manages the results from comparing two runs. A `Simulink.sdi.DiffRunResult` object contains a `Simulink.sdi.DiffSignalResult` object for each signal compared.

Construction

The function `Simulink.sdi.compareRuns` returns a `Simulink.sdi.DiffRunResult` object.

Properties

`count` — Compared signals

integer

Number of compared signal results, stored as an integer.

`dateCreated` — Comparison creation date

object

Date of at time of object creation, stored as a `datetime` object.

Example:

`matlabVersion` — Version of MATLAB used

string

Version of MATLAB used to create an instance of `Simulink.sdi.DiffRunResult`, stored as a string.

Example:

`runID1` — Run identifier

integer

Run ID, a unique number identifying the first run compared, stored as a string.

`runID2` — Run identifier

integer

Run ID, a unique number identifying the second run compared, stored as a string.

Methods

getResultByIndex	Return signal comparison result

Examples

Compare Runs and Test If Signals Match

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
% simOut in the base workspace
runID1 = Simulink.sdi.createRun('First Run','namevalue',{'simOut'},{simOut});

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create another Simulation Data Inspector run
runID2 = Simulink.sdi.createRun('Second Run','namevalue',{'simOut'},{simOut});

% Compare two runs 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)...
           ' do not match']);
   end
end
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.compareRuns` | `Simulink.sdi.createRun` | Simulink.sdi.DiffSignalResult

Introduced in R2012b

Simulink.sdi.DiffRunResult.getResultByIndex

Class: Simulink.sdi.DiffRunResult
Package: Simulink.sdi

Return signal comparison result

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` — Signal result array index

integer

An index to the array of `Simulink.sdi.DiffSignalResult` objects contained in a `Simulink.sdi.DiffRunResult` object, specified as an integer.

Output Arguments

`diffSignalObj` — Signal comparison difference

object

Results of comparing two signals between simulation runs, returned as a `Simulink.sdi.DiffSignalResult` object.

Examples

Determine Results of Signal Comparison

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 simOut in the base workspace
runID1 = Simulink.sdi.createRun('First Run','namevalue',{'simOut'},{simOut});

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample','SaveOutput','on',...
             'SaveFormat','StructureWithTime',...
             'ReturnWorkspaceOutputs','on');

% Create another Data Inspector run and get signal IDs
runID2 = Simulink.sdi.createRun('Second Run','namevalue',{'simOut'},{simOut});

% 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
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.compareRuns` | Simulink.sdi.DiffRunResult | Simulink.sdi.DiffSignalResult | Simulink.sdi.Run | Simulink.sdi.Signal

Introduced in R2012b

Simulink.sdi.DiffSignalResult class

Package: Simulink.sdi

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

`Diff` — Difference

object

A MATLAB timeseries object specifying the value differences after synchronizing the two time series data.

`Match` — Signal match indicator

logical

A Boolean indicating if the two MATLAB timeseries objects match according to the specified tolerance and time synchronization options.

Example:

`MaxDifference` — Maximum difference
double

The maximum numerical difference between the two signals, stored as a double.

Example:

`SignalID1` — First signal identifier
integer

Signal ID, a unique number identifying the first signal compared, stored as an integer.

`SignalID2` — Second signal identifier
integer

Signal ID, a unique number identifying the second signal compared, stored as an integer.

`Sync1` — First synchronization timeseries
object

A MATLAB timeseries object specifying first time series after synchronization has been applied.

`Sync2` — Second synchronization timeseries
object

A MATLAB timeseries object specifying second time series after synchronization has been applied.

`Tol` — Absolute tolerance
object

A MATLAB timeseries object specifying the absolute tolerance value at each synchronized time point.

Examples

Compare Signals and Test If Signals Match

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',{'MyData'},{simOut});

sig1 = signalIDs(1);
sig2 = signalIDs(2);

% Compare two signals, which returns the results in an
% instance of Simulink.sdi.DiffSignalResult
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;
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.compareRuns` | `Simulink.sdi.createRun` | Simulink.sdi.DiffRunResult

Simulink.sdi.Run class

Package: Simulink.sdi

Manages signal data and metadata of simulation run

Description

The `Simulink.sdi.Run` object contains the signal information for one simulation run, which includes the 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` — Run creation date

object

Date and time of the run, stored as a `datetime` object.

`description` — Run description

empty string (default)

Description of the run, specified as a string. The default value is an empty string.

`id` — Run identifier

integer

Unique number to identifying a run, stored as an integer.

`name` — Run name

empty string (default)

Name of the run, specified as a string. The default value is an empty string.

`signalCount` — Number of signals in run

integer

Number of signals in the run, stored as an integer.

`tag` — Information tag

empty string (default)

Tag for categorization, identification, or attaching other information to this run, specified as a string. The default value is an empty string.

Methods

getSignal	Return `Simulink.sdi.Signal` object by signal ID
getSignalByIndex	Return`Simulink.sdi.Signal` object by index
getSignalIDByIndex	Return signal ID at array index
isValidSignalID	Determine if signal ID is valid within run

Examples

Create Run From Workspace Data

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 in the base workspace
runID = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Get the handle to the run object, Simulink.sdi.Run,
```

```
% corresponding to the new run ID
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 run by setting the checked property to 'true'
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;
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRun` | `Simulink.sdi.view`

Introduced in R2012b

getSignal

Class: Simulink.sdi.Run
Package: Simulink.sdi

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 identifier

integer

Unique number identifying a signal in a run, specified as an integer.

Output Arguments

`signalObj` — Signal object

object

A signal within a run object, `Simulink.sdi.Run`, returned as a `Simulink.sdi.Signal` object.

Examples

Plot Signals In a Run

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',{'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 objects for each signal
% in the run and select for plotting
for i = 1:numSignals
   signalObjs(i) = runObj.getSignal(signalIDs(i));
   signalObjs(i).checked = true;
end
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.createRun` | Simulink.sdi.Run | Simulink.sdi.Signal

Introduced in R2012b

getSignalByIndex

Class: Simulink.sdi.Run
Package: Simulink.sdi

Return`Simulink.sdi.Signal` object by index

Syntax

```
signalObj = runObj.getSignalByIndex(index)
```

Description

`signalObj = runObj.getSignalByIndex(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` — Signal array index

integer

Index to the array of signals contained in a `Simulink.sdi.Run` object, specified as an integer. The first index in the array is 1.

Output Arguments

`signalObj` — Signal object

object

Signal in a run object, `Simulink.sdi.Run`, returned as a `Simulink.sdi.Signal` object.

Examples

Get the Signal Object in a Run

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','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 Simulink.sdi.Signal object for the first signal in the run
if numSignals > 0
   signalObj = runObj.getSignalByIndex(1);
end
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRun` | Simulink.sdi.Run | Simulink.sdi.Signal

Introduced in R2012b

getSignalIDByIndex

Class: Simulink.sdi.Run
Package: Simulink.sdi

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` — Signal array index

integer

Index to the array of signals contained in a `Simulink.sdi.Run` object, specified as an integer. The first index in the array is 1.

Output Arguments

`signalID` — Signal identifier

integer

Unique number identifying a signal in a run, returned as an integer.

Examples

Get the Signal ID in a Run

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
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.compareSignals` | `Simulink.sdi.createRun` | `Simulink.sdi.getRun` | Simulink.sdi.Run | Simulink.sdi.Signal

Introduced in R2012b

isValidSignalID

Class: Simulink.sdi.Run
Package: Simulink.sdi

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 identifier

integer

Unique number identifying a signal stored in the run object, specified as an integer.

Output Arguments

`isValid` — Valid signal indicator

boolean

A Boolean value: `true`, if the signal exists; `false`, otherwise.

Examples

Check That a Signal ID Is Valid

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
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRun` | Simulink.sdi.Run | Simulink.sdi.Run.getSignal | Simulink.sdi.Run.getSignalByIndex | Simulink.sdi.Run.isValidSignalID | Simulink.sdi.Signal

Introduced in R2012b

Simulink.sdi.Signal class

Package: Simulink.sdi

Manages signal time series data and metadata

Description

The `Simulink.sdi.Signal` object contains the signal 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)

`blockSource` — Block path

string

Path to the block that defines the signal, stored as a string.

`channel` — Channel index

integer array

Channel index, stored as an integer array. This property applies to matrix data only. Matrix data is flattened into a scalar time series using channels.

`dataSource` — Data source

string

Access to the signal data values, stored as a string.

`dataValues` — Time series values

structure

Time series data for this signal, stored as a structure.

`fullBlockPath` — Block path
string

Block path to the signal with entire model hierarchy, stored as a string.

`id` — Signal identifier
integer

Unique number identifying the signal, stored as a string.

`modelSource` — Model source
string

Name of the model that defines the signal, stored as a string.

`portIndex` — Block port index
integer

Index of the block port that defines the signal, stored as an integer.

`rootSource` — Root source
string

Access to the high-level logging structure this signal was derived from, stored as a string.

`runID` — Run identifier
integer

Run ID, a unique number identifying the signal's parent run, stored as an integer.

`sampleDims` — Data sample dimensions
integer

Dimensions of a data sample, stored as an integer.

`sampleTime` — Sample time
string

Sample time of the signal derived from the model configuration settings, stored as a string.

`SID` — Simulink identifier
string

Simulink identifier of the block that defines the signal, stored as a string. The SID format is `model_name:sid_number`. For details, see "Locate Diagram Components Using Simulink Identifiers".

`signalLabel` — Signal name
string

Name of the signal, stored as a string.

`sourceType` — Signal source
`'logged' | 'visualized' | 'imported' | 'comparison'`

Source of the signal from the model or Simulation Data Inspector, stored as a string. Sources from the model are logged or visualized. Sources from the Simulation Data Inspector are imported or comparison.

`timeDim` — Time dimension
integer

For any given data sample, the time dimension, stored as an integer.

`timeSource` — Time source
string

Access to the logged signal's time vector, stored as a string.

`units` — Physical units
string

Units of the signal if you are using physical modeling, stored as a string

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 baseline signal.

`absTol` — Absolute tolerance
0 (default) | double

Absolute tolerance of the signal used for signal and run comparison, specified as a double. Must be a positive number.

`interpMethod` — Interpolation method

`'linear'` (default) | `'zoh'`

Interpolation method to align data. Possible values are zero-order hold, `'zoh'`, and `'linear'`.

`relTol` — Relative tolerance

0 (default) | double

Relative tolerance of the signal used for signal and run comparison, specified as a double. Must be a positive number.

`syncMethod` — Synchronization method

`'union'` (default) | `'uniform'` | `'intersection'`

Time synchronization method to align time vector when comparing signals, specified as a string. Possible values are `'intersection'`, `'uniform'`, or `'union'`.

Visualization Properties (read and write)

`checked` — Plotted indicator

boolean

Specifies if the signal is selected for plotting: `true` for selected and `false` for cleared.

`lineColor` — Line color

vector

Signal line color in the plot specified as a vector `[r g b]`. `r` is the red component, `g` the green component, and `b` the blue component.

`lineDashed` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'`

Signal line format in the plot, specified as a string. Possible values are `'-'` solid, `'--'` dashed, `':'` dotted, or `'-.'` dash-dot.

`marker` — Marker symbol style

string

Line marker style, specified as a string. Marker types and strings are the same as `LineSpec` in the `plot` function.

Examples

Modify Signal Data in a Run

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 IDs for signals contained in the run
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Get the signal object corresponding to the first signal ID
signalObj = Simulink.sdi.getSignal(signalIDs(1));

% signalObj is an instance of Simulink.sdi.Signal. Get the run ID for this signal
runID = signalObj.runID;

% Modify or define comparison and visualization properties for this signal
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;
```

- “Inspect and Compare Signal Data Programmatically”

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getSignal` | Simulink.sdi.Run

Introduced in R2012b

Simulink.Signal

Specify attributes of signal

Description

This class enables you to create workspace objects that you can use to assign or validate the attributes of a signal or discrete state, such as its data type, numeric type, dimensions, and so on. You can use a signal object to:

- Assign values to signal attributes that are left unassigned (have a value of `-1` or `auto`) by the signal source.
- Validate signal attributes whose values are explicitly assigned by the signal source. Such attributes have values other than `-1` or `auto`. Successful validation guarantees that the signal has the attributes that you intended it to have.

You can create a `Simulink.Signal` object in the MATLAB workspace or in a model workspace.

Use signal objects to assign or validate signal or discrete state attributes by giving the signal or discrete state the same name as the workspace variable that references the `Simulink.Signal` object.

Signal Specification Block: An Alternative to Simulink.Signal

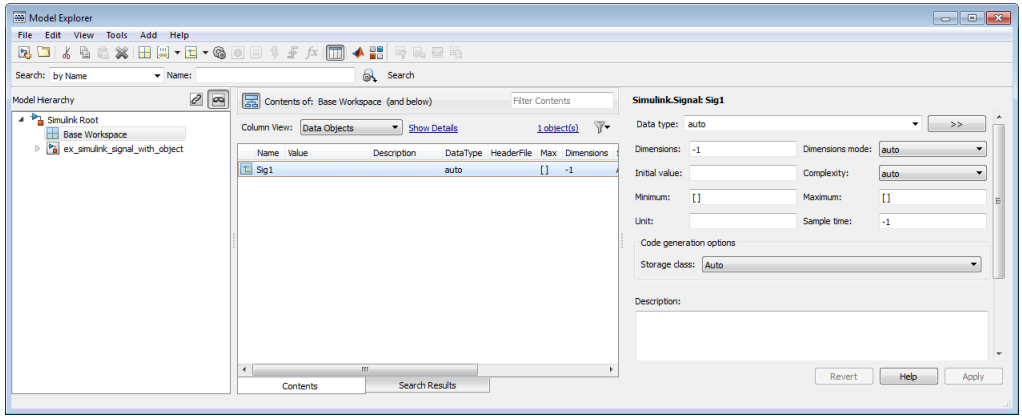
You can use a `Signal Specification` block rather than a `Simulink.Signal` object to assign properties left unspecified by a signal source. Each technique has advantages and disadvantages:

- 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`.



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.
   a. Open the Model Explorer.
   b. 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`.
   c. 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 5-369 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 "Use Signal Objects to Initialize Signals and Discrete States", "Using Signal Objects to Tune Initial Values", and "Control Data Representation by Applying 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.

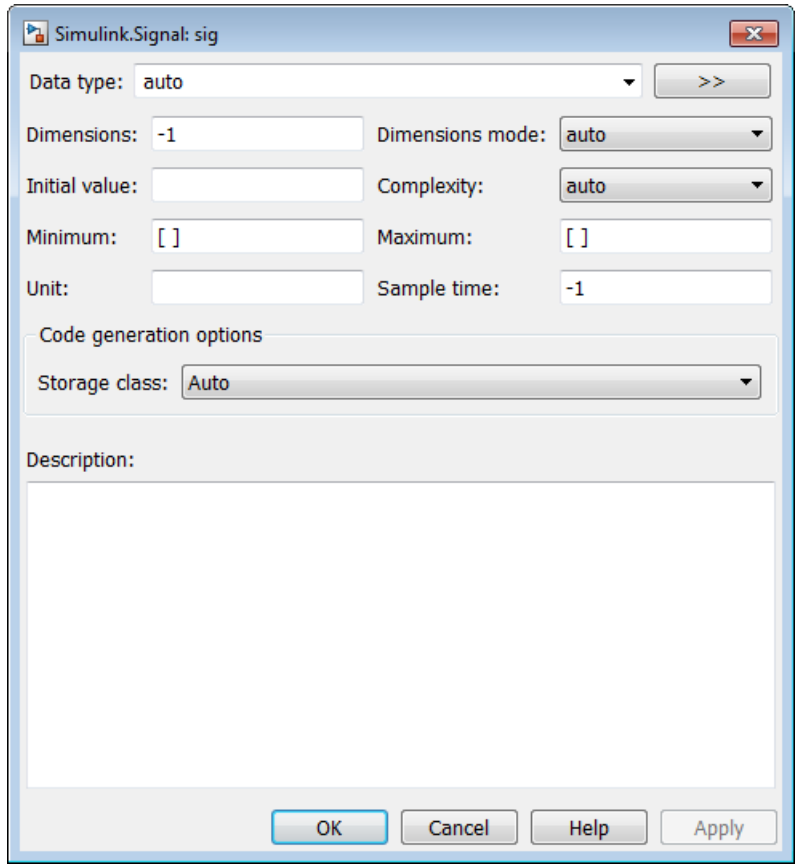
Multiple Signal Objects

You can associate a given *signal object* with more than one signal if the storage class of the signal object is `Auto` or `SimulinkGlobal`. If the storage class is `SimulinkGlobal`, or if it is `Auto` and you clear optimizations such as **Signal storage reuse** so that the generated code allocates memory for all of the associated signals, the signals each appear as a uniquely named field of the global structure that contains signal and state data. If the storage class of the object is other than `Auto` or `SimulinkGlobal`, 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



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 5-373 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.

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 expression, including the name of a workspace variable, that evaluates to a numeric scalar value or array.

You can use the MATLAB command prompt to provide an initial value for a signal. Even if you use a number, specify the initial value as a string.

```
mySigObject.InitialValue='5.3';

mySigObject.InitialValue = 'myNumericVariable';
```

To specify an initial value for a signal that uses a numeric data type other than `double`, cast the initial value to the signal data type. For example, you can specify `single(73.3)` to use `73.3` as the initial value for a signal of data type `single`.

If you use a bus object as the data type for the signal object, set **Initial value** to a string containing either `0` or a MATLAB structure that matches the bus object. See "Bus Support" on page 5-373 for details.

If the initial value evaluates to a MATLAB structure, then in the **Configuration Parameters** > **All Parameters** tab, set **"Underspecified initialization detection"** to `simplified`.

If necessary, Simulink converts the initial value to ensure type, complexity, and dimension consistency with the corresponding block parameter value. If you specify an invalid value or expression, an error message appears when you update the model. Also, Simulink performs range checking of the initial value. The software alerts you when the initial value of the signal lies outside a range that corresponds to its specified minimum and maximum values and data type.

Classic initialization mode: In this mode, initial value settings for signal objects that represent the following signals and states override the corresponding block parameter initial values if undefined (specified as `[]`):

- Output signals of conditionally executed subsystems and Merge blocks
- Block states

Simplified initialization mode: In this mode, initial values of signal objects associated with the following blocks are ignored. The initial values of the corresponding blocks are used instead.

- Outport blocks of conditionally executed subsystems
- Merge blocks

Unit

Physical unit in which the value of this signal is expressed, (for example, inches). To specify a unit, begin typing in the text box. As you type, the parameter displays potential unit string matches. For more information, see "Unit Specification in Simulink Models".

Storage class

Storage class of this signal. For more information, see "Storage Classes for Signals and States" in the Simulink Coder documentation and "Simulink Package Custom Storage Classes" in the Embedded Coder documentation.

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 "Data Alignment for Code Replacement". 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.

If you have an Embedded Coder license, you can add the signal description as a comment for the variable declaration in generated code.

- Specify a storage class for the signal object other than `Auto`.
- On the **Code Generation** > **Comments** pane of the Model Configuration Parameters dialog box, select the model configuration parameter **Simulink data object descriptions**. For more information, see "Simulink data object descriptions".

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.CoderInfo` 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)

Name	Access	Description
`Min`	`RW`	Minimum value that this signal can have. (Minimum)
`Max`	`RW`	Maximum value that this signal can have. (Maximum)
`Unit`	`RW`	Physical unit used for expressing this signal value. (Unit)
`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)
`InitialValue`	`RW`	Signal or state value before a simulation takes its first time step. (Initial Value)

More About

- “Signal Basics”
- “Data Objects”
- “Data Types Supported by Simulink”
- “MPT Data Object Properties”

See Also

`AUTOSAR.Signal` | `Simulink.CoderInfo` | `Simulink.Parameter`

Related Examples

- “Determine Where to Store Data for Simulink Models”
- “Control Signal Data Types”
- “Control Signals and States in Code by Applying Storage Classes”
- “Define Data Classes”

Introduced before R2006a

Simulink.SimulationData.BlockPath

Fully specified Simulink block path

Description

Simulink creates block path objects when creating dataset objects for signal logging and data store logging. `Simulink.SimulationData.Signal` and `Simulink.SimulationData.DataStoreMemory` objects include block path objects.

You can create a block path that you can use with the `Simulink.SimulationData.Dataset.getElement` method to access a specific dataset element. If you want to create a dataset in MATLAB to use as a baseline to compare against a signal logging or data store logging dataset, then you need to create the block paths as part of that dataset.

The `Simulink.SimulationData.BlockPath` class is very similar to the `Simulink.BlockPath` class.

You do not have to have Simulink installed to use the `Simulink.SimulationData.BlockPath` class. However, you must have Simulink installed to use the `Simulink.BlockPath` class. If you have Simulink installed, consider using `Simulink.BlockPath` instead of `Simulink.SimulationData.BlockPath`, because the `Simulink.BlockPath` class includes a method for checking the validity of block path objects without you having to update the model diagram.

Property Summary

Name	Description
`SubPath`	Individual component within the block specified by the block path

Method Summary

Name	Description
`BlockPath`	Create a block path.

Name	Description
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.

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
```

More About

- "Specify the Instance That Has Normal Mode Visibility"

See Also

`Simulink.BlockPath` | Simulink.SimulationData.Dataset

Introduced in R2012b

Simulink.SimulationData.Dataset class

Package: Simulink.SimulationData

Create Simulink.SimulationData.Dataset object

Description

Simulink creates `Simulink.SimulationData.Dataset` objects to store data elements when:

- Performing signal logging, which use the `Dataset` format
- Logging states or outputs, if you use the default format of `Dataset`.
- Logging a data store

Using the `Dataset` format for state and output logging offers several advantages compared to `Array`, `Structure`, or `Structure with time`. For details, see "Format for State Information Saved Without SimState".

`Simulink.SimulationData.Dataset` provides a `getElement` method for accessing individual elements in the data set. You can specify an element by index, name, or block path.

Construction

`convertedDataset = Simulink.SimulationData.Dataset(loggedDataToConvert)` converts the `loggedDataToConvert` to a `Simulink.SimulationData.Dataset` object. You can then use the Simulink.SimulationData.DataSet.concat method to combine elements of two `Dataset` objects.

`constructedDataset = Simulink.SimulationData.Dataset(variableName,'DatasetName','dsname')` constructs a `Simulink.SimulationData.Dataset` object, adds variable `variableName`, and names the data set `dsname`.

Input Arguments

`loggedDataToConvert` — Data element to convert

string

Data element to convert to a data set, specified as a string. You can convert elements such as:

- Array
- Structure
- Structure with time
- MATLAB time series
- Structure of MATLAB time-series elements
- `ModelDataLogs`

`variableName` — Variable to add to data set

string

Variable to add to data set, specified as a string.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DatasetName','dsname'`

`'DatasetName'` — Data set name

string

Data set name, specified as a string.

Output Arguments

`convertedDataset` — Converted data set

`Simulink.SimulationData.Dataset` object

Converted data set, returned as a `Simulink.SimulationData.Dataset` object.

`constructedDataset` — Constructed data set

`Simulink.SimulationData.Dataset` object

Constructed data set, returned as a `Simulink.SimulationData.Dataset` object.

Properties

`Name` — Name of the data set

same as the logging variable (default) | string

Name of the data set, specified as a string or logging variable (for example, `logsout` for signal logging). Specify a name when you want to distinguish easily one data set from another. For example, you could reset the name when comparing multiple simulations. This property is read/write.

```
ds = Simulink.SimulationData.Dataset
ds.Name = 'Dataset1'
```

`Total Elements` — Total number of elements

double

Total number of elements in data set, specified as a double. This property is read only. To get this value, use the Simulink.SimulationData.DataSet.numElements method.

Methods

addElement	Add element to end of data set
concat	Concatenate dataset to another dataset
get	Get element or collection of elements from dataset
getElementNames	Return names of all elements in dataset
find	Get element or collection of elements from dataset
numElements	Get number of elements in data set
setElement	Change element stored at specified index

Tip To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Concatenate Dataset ds1 to Dataset ds

Convert data from two To Workspace blocks, convert to `Dataset` format, and concatenate them. `myvdp` is the `vdp` model with two `To Workspace` blocks with variables named `simout` and `simout1`. These blocks log data in time-series format.

```
mdl = 'myvdp';
open_system(mdl);
sim(mdl)
ds = Simulink.SimulationData.Dataset(simout);
ds1 = Simulink.SimulationData.Dataset(simout1);
dsfinal = concat(ds,ds1)
```

- “Convert Logged Data to Dataset Format”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Migrate Scripts That Use ModelDataLogs API”
- “Access Logged Data from Persistent Storage”

See Also

Simulink.SimulationData.DatasetRef | Simulink.SimulationData.DataSet.concat | Simulink.SimulationData.DataSet.addElement | Simulink.SimulationData.DataSet.get | Simulink.SimulationData.DataSet.getElementNames | Simulink.SimulationData.DataSet.numElements | Simulink.SimulationData.DataSet.setElement | `Simulink.ModelDataLogs` | `Simulink.SimulationData.BlockPath` |

```
Simulink.SimulationData.DatasetRef.getDatasetVariableNames
| Simulink.SimulationData.DataStoreMemory |
Simulink.SimulationData.Signal
```

Introduced in R2011a

Simulink.SimulationData.DatasetRef class

Package: Simulink.SimulationData

Create `Simulink.SimulationData.DatasetRef` object

Description

To use a reference for accessing a `Simulink.SimulationData.Dataset` object stored in a MAT-file, create a `Simulink.SimulationData.DatasetRef` object. You can use this reference to avoid running out of memory, by retrieve data signal by signal, for data that you log to persistent storage.

Construction

`datasetRefObj = Simulink.SimulationData.DatasetRef( location,identifier)` creates a reference to the contents of a `Simulink.SimulationData.Dataset` variable stored in a MAT-file.

Input Arguments

`location` — MAT-file containing `Simulink.SimulationData.Dataset` object to reference

string

MAT-file containing `Simulink.SimulationData.Dataset` object to reference, specified as a string. The string is a path to the MAT-file. Do not use a file name from one locale in a different locale.

`identifier` — Name of variable in MAT-file

string

Name of a `Simulink.SimulationData.Dataset` variable in MAT-file, specified as a string. When you log to persistent storage, Simulink uses the variable names specified for each kind of logging.

Suppose that you use the default variable name for signal logging (`logsout`) and default MAT-file name for persistent storage logging (`mat.out`), After you simulate the model,

then to create a reference to the `Dataset` object for signal logging, at the MATLAB command line, enter:

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logsout');
```

Output Arguments

`datasetRefObj` — Reference to `Dataset` object

`Simulink.SimulationData.DatasetRef` object

Reference to logging dataset, returned as a `Simulink.SimulationData.DatasetRef` object.

Properties

`Location` — MAT-file containing `Simulink.SimulationData.Dataset` object to reference

string

MAT-file containing `Simulink.SimulationData.Dataset` object to reference, specified as a string. The string is a path to the MAT-file. Include the `.mat` extension in the file name. Do not use a file name from one locale in a different locale.

`Identifier` — Name of variable in MAT-file

string

Name of a `Simulink.SimulationData.Dataset` variable in MAT-file, specified as a string. When you log to persistent storage, Simulink uses the variable names specified for each kind of logging (for example, `'logsout'` for signal logging data).

Methods

Use the `numElements`, `getElement`, and `getElementNames` methods for a `Simulink.SimulationData.DatasetRef` object the same way that you use those methods for a `Simulink.SimulationData.Dataset` object.

Method	Purpose
numElements	Get number of elements from dataset

Method	Purpose
getElementNames	Return names of all elements in dataset
get The `get` method is an alias for the `getElement` method.	Get element from dataset

Tip To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

Copy Semantics

You can copy `DatasetRef` object properties by value. However, copying the `DatasetRef` object is a handle object. Copying the `DatasetRef` object does not copy the data in the MAT-file the object references. For details about copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create References to Persistent Storage Dataset Objects

This example shows how to construct and use `Simulink.SimulationData.DatasetRef` objects to access data for a model that logs to persistent storage. This simple example shows the basic steps for logging to persistent storage. This example does not represent a realistic situation for logging to persistent storage, because it shows a short simulation with small memory requirements.

Open the vdp model.

In the **Configuration Parameters** > **Data Import/Export** pane, select these parameters:

- **States**
- **Log Dataset data to file**

Set the **Format** parameter to `Dataset`.

Leave the other parameter settings as they are and click **Apply**.

In the model, right-click a signal and in the Signal Properties dialog box, select **Log signal data**.

Simulate the model.

Get a list of `Dataset` variable names in the `out.mat` file.

```
varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('out.mat')
```

```
varNames =

  'xout'
  'logsout'
```

Create a reference to the logged states data that is stored in `out.mat`. The variable for the logged states data is `xout`.

```
statesLogRef = Simulink.SimulationData.DatasetRef('out.mat.','xout')
```

```
statesLogRef =

  Simulink.SimulationData.DatasetRef
  Package: Simulink.SimulationData

  Characteristics:
          Location: out.mat. (H:\out.mat)
        Identifier: xout

  Resolved Dataset:
              Name: 'xout'
    Total Elements: 2

  Elements:
    1 : ''
    2 : ''

  Methods, Superclasses
```

Create a reference to the signal logging data that is stored in `out.mat`. The variable for the signal logging data is `logsout`.

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat.','logsout')
```

```
sigLogRef =

  Simulink.SimulationData.DatasetRef
  Package: Simulink.SimulationData

  Characteristics:
          Location: out.mat. (H:\out.mat)
        Identifier: logsout

  Resolved Dataset:
              Name: 'logsout'
    Total Elements: 1

  Elements:
    1 : 'x1'

  Methods, Superclasses
```

Use the `DatasetRef` to access the number of elements in the logged states dataset.

```
statesLogRef.numElements
```

```
ans =

     2
```

Use the `DatasetRef` to access the first element of the signal logging dataset.

```
sigLogRef.get(1)
```

```
ans =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
              Name: 'x1'
    PropagatedName: ''
         BlockPath: [1x1 Simulink.SimulationData.BlockPath]
          PortType: 'outport'
         PortIndex: 1
            Values: [1x1 timeseries]


  Methods, Superclasses
```

Delete the persistent storage MAT-file and try to use one of the `DatasetRef` objects.

```
delete('out.mat');
statesLogRef.get(1)
```

```
File does not exist.
```

The `statesLogRef` still exists, but it is a reference to a `Dataset` object that is in a file that no longer exists.

- "Log Data Using Persistent Storage"
- "Access Logged Data from Persistent Storage"
- "Convert Logged Data to Dataset Format"

See Also

Simulink.SimulationData.Dataset |
`Simulink.SimulationData.DatasetRef.getDatasetVariableNames`

Introduced in R2016a

Simulink.SimulationData.DataStoreMemory

Container for data store logging information

Description

Simulink uses `Simulink.SimulationData.DataStoreMemory` objects to store logging information from Data Store Memory blocks during simulation. The objects contain information about the blocks that write to the data store.

Property Summary

Name	Description
`BlockPath`	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

More About

- "Log Data Stores"

See Also

Simulink.SimulationData.Dataset | `Simulink.SimulationData.BlockPath` | `Data Store Memory` | `Data Store Write`

Simulink.SimulationData.LoggingInfo

Signal logging override settings

Description

This object specifies a set of signal logging override settings.

Use a `Simulink.SimulationData.LoggingInfo` object to specify the signal logging override settings for a signal. You can use this object for the `LoggingInfo` property of a `Simulink.SimulationData.SignalLoggingInfo` object.

Property Summary

Name	Description
`DataLogging`	Signal logging mode.
`NameMode`	Source of signal logging name.
`LoggingName`	Custom signal logging name.
`DecimateData`	Use subset of sample points.
`Decimation`	Decimation value (`n`): Simulink logs every `n`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. The first sample point is always logged.

Data Type

logical value — `true | {false}`

Access

RW

Decimation

Description

Decimation value (`n`). If the `DecimateData` property is `true`, then Simulink logs every `n`th data point.

Data Type

positive 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

positive 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`.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_mdlref_cou
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_bus_loggin
log_info = Simulink.SimulationData.LoggingInfo();
log_info.DecimateData = true;
log_info.Decimation = 2;
mi = Simulink.SimulationData.SignalLoggingInfo('ex_bus_logging');
mi.LoggingInfo = log_info
```

```
Simulink.SimulationData.SignalLoggingInfo
  Package: Simulink.SimulationData

  BlockPath:
    'ex_bus_logging'
```

```
OutputPortIndex: 1

LoggingInfo:
      DataLogging: 1
         NameMode: 0
      LoggingName: ''
     DecimateData: 1
       Decimation: 2
  LimitDataPoints: 0
        MaxPoints: 5000
```

More About

- "Override Signal Logging Settings from MATLAB"
- "Migrate Scripts That Use ModelDataLogs API"
- "Export Signal Data Using Signal Logging"
- "Log Data Stores"

See Also

`Simulink.SimulationData.ModelLoggingInfo` | `Simulink.SimulationData.SignalLoggingInfo` | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.ModelDataLogs`

Introduced in R2012b

Simulink.SimulationData.ModelLoggingInfo

Signal logging override settings for a model

Description

This class is a collection of `Simulink.SimulationData.SignalLoggingInfo` objects that specify all signal logging override settings for a model.

Use methods and properties of this class to:

- Turn off logging for a signal or a Model block.
- Change logging settings for any signals that are already marked for logging within a model.

You can control whether a top-level model and referenced models use override signal logging settings or use the signal logging settings specified by the model. Use the `LoggingMode` and `LogAsSpecifiedByModels` properties to control which logging settings to apply.

Logging Mode for Models	Property Settings
For top-level model and all referenced models, use logging settings specified in the model.	Set `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 “Override Signal Logging Settings from MATLAB”.

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')


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)
```

```
signal_index =

     1
```

To find a signal in a specific instance of a referenced model that is not in a Stateflow chart, use the following approach:

```
signal_index = mi.findSignal({'ex_bus_logging/CounterA', ...
'ex_mdlref_counter_bus/Bus Creator'})

signal_index =

     4
```

For an example that uses the `findSignal` method with a Stateflow chart, see "Override Logging Properties with the Command-Line API" 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';
mi = 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')
```

More About

- "Override Signal Logging Settings from MATLAB"
- "Migrate Scripts That Use ModelDataLogs API"
- "Export Signal Data Using Signal Logging"
- "Log Data Stores"

See Also

`Simulink.BlockPath` | `Simulink.SimulationData.LoggingInfo` | `Simulink.SimulationData.SignalLoggingInfo` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.ModelDataLogs`

Introduced in R2012b

Simulink.SimulationData.SignalLoggingInfo

Signal logging override settings for signal

Description

This object contains the signal override signal logging settings for a single logged signal.

Property Summary

Name	Description
`BlockPath`	`Simulink.BlockPath` of source block of a signal to log.
`OutputPortIndex`	Index of an output 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

OutputPortIndex

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
```

More About

- “Override Signal Logging Settings from MATLAB”
- “Migrate Scripts That Use ModelDataLogs API”

- “Export Signal Data Using Signal Logging”
- “Log Data Stores”

See Also

`Simulink.SimulationData.ModelLoggingInfo` | `Simulink.SimulationData.LoggingInfo` | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.ModelDataLogs`

Introduced in R2012b

Simulink.SimulationData.Signal

Container for signal logging information

Description

Simulink uses `Simulink.SimulationData.Signal` objects to store signal logging information during simulation. The objects contain information about the source block for the signal, including the port type and index.

Property Summary

Name	Description
`BlockPath`	Block path for the source block for the signal
`Name`	Name of signal element to use for name-based access
`PropagatedName`	Propagated signal name, if any
`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

PropagatedName

Description

Propagated name of signal element

Signal logging and root Outport block logging data for a signal captures the propagated signal name if the logging format is `Dataset` and:

- For signal logging, you:
  - Mark the signal for signal logging and in the Signal Properties dialog box select **Show Propagated Signals**.
  - Enable **Configuration Parameters > Data Import/Export > Signal logging**.
- For root Outport block logging, you select **Configuration Parameters > Data Import/Export > Output**.

The propagated signal name does not include angle brackets (<>).

Data Type

string

Access

RO

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

More About

- “Access Signal Logging Data”
- “Create MATLAB Timeseries Data for Root Inports”
- “Import Bus Data to Top-Level Input Ports”

See Also

`Simulink.BlockPath` | Simulink.SimulationData.Dataset | timeseries

Simulink.SimulationData.State class

Package: Simulink.SimulationData

State logging element

Description

Simulink uses `Simulink.SimulationData.State` objects to store state logging information during simulation. The objects contain state information about which block the state data is coming from and the type of state.

Properties

`Name` — Name of state element to use for name-based access

string

Name of state element to use for name-based access, specified as a string. If you do not specify a name, `'CSTATE'` or `'DSTATE'` is used, depending on whether it a continuous or discrete state.

`BlockPath` — Block path for state source block

a `Simulink.SimulationData.BlockPath` object

Block path for state source block, specified as a `Simulink.SimulationData.BlockPath` object

`Label` — Type of state

`'CSTATE' | 'DSTATE'`

Type of state, returned as `'CSTATE'` or `'DSTATE'`. Read-only property.

- `'CSTATE'` – Continuous state
- `'DSTATE'` – Discrete state

`Values` — State element information

single MATLAB `timeseries` object | a structure of MATLAB `timeseries` objects

State element information, specified as a single MATLAB `timeseries` object or as a structure of MATLAB `timeseries` objects.

Examples

Final State Information in Structure with Dataset Format

Saved final state information in `Dataset` format and access the state data after simulation.

Open the `vdp` model and specify to log final states in `Dataset` format. Use the default logged state variable, `xFinal`.

```
open_system('vdp');
set_param(gcs,'SaveFinalState','on','SaveFormat','Dataset');
```

Simulate the model.

```
sim('vdp');
```

View the state logging information in `xFinal`.

```
xFinal
```

```
xFinal =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'xFinal'
    Total Elements: 2

  Elements:
    1: 'CSTATE'
    2: 'DSTATE'
```

Examine the first element of the state dataset.

```
xFinal.get(1)
```

```
ans =
```

```
Simulink.SimulationData.State
Package: Simulink.SimulationData

Properties:
       Name: 'STATE'
  BlockPath: [1x1 Simulink.SimulationData.BlockPath]
      Label: CSTATE
     Values: [1x1 timeseries]
```

See Also

Simulink.SimulationData.Dataset

More About

- "State Information"

Introduced in R2015a

Simulink.SimulationData.Unit class

Package: Simulink.SimulationData

Store units for simulation data

Description

Simulink creates `Simulink.SimulationData.Unit` objects to store unit information for signals when:

- Performing signal logging, which uses the `Dataset` format
- Logging root Outport blocks, if in **Configurations Parameters** > **All Parameters** you select the **Output** parameter and set the **Format** parameter to `Dataset`
- Logging to a To Workspace block or To File block, if you set the **Save format** block parameter to the default of `Timeseries`

Construction

`unitsObj = Simulink.SimulationData.Unit(unitName)` creates a `Simulink.SimulationData.Unit` object with the unit that you specify.

Input Arguments

`unitName` — Name of logging data units

string

Name of logging data units, specified as a string.

Output Arguments

`unitObj` — Logging data units

`Simulink.SimulationData.Unit` object

Logging data units, returned as a `Simulink.SimulationData.Unit` object.

Properties

`Name` — Name of the units

string

Name of the units, specified as a string.

Methods

Method	Purpose
Simulink.SimulationData.Unit.se	Specify name of logging data unit

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create and Use Inches Logging Units

Create a `Simulink.SimulationData.Unit` object representing inches.

```
inchesUnit = Simulink.SimulationData.Unit('inches');
```

Create a MATLAB `timeseries` object and set its `Units` field to the `Simulink.SimulationData.Unit` object.

```
ts = timeseries(1:10);
ts.DataInfo.Units = inchesUnit;
ts.DataInfo.Units
```

```
ans =

  Units with properties:
```

```
    Name: 'inches'
```

- “Log Signal Data That Uses Units”
- “Load Signal Data That Uses Units”
- “Convert Logged Data to Dataset Format”
- “Prepare Model Inputs and Outputs”

See Also

Simulink.SimulationData.Dataset

Introduced in R2016a

Simulink.SimulationMetadata class

Package: Simulink

Access metadata of simulation runs

Description

The `SimulationMetadata` class contains information about a simulation run including:

- Model information
- Timing information
- Execution and diagnostic information
- Custom string to tag the simulation
- Custom data to describe the simulation

`SimulationMetadata` packages this information with the `SimulationOutput` object. To use `SimulationMetadata`, use one of these approaches:

- In **Configuration Parameters** > **Data Import/Export**, under **Save options**, select **Single simulation output**.
- Use `set_param` to set `ReturnWorkspaceOutputs` to `on`.

```
set_param(model_name,'ReturnWorkspaceOutputs','on');
```

To retrieve the `SimulationMetadata` object, use the `getSimulationMetadata` method on a `SimulationOutput` object.

Properties

`ModelInfo` — Information about the model and simulation operating environment

structure

The `ModelInfo` structure has these fields.

Field Name	Type	Description
`ModelName`	char	Name of the model
`ModelVersion`	char	Version of the model

Field Name	Type	Description
`ModelFilePath`	char	Absolute location of the `.mdl`/`.slx` file
`UserID`	char	System user ID of the machine used for the simulation
`MachineName`	char	Hostname of the machine used for the simulation
`Platform`	char	Operating system of the machine used for the simulation
`ModelStructuralChecksu`	4–by–1 uint32	Structural checksum of the model calculated after an update diagram
`SimulationMode`	char	Simulation mode
`StartTime`	double	Simulation start time
`StopTime`	double	Time at which the simulation was terminated
`SolverInfo`	struct	Solver information: • Fixed-step solvers – Solver type, name, and fixed step size • Variable solvers – Solver type, name, and max step size (initial setting)
`SimulinkVersion`	struct	Version of Simulink

`ExecutionInfo` — Structure to store information about a simulation run
structure

Structure to store information about a simulation run, including the reason a simulation stopped and any diagnostics reported during the simulation. The structure has these fields.

Field Name	Type	Description
`StopEvent`	Nontranslated string	Reason the simulation stopped, represented by one of these strings. • `ReachedStopTime` – Simulation stopped upon reaching stop time and no errors were reported during

Field Name	Type	Description
		execution. `StopEvent` has value `ReachedStopTime`, even if errors are reported in the stop callbacks, which are executed after the simulation ends. • `ModelStop` – Simulation stopped by a block or by solver before reaching stop time • `StopCommand` – Simulation stopped manually by clicking the **Stop** button or programmatically using the `set_param` command • `DiagnosticError` – Simulation stopped because an error was reported during simulation • `KeyboardControlC` – Simulation stopped using keystroke `Ctrl+C`. • `PauseCommand` – Simulation paused manually by clicking the **Pause** button or programmatically using the `set_param` command • `ConditionalPause` – Simulation paused using a conditional breakpoint • `PauseTime` – Simulation paused at or after specified pause time • `StepForward` – Simulation paused after clicking step forward • `StepBackward` – Simulation paused after clicking step backward • `TimeOut` – Simulation stopped because execution time exceeded timeout specified by `TimeOut`
`StopEventSource`	`Simulink.Simu`	Source of stop event, if it is a valid Simulink object

Field Name	Type	Description
`StopEventDescription`	Translated string	Super set of information stored in `StopEvent` and `StopEventSource`
`ErrorDiagnostic`	struct	Error reported during simulation, represented by these fields. • `Diagnostic` – `MSLDiagnostic` object that includes object paths, ID, message, cause, and stack • `SimulationPhase` – Represented by one of these strings: `Initialization`, `Execution`, or `Termination` • `SimulationTime` – Simulation time represented as a double, if reported during Execution; else, represented as `[]` By passing the name–value pair `'CaptureErrors', 'on'`, the `sim` command does not display errors, instead populating the left-hand-side argument. This option is not available for simulation in SIL and PIL modes.

Field Name	Type	Description
`WarningDiagnostics`	Array of struct	Array of all warnings reported during the simulation. Each array item is represented by these fields. • `Diagnostic` – `MSLDiagnostic` object that includes object paths, ID, message, cause, and stack • `SimulationPhase` – Represented by one of these strings: `Initialization`, `Execution`, or `Termination` • `SimulationTime` – Simulation time represented as a double, if reported during `Execution`; else, represented as `[]`

`TimingInfo` — Structure to store profiling information about the simulation
structure

Structure to store profiling information about the simulation, including the time stamps for the start and end of the simulation. The structure has these fields.

Field Name	Type	Description
`WallClockTimestampStar`	string	Wall clock time when the simulation started, in `YYYY-MM-DD HH:MI:SS` format with microsecond resolution
`WallClockTimestampStop`	string	Wall clock time when the simulation stopped, in `YYYY-MM-DD HH:MI:SS` format with microsecond resolution
`InitializationElapsedW`	double	Time spent before execution, in seconds
`ExecutionElapsedWallTi`	double	Time spent during execution, in seconds
`TerminationElapsedWall`	double	Time spent after execution in, seconds
`TotalElapsedWallTime`	double	Total time spent in initialization, execution, and termination, in seconds

The `ExecutionElapsedWallTime` includes the time that Simulink spent to roll back or step back in a simulation. The `ExecutionElapsedWallTime` does not include the time spent between steps. For example, if you use Stepper to step through a simulation, the `ExecutionElapsedWallTime` time does not include the time when the simulation is in a paused state. For more information about using Stepper, see "How Simulation Stepper Helps With Model Analysis".

`UserString` — Custom string to describe the simulation

string

Use `Simulink.SimulationOutput.setUserString` to directly store a string in the `SimulationMetadata` object that is contained in the `SimulationOutput` object.

`UserData` — Custom data to store in `SimulationMetadata` object that is contained in the `SimulationOutput` object

string

Use `Simulink.SimulationOutput.setUserData` to store custom data in the `SimulationMetadata` object that is contained in the `SimulationOutput` object.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Get a `SimulationMetadata` Object for `vdp` Simulation

Simulate the `vdp` model. Retrieve metadata from a `SimulationMetadata` object of the simulation.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.

```
open_system('vdp');
simout = sim(bdroot,'ReturnWorkspaceOutputs','on');
```

Retrieve metadata information about this simulation using `mData`. This is the `SimulationMetadata` object that `simout` contains.

```
mData=simout.getSimulationMetadata()
```

```
SimulationMetadata with properties:
     ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
  ExecutionInfo: [1x1 struct]
    UserString: ''
      UserData: []
```

Store custom data or string in `simout`.

```
simout=simout.setUserData(struct('param1','value1','param2','value2','param3','value3
simout=simout.setUserString('Store first simulation results');
```

Retrieve the custom data you stored from `mData`.

```
mData=simout.getSimulationMetadata()
disp(mData.UserData)
```

```
    param1: 'value1'
    param2: 'value2'
    param3: 'value3'
```

Retrieve the custom string you stored from `mData`.

```
mData=simout.getSimulationMetadata()
disp(mData.UserString)
```

```
    Store first simulation results
```

See Also

Simulink.SimulationOutput.getSimulationMetadata | Simulink.SimulationOutput.setUserData | Simulink.SimulationOutput.setUserString

Introduced in R2015a

Simulink.SimulationOutput class

Package: Simulink

Access object values of simulation results

Description

The `SimulationOutput` class contains all simulation outputs, including workspace variables.

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

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

- "Migrate Scripts That Use ModelDataLogs API"
- "Export Simulation Data"

Simulink.SubsysDataLogs

Container for subsystem signal data logs

Description

Note: Before R2016a, the `Simulink.SubsysDataLogs` class was used in conjunction with the `ModelDataLogs` logging data format. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format.

However, you can use data that was logged in a previous release using `ModelDataLogs` format.

In releases before R2016a, Simulink created instances of this class to contain logs for signals in a subsystem models were logged in `ModelDataLogs` format. Objects of this class have a variable number of properties. The first property, named `Name`, is the name of the subsystem whose log data this object contains. The remaining properties are signal log or signal log container objects containing the data logged for the subsystem specified by this object's `Name` property.

For example, suppose you have this logged data from a model run in a release earlier than R2016a:

```
Simulink.SubsysDataLogs (Gain):
  Name                    elements  Simulink Class

  a                          1      Timeseries
  m                          2      TsArray
```

You can use either fully qualified log names or the `unpack` command to access the signal logs contained by a `SubsysDataLogs` object. For example, to access the amplitudes logged for signal `a` in the preceding example, you could enter the following at the MATLAB command line:

```
data = logsout.Gain.a.Data;
```

or

```
>> logsout.unpack('all');
```

```
data = a.Data;
```

See Also

"Create Signal Data to Load", `Simulink.ModelDataLogs`, `Simulink.Timeseries`, `Simulink.TsArray`, Simulink.SimulationData.Dataset, `who`, `whos`, `unpack`

Introduced before R2006a

Simulink.TimeInfo

Provide information about time data in `Simulink.Timeseries` object

Description

Simulink software creates instances of these objects to describe the time data that it includes in `Simulink.Timeseries` objects.

Note: The `Simulink.Timeseries` class is supported for backwards compatibility. The `ModelDataLogs` format created `Simulink.Timeseries` objects for signal logging data. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

Properties

Name	Access	Description
`Units`	`RW`	The units, e.g., `'seconds'`, in which the time series data are expressed in the associated `Simulink.Timeseries` object.
`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

Name	Access	Description
		conditionally executed subsystem, this field contains an array of times when the system became active.
`end`	`RW`	If the associated signal is not in a conditionally executed subsystem, this field contains the simulation time of the last signal value recorded in the associated `Simulink.Timeseries` object. If the signal is in a conditionally executed subsystem, this field contains an array of times when the system became inactive.
`Increment`	`RW`	The interval between simulation times at which signal data is logged in the associated `Simulink.Timeseries` object. If the signal is aperiodic (continuous signal with variable-step solver), this property has a value of `NaN`. A signal is periodic if it has a discrete sample time (not continuous or constant) or is continuous with a fixed-step solver.
`Length`	`W`	The number of signal samples recorded in the associated `Simulink.Timeseries` object, i.e., the length of the arrays referenced by the object's `Time` and `Data` properties.

See Also

`Simulink.Timeseries` , Simulink.SimulationData.Dataset

Introduced before R2006a

Simulink.Timeseries

Store data for any signal except mux or bus signal

Description

Note: The `Simulink.Timeseries` class is supported for backwards compatibility. The `ModelDataLogs` format created `Simulink.Timeseries` objects for signal logging data. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see "Convert Logged Data to Dataset Format".

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See "Migrate Scripts That Use ModelDataLogs API".

Simulink software creates instances of this class to store signal data that it logs for any signal except a mux or bus signal, which are stored in a `Simulink.TsArray`. See "Export Signal Data Using Signal Logging" for more information.

Properties

Name	Access	Description
`Name`	`RW`	Name of this signal log.
`BlockPath`	`RW`	Path of the block that output the signal logged in this signal log.
`PortIndex`	`RW`	Index of the output port that emitted the signal logged in this signal log.

Name	Access	Description
`SignalName`	`RW`	Name of the signal logged in this signal log.
`ParentName`	`RW`	Name of the parent of the signal recorded in this log, if the signal is an element of a mux or a virtual bus; otherwise, the same as `SignalName`.
`TimeInfo`	`RW`	An object of `Simulink.TimeInfo` class that describes the time data in this log.
`Time`	`RW`	An array containing the simulation times at which signal data was logged.
`Data`	`RW`	An array containing the signal data.

See Also

"Export Signal Data Using Signal Logging", `Simulink.TimeInfo`, Simulink.SimulationData.Dataset, `Simulink.ModelDataLogs`, `Simulink.SubsysDataLogs`, `Simulink.TsArray`, `who`, `whos`, `unpack`

Introduced before R2006a

Simulink.TsArray

Store data for mux or bus signal

Description

Note: Before R2016a, the `Simulink.TsArray` class was used in conjunction with the `ModelDataLogs` logging data format. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format.

However, you can use data that was logged in a previous release using `ModelDataLogs` format.

In releases earlier than R2016a, Simulink software created instances of this class to contain the data that it logs for a mux or bus signal. Other types of signals were stored in a `Simulink.Timeseries`.

Objects of the `Simulink.TsArray` class have a variable number of properties. The first property, called `Name`, specifies the log name of the logged signal. The remaining properties reference logs for the elements of the logged signal: `Simulink.Timeseries` objects for elementary signals and `Simulink.TSArray` objects for mux or bus signals. The name of each property is the log name of the corresponding signal.

For example, suppose you have this logged data from a model run in a release earlier than R2016a that was configured to log in `ModelDataLogs` format.

```
logsout.b2
```

```
Simulink.TsArray (untitled/Bus Creator1):
  Name                    elements   Simulink Class

  x1                         1       Timeseries
  b1                         2       TsArray
```

The `Simulink.ModelDataLogs` object, named `logsout`, contains a `Simulink.TsArray` object, named `b2`, that contains the logs for the elements of `b2` (that is, the elementary signal `x1` and the bus signal `b1`). Entering the fully qualified name of

the `Simulink.TsArray` object, (`logsout.b2`) at the MATLAB command line reveals the structure of the signal log for this model.

You can use either fully qualified log names or the `unpack` command to access the signal logs contained by a `Simulink.TsArray` object. For example, to access the amplitudes logged for signal `x1` in the preceding example, you can enter the following at the MATLAB command line:

```
data = logsout.b2.x1.Data;
```

or

```
logsout.unpack('all');
data = x1.Data;
```

See Also

`Simulink.ModelDataLogs`, `Simulink.SubsysDataLogs`, `Simulink.Timeseries`, Simulink.SimulationData.Dataset, `who`, `whos`, `unpack`

Introduced before R2006a

Simulink.Variant class

Package: Simulink

Specify conditions that control variant selection

Description

An object of the `Simulink.Variant` class represents a conditional expression called a variant control. The object allows you to specify a Boolean expression that activates a specific variant choice when it evaluates to `true`.

A variant control comprises one or more variant control variables, specified using MATLAB variables or `Simulink.Parameter` objects.

You specify variant controls for each variant choice represented in a `Variant Subsystem` or `Model Variant` block. For a given `Variant Subsystem` or `Model Variant` block, only one variant control can evaluate to `true` at a time. When a variant control evaluates to `true`, Simulink activates the variant choice that corresponds to that variant control.

Construction

`variantControl = Simulink.Variant(conditionExpression)` creates a variant control.

Properties

`conditionExpression` — Variant condition expression

`''` (default) | string

Variant condition expression, specified as a string containing one or more of these operands and operators.

Operands

- Variable names that resolve to MATLAB variables or `Simulink.Parameter` objects with integer or enumerated data type and scalar literal values

- Variable names that resolve to `Simulink.Variant` objects
- Scalar literal values that represent integer or enumerated values

Operators

- Parentheses for grouping
- Arithmetic, relational, logical, or bit-wise operators

The variant condition expression evaluates to a Boolean value. This property has read and write access.

Example: `'(Fuel==2 || Emission==1) && Ratio==2'`

Examples

Create Variant Controls Using MATLAB Variables

Use MATLAB variables when you want to simulate the model but are not considering code generation.

Create MATLAB variables with scalar literal values.

```
Fuel = 3;
Emission = 1;
Ratio = 3;
```

Develop conditional expressions using the variables.

```
Variant1 = Simulink.Variant('Fuel==1 && Emission==2');
Variant2 = Simulink.Variant('(Fuel==2 || Emission==1) && Ratio==2');
Variant3 = Simulink.Variant('Fuel==3 || Ratio==4');
```

Create Variant Controls Using `Simulink.Parameter` Objects

If you want to generate preprocessor conditionals for code generation, use `Simulink.Parameter` objects instead of MATLAB variables.

Create variant `Simulink.Parameter` objects with scalar literal values.

```
Fuel = Simulink.Parameter(3);
```

```
Emission = Simulink.Parameter(1);
Ratio = Simulink.Parameter(3);
```

Specify the custom storage class for these objects as `ImportedDefine` so that the values are specified by an external header file.

Other valid values for the custom storage class are `Define` and `CompilerFlag`.

```
Fuel.CoderInfo.StorageClass = 'Custom';
Fuel.CoderInfo.CustomStorageClass = 'ImportedDefine';

Emission.CoderInfo.StorageClass = 'Custom';
Emission.CoderInfo.CustomStorageClass = 'ImportedDefine';

Ratio.CoderInfo.StorageClass = 'Custom';
Ratio.CoderInfo.CustomStorageClass = 'ImportedDefine';
```

Develop conditional expressions using the variables and create variant controls.

```
Variant1 = Simulink.Variant('Fuel==1 && Emission==2');
Variant2 = Simulink.Variant('(Fuel==2 || Emission==1) && Ratio==2');
Variant3 = Simulink.Variant('Fuel==3 || Ratio==4');
```

See Also

"Operators and Operands in Variant Condition Expressions"

More About

- "Approaches for Specifying Variant Controls"

Simulink.VariantConfigurationData class

Package: Simulink

Class representing a variant configurations data object

Description

The variant configuration data object, stores a collection of variant configurations, constraints, and the name of the default active configuration. The `Simulink.VariantConfigurationData` class has properties that enable you to add, modify, or remove variant configurations, constraints, and control variables. Use an instance of `Simulink.VariantConfigurationData` class to do the following:

- Define and edit variant configurations.
- Add control variables to variant configurations.
- Add copy of variant configuration.
- Delete existing variant configurations, constraints, and sub model configurations.
- Set a specific configuration as default active.
- Validate model using default or a specific variant configuration.
- Query or create variant configurations data object for a given model.

Properties

`VariantConfigurations`

Set of variant configurations. The names of the configurations must be unique and valid MATLAB variable names.

`Constraints`

Set of constraints that must always be satisfied by the model for all variant configurations. The name of the constraints must be unique and valid MATLAB variable names.

`DefaultConfigurationName`

Name of the variant configuration to be used by default for validation.

Methods

addConfiguration	Add a new variant configuration to the variant configuration data object
addConstraint	Add a constraint to the variant configuration data object
addControlVariables	Add control variables to an existing variant configuration
addCopyOfConfiguration	Add a copy of an existing variant configuration to the variant configuration data object
addSubModelConfigurations	Add to a variant configuration the names of the configurations to be used for submodels
existsFor	Check if variant configuration data object exists for a model
getConfiguration	Returns the variant configuration with a given name from a variant configuration data object
getDefaultConfiguration	Returns default variant configuration, if any, for a variant configuration data object
getFor	Get existing variant configuration data object for a model
getOrCreateFor	Get existing or create a new variant configuration data object for a model
removeConfiguration	Remove a variant configuration with a given name from the variant configuration data object
removeConstraint	Remove a constraint from the variant configuration data object
removeControlVariable	Remove a control variable from a variant configuration

removeSubModelConfiguration	Remove from a variant configuration, the configuration to be used for a sub model.
setDefaultConfigurationName	Set name of the default variant configuration for a variant configuration data object
validateModel	Validate all variant blocks in the model and submodels in the hierarchy during simulation
VariantConfigurationData	Object constructor with optional arguments for variant configurations, constraints, and default configuration name

More About

- "Variant Management"

addConfiguration

Class: Simulink.VariantConfigurationData
Package: Simulink

Add a new variant configuration to the variant configuration data object

Syntax

```
vcdataObj.addConfiguration(name)
vcdataObj.addConfiguration(name,description)
vcdataObj.addConfiguration(name,description,controlVars)
vcdataObj.addConfiguration(name,description,controlVars,
subModelConfigurations)
```

Description

`vcdataObj.addConfiguration(name)` adds a new variant configuration with a given name to the variant configuration data object.

`vcdataObj.addConfiguration(name,description)` adds a new variant configuration with a given name and optional description to the variant configuration data object.

`vcdataObj.addConfiguration(name,description,controlVars)` adds a new variant configuration with a given name, optional description, and control variables to the variant configuration data object.

`vcdataObj.addConfiguration(name,description,controlVars, subModelConfigurations)` adds a new variant configuration with a given name, optional description, control variables, and submodel configurations to the variant configuration data object.

Input Arguments

`name`

Name of variant configuration being added.

`description`

Description text for the variant configuration being added.

`controlVars`

Control variables for the variant configuration being added. This argument must be a vector of structures with required fields: `Name` and `Value`. The values assigned to the `Name` field must be unique and valid MATLAB variable names. The `Value` field can contain either strings or `Simulink.Parameter` objects. The values of control variables are checked during validation of the variant configuration.

`subModelConfigurations`

Vector of structures containing fields: `ModelName`, `ConfigurationName`. The names of submodels must be unique and valid MATLAB variable names and configuration names must be valid MATLAB variables.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addControlVariables |
Simulink.VariantConfigurationData.addSubModelConfigurations

addConstraint

Class: Simulink.VariantConfigurationData
Package: Simulink

Add a constraint to the variant configuration data object

Syntax

```
vcdataObj.addConstraint(nameOfConstraint)
vcdataObj.addConstraint(nameOfConstraint,condition)
vcdataObj.addConstraint(nameOfConstraint,condition,description)
```

Description

`vcdataObj.addConstraint(nameOfConstraint)` adds a new constraint with a given name to `vcdataObj`.

`vcdataObj.addConstraint(nameOfConstraint,condition)` adds a new constraint with a given name and condition expression to `vcdataObj`.

`vcdataObj.addConstraint(nameOfConstraint,condition,description)` adds a new constraint with a given name, condition expression, and description to `vcdataObj`.

Input Arguments

`nameOfConstraint`

Name of constraint being added. Must be unique and valid MATLAB variable name.

`condition`

Boolean expression that must evaluate to true. When the expression evaluates to true, it means the constraint is satisfied.

`description`

Text that describes the constraint.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a constraint named LinNotExtern
vcdataObj.addConstraint('LinNotExtern','((Ctrl~=1)...
  || (PlantLocation ~=1))','Description of the constraint')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.removeConstraint |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.removeConfiguration

addControlVariables

Class: Simulink.VariantConfigurationData
Package: Simulink

Add control variables to an existing variant configuration

Syntax

```
vcdataObj.addControlVariables(nameOfConfiguration,controlVars)
```

Description

`vcdataObj.addControlVariables(nameOfConfiguration,controlVars)`, adds control variables to a variant configuration.

Input Arguments

`nameOfConfiguration`

Specifies the name of an existing configuration.

`controlVars`

Control variables being added. This argument must be a vector of structures with required fields: `Name` and `Value`. The values assigned to the `Name` field must be unique and valid MATLAB variable names. The `Value` field can contain either strings or `Simulink.Parameter` objects. The values of control variables are checked during validation of the variant configuration.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;
```

```
% Add a variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
'Linear Internal Experimental Plant Controller');

% Add control variables SmartSensor1Mod and PlanLocation
vcdataObj.addControlVariables('LinInterExp',...
  cell2struct({'SmartSensor1Mod', '2';...
               'PlantLocation', '1'},...
              {'Name', 'Value'}, 2))
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.removeControlVariable |
Simulink.VariantConfigurationData.addSubModelConfigurations |
Simulink.VariantConfigurationData.removeSubModelConfiguration

addCopyOfConfiguration

Class: Simulink.VariantConfigurationData
Package: Simulink

Add a copy of an existing variant configuration to the variant configuration data object

Syntax

```
vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration)
vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration,
nameOfTobeAddedConfiguration)
```

Description

`vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration)`, adds a new configuration with a default name (default name is based on existing configuration name being copied) as a copy of the existing configuration to the variant configuration data object.

`vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration, nameOfTobeAddedConfiguration)`, adds a new configuration with a specified name, as a copy of the existing configuration, to the variant configuration data object.

Input Arguments

`nameOfExistingConfiguration`

Name of existing configuration.

Default:

`nameOfTobeAddedConfiguration`

Name of new configuration to be added as a copy of the configuration.

Default:

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller');

% Add a copy of variant configuration LinInterExp
% and name the copy as LinExtExp
vcdataObj.addCopyOfConfiguration('LinInterExp','LinExtExp')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.removeConfiguration |
Simulink.VariantConfigurationData.setDefaultConfiguration

addSubModelConfigurations

Class: Simulink.VariantConfigurationData
Package: Simulink

Add to a variant configuration the names of the configurations to be used for submodels

Syntax

```
vcdataObj.addSubModelConfigurations(nameOfConfiguration,
subModelConfigurations)
```

Description

`vcdataObj.addSubModelConfigurations(nameOfConfiguration, subModelConfigurations)`, specifies names of the configurations to be used for submodels.

Input Arguments

`nameOfConfiguration`

Name for the configuration of submodels that are model references.

`subModelConfigurations`

Vector of structures containing fields: `ModelName`, `ConfigurationName`. The names of submodels must be unique and valid MATLAB variable names and configuration names must be valid MATLAB variables.

Examples

```
% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));
```

```
% Load the model
load_system('slexVariantManagementExample');

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller');

% Add a new submodel configuration to LinInterExp
vcdataObj.addSubModelConfigurations('LinInterExp',...
  [struct('ModelName', 'slexVariantManagementExternalPlantMdlRef',...
          'ConfigurationName', 'LowFid')])
```

See Also

Simulink.VariantConfigurationData | Simulink.VariantConfigurationData.removeSubModelConfiguration | Simulink.VariantConfigurationData.addControlVariables | Simulink.VariantConfigurationData.removeControlVariable

existsFor

Class: Simulink.VariantConfigurationData
Package: Simulink

Check if variant configuration data object exists for a model

Syntax

```
Simulink.VariantConfigurationData.existsFor(modelNameOrHandle)
```

Description

`Simulink.VariantConfigurationData.existsFor(modelNameOrHandle)` returns true if the variant configuration data object exists for the model.

Input Arguments

`modelNameOrHandle`

Name or handle to the model.

Examples

```
% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Checks whether a variant configuration
% data object exists for model
[exists] = Simulink.VariantConfigurationData.existsFor...
  ('slexVariantManagementExample')
```

See Also

Simulink.VariantConfigurationData | Simulink.VariantConfigurationData.getFor

getConfiguration

Class: Simulink.VariantConfigurationData
Package: Simulink

Returns the variant configuration with a given name from a variant configuration data object

Syntax

```
vcdataObj.getConfiguration(nameOfConfiguration)
```

Description

`vcdataObj.getConfiguration(nameOfConfiguration)` returns a specific variant configuration that is associated with the variant configuration data object.

Input Arguments

`nameOfConfiguration`

Name of the variant configuration to be returned.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller');

% Add a control variable SmartSensor1Mod
vcdataObj.addControlVariables('LinInterExp',...
  [struct('Name','SmartSensor1Mod','Value','2')]);
```

```
% Obtain information on the variant configuration..
% LinInterExp from the variant configuration data object
vc = vcdataObj.getConfiguration('LinInterExp')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.removeConfiguration |
Simulink.VariantConfigurationData.getDefaultConfiguration

getDefaultConfiguration

Class: Simulink.VariantConfigurationData
Package: Simulink

Returns default variant configuration, if any, for a variant configuration data object

Syntax

```
vcdataObj.getDefaultConfiguration
```

Description

`vcdataObj.getDefaultConfiguration` returns the default variant configuration. If no default variant configuration is defined, then `[]` is returned.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller');

% Add the variant configuration LinInterStd
vcdataObj.addConfiguration('LinInterStd',...
  'Linear Internal Standard Plant Controller');

% Set LinExtExp as the default variant configuration
vcdataObj.setDefaultConfigurationName('LinExtExp');

% Obtain the default variant configuration
defvc = vcdataObj.getDefaultConfiguration
```

See Also

Simulink.VariantConfigurationData.setDefaultConfigurationName |
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.getConfiguration

getFor

Class: Simulink.VariantConfigurationData
Package: Simulink

Get existing variant configuration data object for a model

Syntax

```
Simulink.VariantConfigurationData.getFor(modelNameOrHandle)
```

Description

`Simulink.VariantConfigurationData.getFor(modelNameOrHandle)`, returns the variant configuration object for the model. If no default variant configuration is defined, then `[ ]` is returned.

Input Arguments

`modelNameOrHandle`

Model name or handle.

Examples

```
% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Obtain variant configuration data object for the model
% slexVariantManagementExample
vcdataObj = Simulink.VariantConfigurationData.getFor...
  ('slexVariantManagementExample')
```

See Also

Simulink.VariantConfigurationData | Simulink.VariantConfigurationData.existsFor | Simulink.VariantConfigurationData.getOrCreateFor

getOrCreateFor

Class: Simulink.VariantConfigurationData
Package: Simulink

Get existing or create a new variant configuration data object for a model

Syntax

```
Simulink.VariantConfigurationData.getOrCreateFor(modelNameOrHandle)
```

Description

`Simulink.VariantConfigurationData.getOrCreateFor(modelNameOrHandle)`, returns the object if the variant configuration data objects exists otherwise, creates an empty object.

Input Arguments

`modelNameOrHandle`

Model name or handle to the model.

Examples

```
% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Obtain existing or create an empty variant configuration
% data object for the slexVariantManagementExample model
vcdataObj = Simulink.VariantConfigurationData.getOrCreateFor...
  ('slexVariantManagementExample')
```

See Also

Simulink.VariantConfigurationData | Simulink.VariantConfigurationData.existsFor | Simulink.VariantConfigurationData.getFor

removeConfiguration

Class: Simulink.VariantConfigurationData
Package: Simulink

Remove a variant configuration with a given name from the variant configuration data object

Syntax

```
vcdataObj.removeConfiguration(nameOfConfiguration)
```

Description

`vcdataObj.removeConfiguration(nameOfConfiguration)` removes the configuration from the variant configuration data object.

Input Arguments

`nameOfConfiguration`

Name of the configuration to be removed.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the LinInterExp variant configuration
% to the variant configuration data object
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller');

% Remove the LinInterExp configuration
% from the variant configuration data object
vcdataObj.removeConfiguration('LinInterExp')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.getConfiguration

removeConstraint

Class: Simulink.VariantConfigurationData
Package: Simulink

Remove a constraint from the variant configuration data object

Syntax

```
vcdataObj.removeConstraint(nameOfConstraint)
```

Description

`vcdataObj.removeConstraint(nameOfConstraint)`, removes the constraint from the variant configuration data object.

Input Arguments

`nameOfConstraint`

Name of the constraint to be removed.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a constraint named LinNotExtern
vcdataObj.addConstraint('LinNotExtern','((Ctrl~=1)...
  || (PlantLocation ~=1))',..
  'Description of the constraint');

% Remove the constraint LinNotExtern
% from the variant configuration
vcdataObj.removeConstraint('LinNotExtern')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConstraint

removeControlVariable

Class: Simulink.VariantConfigurationData
Package: Simulink

Remove a control variable from a variant configuration

Syntax

```
vcdataObj.removeControlVariable(nameOfConfiguration,
nameOfControlVariable)
```

Description

`vcdataObj.removeControlVariable(nameOfConfiguration, nameOfControlVariable)` removes a control variable from a variant configuration.

Input Arguments

`nameOfConfiguration`

Name of the variant configuration.

`nameOfControlVariable`

Name of the control variable to be deleted.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller');
```

```
% Add control variables SmartSensor1Mod and PlanLocation
vcdataObj.addControlVariables('LinInterExp',...
  [struct('Name','SmartSensor1Mod','Value','2')]);

% Remove the control variable SmartSensor1Mod
% from the configuration LinInterExp
vcdataObj.removeControlVariable('LinInterExp',...
  'SmartSensor1Mod')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addControlVariables

removeSubModelConfiguration

Class: Simulink.VariantConfigurationData
Package: Simulink

Remove from a variant configuration, the configuration to be used for a sub model.

Syntax

```
vcdataObj.removeSubModelConfiguration(nameOfConfiguration,
nameOfSubModel)
```

Description

`vcdataObj.removeSubModelConfiguration(nameOfConfiguration, nameOfSubModel)`, removes the configuration specified for a submodel.

Input Arguments

`nameOfConfiguration`

Name of the submodel configuration to be removed.

`nameOfSubModel`

Name of the submodel from which the configuration must be removed.

Examples

```
% Load the model
load_system('slexVariantManagementExample');

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration named LinInterExp
```

```
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller',controlvars);

% Add a new submodel configuration to LinInterExp
vcdataObj.addSubModelConfigurations('LinInterExp',...
  [struct('ModelName','slexVariantManagementExternalPlantMdlRef',...
  'ConfigurationName', 'LowFid')]);

% Remove the submodel configuration LinInterExp
% from the submodel slexVariantManagementExternalPlantMdlRef
vcdataObj.removeSubModelConfiguration('LinInterExp',..
  'slexVariantManagementExternalPlantMdlRef')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addSubModelConfigurations

setDefaultConfigurationName

Class: Simulink.VariantConfigurationData
Package: Simulink

Set name of the default variant configuration for a variant configuration data object

Syntax

```
vcdataObj.setDefaultConfigurationName(nameOfConfiguration)
```

Description

`vcdataObj.setDefaultConfigurationName(nameOfConfiguration)` sets the default configuration name. A variant configuration must exist with the same name. If an empty value is passed, then the default configuration name is cleared.

Input Arguments

`nameOfConfiguration`

Name of the configuration to be set as the default.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the LinInterExp variant configuration
vcdataObj.addConfiguration('LinInterExp',...
  'Linear Internal Experimental Plant Controller');

% Set the configuration LinInterExp as default
vcdataObj.setDefaultConfigurationName('LinInterExp');

% Obtain the default variant configuration
```

```
dconfig = vcdataObj.getDefaultConfiguration
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.getDefaultConfiguration

validateModel

Class: Simulink.VariantConfigurationData
Package: Simulink

Validate all variant blocks in the model and submodels in the hierarchy during simulation

Syntax

```
Simulink.VariantConfigurationData.validateModel(modelName)
Simulink.VariantConfigurationData.validateModel(modelName,
configName)
```

Description

`Simulink.VariantConfigurationData.validateModel(modelName)`, validates the model and referenced models during simulation.

`Simulink.VariantConfigurationData.validateModel(modelName, configName)`, validates the model and referenced models during simulation optionally using a variant configuration.

Input Arguments

`modelName`

Name of the model

`configName`

Name of the configuration to be validated

Examples

```
% Add the path to the model file
```

```
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp');

% Add control variables to LinInterExp
vcdataObj.addControlVariables('LinInterExp',...
  cell2struct({'Ctrl', '1';...
               'PlantLocation', '2';...
               'SimType', '2'},...
              {'Name', 'Value'}, 2));

% Associate this object with the model
set_param('slexVariantManagementExample',...
  'VariantConfigurationObject', 'vcdataObj');

% Validate the model slexVariantManagementExample using
% the configuration LinInterExp
[valid, errors] = Simulink.VariantConfigurationData.validateModel...
  ('slexVariantManagementExample','LinInterExp')
```

See Also

Simulink.VariantConfigurationData | Simulink.VariantConfigurationData.getFor | Simulink.VariantConfigurationData.existsFor | Simulink.VariantConfigurationData.getOrCreateFor

VariantConfigurationData

Class: Simulink.VariantConfigurationData
Package: Simulink

Object constructor with optional arguments for variant configurations, constraints, and default configuration name

Syntax

```
vardataObj = Simulink.VariantConfigurationData(
variantConfigurations)
```

Description

`vardataObj = Simulink.VariantConfigurationData(variantConfigurations)`, constructor that creates an empty variant configuration data object. Optionally, can also accept constraints and a default configuration name as inputs.

Input Arguments

`variantConfigurations`

Configurations that are part of the variant configuration data object.

`constraints`

Constraints to be satisfied by the model.

`defaultConfigurationName`

Name of the default configuration

Examples

```
% Create an empty variant configuration data object
```

```
vcdataObj = Simulink.VariantConfigurationData
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration
| Simulink.VariantConfigurationData.addConstraint |
Simulink.VariantConfigurationData.addControlVariables |
Simulink.VariantConfigurationData.addSubModelConfigurations

Simulink.WorkspaceVar class

Package: Simulink

Contains information about workspace variables and blocks that use them

---

Note: `Simulink.WorkspaceVar` will be removed in a future release. Use Simulink.VariableUsage instead.

Description

A `Simulink.WorkspaceVar` object describes attributes of a workspace variable and lists the blocks that use the variable. The `Simulink.findVars` function returns one or more `Simulink.WorkspaceVar` objects that embody the results of searching for variables.

Tip

Only a `Simulink.WorkspaceVar` constructor can set any field value in a `Simulink.WorkspaceVar` object. The fields are otherwise read-only.

Properties

`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`.

Workspace value	Meaning
`'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

Simulink.VariableUsage class

Package: Simulink

Get information about workspace variables and blocks that use them

Tip

Only a `Simulink.VariableUsage` constructor can set any property value in a `Simulink.VariableUsage` object. The properties 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.

You can also use the `Simulink.findVars` function to create `Simulink.VariableUsage` objects that describe the variables and, optionally, enumerated data types that are used by a model.

Construction

`vars = Simulink.VariableUsage(VarNames,SourceName)` creates an array of `Simulink.VariableUsage` objects to describe the variables `VarNames`. The constructor sets the `Name` property of each object to one of the variable names specified by `VarNames`, and sets the `Source` property of all the objects to the source specified by `SourceName`.

You can specify `VarNames` with variables that are not used in any loaded models.

Input Arguments

`VarNames` — Names of target variables

string | cell array of strings

Names of target variables, specified as a string or a cell array of strings. The constructor creates a `Simulink.VariableUsage` object for each variable name.

Example: `'k'`

Example: `{'k','asdf','fuelFlow'}`

Data Types: `char | cell`

`SourceName` — Name of variable source

string

Name of the source that defines the target variables, specified as a string. For example, you can specify the MATLAB base workspace or a data dictionary as a source. The constructor also determines and sets the `SourceType` property of each of the returned `Simulink.VariableUsage` objects.

Example: `'base workspace'`

Example: `'myModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

Properties

`Name` — Name of variable or enumerated type

string

The name of the variable or enumerated data type the object describes, returned as a string.

`Source` — Name of defining workspace

string

The name of the workspace or data dictionary that defines the described variable, returned as a string. The table shows some examples.

`Source` value	Meaning
`'base workspace'`	MATLAB base workspace
`'MyModel'`	Model workspace for the model `MyModel`
`'MyModel/Mask1'`	Mask workspace for the masked block `Mask1` in the model `MyModel`

Source value	Meaning
'sldemo_fuelsys_dd_controller.sl	The data dictionary named 'sldemo_fuelsys_dd_controller.sldd'

The table shows some examples if you created the Simulink.VariableUsage object by using the Simulink.findVars function to find enumerated data types.

Source value	Meaning
'BasicColors.m'	The enumerated type is defined in the MATLAB file 'BasicColors.m'.
''	The enumerated type is defined dynamically and has no source.
'sldemo_fuelsys_dd_controller.sl	The enumerated type is defined in the data dictionary named 'sldemo_fuelsys_dd_controller.sldd'.

SourceType — Type of defining workspace

string

The type of the workspace that defines the variable, returned as a string. The possible values are:

- 'base workspace'
- 'model workspace'
- 'mask workspace'
- 'data dictionary'

If you created the Simulink.VariableUsage object by using the Simulink.findVars function to find enumerated data types, the possible values are:

- 'MATLAB file'
- 'dynamic class'
- 'data dictionary'

Users — Model blocks that use the variable or models that use the enumerated type

cell array of strings

Model blocks that use the variable or models that use the enumerated type, returned as a cell array of strings. Each string names a block or model that uses the variable or enumerated type. The Simulink.findVars function populates this field.

Methods

intersect	Intersection of two arrays of `Simulink.VariableUsage` objects
setdiff	Return difference between two arrays of `Simulink.VariableUsage` objects

Examples

Return a `Simulink.VariableUsage` object for the variable `'k'` in the base workspace.

```
var = Simulink.VariableUsage('k','base workspace');
```

Return an array of `Simulink.VariableUsage` objects containing one object for each variable returned by the `whos` command.

```
vars = Simulink.VariableUsage(whos,'base workspace')
```

Return an array of `Simulink.VariableUsage` objects that describes all the variables in a model workspace.

```
hws = get_param('mymodel','ModelWorkspace');
vars = Simulink.VariableUsage(hws.whos,'MyModel')
```

Return an array of `Simulink.VariableUsage` objects that describes all the variables in a mask workspace.

```
maskVars = get_param('mymodel/maskblock','MaskWSVariables');
vars = Simulink.VariableUsage(maskVars,'mymodel/maskblock');
```

See Also

`Simulink.data.existsInGlobal` | `Simulink.findVars`

More About

- "Model Exploration"
- "Variables"

intersect

Class: Simulink.VariableUsage
Package: Simulink

Intersection of two arrays of `Simulink.VariableUsage` objects

Syntax

```
VarsOut = intersect(VarsIn1,VarsIn2)
```

Description

`VarsOut = intersect(VarsIn1,VarsIn2)` returns an array that identifies the variables described in `VarsIn1` and in `VarsIn2`, which are arrays of `Simulink.VariableUsage` objects. If a variable is described by a `Simulink.VariableUsage` object in `VarsIn1` and in `VarsIn2`, the function returns a `Simulink.VariableUsage` object that stores the variable usage information from both objects in the `Users` property.

`intersect` compares the `Name`, `Source`, and `SourceType` properties of the `Simulink.VariableUsage` objects in `VarsIn1` with the same properties of the objects in `VarsIn2`. If `VarsIn1` and `VarsIn2` each contain `Simulink.VariableUsage` objects that have the same values for these three properties, they both describe the same variable.

Input Arguments

`VarsIn1` — First array of variables for comparison

array of `Simulink.VariableUsage` objects

First array of variables for comparison, specified as an array of `Simulink.VariableUsage` objects.

`VarsIn2` — Second array of variables for comparison

array of `Simulink.VariableUsage` objects

Second array of variables for comparison, specified as an array of `Simulink.VariableUsage` objects.

Output Arguments

`VarsOut` — Variables described in both input arrays

array of `Simulink.VariableUsage` objects

Variables that are described in both input arrays, returned as an array of `Simulink.VariableUsage` objects. The function returns an object for each variable that is described in `VarsIn1` and in `VarsIn2`.

Examples

Compare Variables Used by Models

Given two models, discover the variables needed by both models.

```
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
commonVars = intersect(model1Vars,model2Vars);
```

See Also

Simulink.VariableUsage | setdiff | `Simulink.findVars`

More About

- "Model Exploration"
- "Variables"

setdiff

Class: Simulink.VariableUsage
Package: Simulink

Return difference between two arrays of `Simulink.VariableUsage` objects

Syntax

```
VarsOut = setdiff(VarsIn1,VarsIn2)
```

Description

`VarsOut = setdiff(VarsIn1,VarsIn2)` returns an array that identifies the variables described in `VarsIn1` but not in `VarsIn2`, which are arrays of Simulink.VariableUsage objects. If a variable is described by a `Simulink.VariableUsage` object in `VarsIn1` but not in `VarsIn2`, the function returns a copy of the object.

`setdiff` compares the `Name`, `Source`, and `SourceType` properties of the `Simulink.VariableUsage` objects in `VarsIn1` with the same properties of the objects in `VarsIn2`. If `VarsIn1` and `VarsIn2` each contain a `Simulink.VariableUsage` object with the same values for these three properties, the objects describe the same variable, and `setdiff` does not return an object to describe it.

Input Arguments

`VarsIn1` — First array of variables for comparison

array of `Simulink.VariableUsage` objects

First array of variables for comparison, specified as an array of `Simulink.VariableUsage` objects.

`VarsIn2` — Second array of variables for comparison

array of `Simulink.VariableUsage` objects

Second array of variables for comparison, specified as an array of `Simulink.VariableUsage` objects.

Output Arguments

`VarsOut` — Variables described in first array but not second array

array of `Simulink.VariableUsage` objects

Variables that are described in the first input array but not in the second input array, returned as an array of `Simulink.VariableUsage` objects. The function returns an object for each variable that is described in `VarsIn1` but not in `VarsIn2`.

Examples

Determine Variable Usage Difference Between Models

Given two models, discover the variables that are needed by the first model but not the second model.

```
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
commonVars = setdiff(model1Vars,model2Vars);
```

Find Variables Not Used by Model

Locate all variables in the base workspace that are not used by a loaded model that has been recently compiled.

```
models = find_system('type','block_diagram','LibraryType','None');
base_vars = Simulink.VariableUsage(who);
used_vars = Simulink.findVars(models,'WorkspaceType','base');
unusedVars = setdiff(base_vars,used_vars);
```

See Also

Simulink.VariableUsage | intersect | `Simulink.findVars`

More About

- “Model Exploration”

- “Variables”

Simulink.data.Dictionary class

Package: Simulink.data

Configure data dictionary

Description

An object of the `Simulink.data.Dictionary` class represents a data dictionary. The object allows you to perform operations on the data dictionary such as save or discard changes, import data from the base workspace, and add other data dictionaries as references.

Construction

The functions `Simulink.data.dictionary.create` and `Simulink.data.dictionary.open` create a `Simulink.data.Dictionary` object.

Properties

`DataSources` — Referenced data dictionaries

cell array of strings

Referenced data dictionaries by file name, returned as a cell array of strings. This property only lists directly referenced dictionaries whose parent is the `Simulink.data.Dictionary` object. This property is read only.

`HasUnsavedChanges` — Indicator of unsaved changes

0 | 1

Indicator of unsaved changes to the data dictionary, returned as `0` or `1`. The value is `1` if changes have been made since last data dictionary save and `0` if not. This property is read only.

`NumberOfEntries` — Total number of entries in data dictionary

integer

Total number of entries in data dictionary, including those in referenced dictionaries, returned as an integer. This property is read only.

Methods

addDataSource	Add reference data dictionary to parent data dictionary
close	Close connection between data dictionary and `Simulink.data.Dictionary` object
discardChanges	Discard changes to data dictionary
filepath	Full path and file name of data dictionary
getSection	Return `Simulink.data.dictionary.Section` object to represent data dictionary section
hide	Remove data dictionary from Model Explorer
importEnumTypes	Import enumerated type definitions to data dictionary
importFromBaseWorkspace	Import base workspace variables to data dictionary
listEntry	List data dictionary entries
removeDataSource	Remove reference data dictionary from parent data dictionary
saveChanges	Save changes to data dictionary
show	Show data dictionary in Model Explorer

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create New Data Dictionary and Data Dictionary Object

Create a data dictionary file `myNewDictionary.sldd` and a `Simulink.data.Dictionary` object representing the new data dictionary. Assign the object to variable `dd1`.

```
dd1 = Simulink.data.dictionary.create('myNewDictionary.sldd')
```

```
dd1 =

  data dictionary with properties:

          DataSources: {0x1 cell}
    HasUnsavedChanges: 0
           NumberOfEntries: 0
```

Open Existing Data Dictionary

Create a `Simulink.data.Dictionary` object representing the existing data dictionary `myDictionary_ex_API.sldd`. Assign the object to variable `dd2`.

```
dd2 = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')
```

```
dd2 =

  Dictionary with properties:

          DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 0
           NumberOfEntries: 4
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.dictionary.create` | Simulink.data.dictionary.Entry | `Simulink.data.dictionary.open` | Simulink.data.dictionary.Section

More About

- “What Is a Data Dictionary?”

Introduced in R2015a

addDataSource

Class: Simulink.data.Dictionary
Package: Simulink.data

Add reference data dictionary to parent data dictionary

Syntax

```
addDataSource(dictionaryObj,refDictionaryFile)
```

Description

`addDataSource(dictionaryObj,refDictionaryFile)` adds a data dictionary, `refDictionaryFile`, as a reference dictionary to a parent dictionary `dictionaryObj`, a Simulink.data.Dictionary object.

The parent dictionary contains all the entries that are defined in the referenced dictionary until the referenced dictionary is removed from the parent dictionary. The `DataSource` property of an entry indicates the dictionary that defines the entry.

Input Arguments

`dictionaryObj` — Parent data dictionary

`Simulink.data.Dictionary` object

Parent data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

`refDictionaryFile` — File name of data dictionary to reference

string

File name of data dictionary to reference, specified as a string that includes the `.sldd` extension. The data dictionary file must be on your MATLAB path.

Example: 'mySubDictionary_ex_API.sldd'

Data Types: char

Examples

Add a Reference Data Dictionary to a Parent Data Dictionary

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Add the data dictionary mySubDictionary_ex_API.sldd as a reference dictionary to myDictionary_ex_API.sldd.

```
addDataSource(myDictionaryObj,'mySubDictionary_ex_API.sldd');
```

Confirm the addition by viewing the DataSources property of variable myDictionaryObj. The property returns the name of the newly referenced dictionary.

```
myDictionaryObj.DataSources
```

```
ans =

    'myRefDictionary_ex_API.sldd'
    'mySubDictionary_ex_API.sldd'
```

- "Store Data in Dictionary Programmatically"

Alternatives

You can use the Model Explorer window to manage reference dictionaries. See "Partition Dictionary Data Using Referenced Dictionaries" for more information.

See Also

removeDataSource | Simulink.data.Dictionary

Introduced in R2015a

close

Class: Simulink.data.Dictionary
Package: Simulink.data

Close connection between data dictionary and `Simulink.data.Dictionary` object

Syntax

```
close(dictionaryObj)
```

Description

`close(dictionaryObj)` closes the connection between the Simulink.data.Dictionary object `dictionaryObj` and the data dictionary it represents. `dictionaryObj` remains as a `Simulink.data.Dictionary` object but no longer represents any data dictionary.

Tips

- Use the `close` function in a custom MATLAB function to disassociate a `Simulink.data.Dictionary` object from a data dictionary. Custom MATLAB functions can create and store variables and objects in function workspaces but cannot delete those variables and objects.
- The `close` function does not affect the content or the state of the represented data dictionary. The function does not discard unsaved changes to the represented dictionary or entries. You can save or discard them later.

Input Arguments

`dictionaryObj` — Target `Simulink.data.Dictionary` object

handle to `Simulink.data.Dictionary` object

Target `Simulink.data.Dictionary` object, specified as a handle to the object.

See Also

Simulink.data.Dictionary

Related Examples

- "Store Data in Dictionary Programmatically"

Introduced in R2015a

discardChanges

Class: Simulink.data.Dictionary
Package: Simulink.data

Discard changes to data dictionary

Syntax

```
discardChanges(dictionaryObj)
```

Description

`discardChanges(dictionaryObj)` discards all changes made to the specified data dictionary since the last time changes to the dictionary were saved using the `saveChanges` function. `discardChanges` also discards changes made to referenced data dictionaries. The changes to the target dictionary and its referenced dictionaries are permanently lost.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Discard Changes to Data Dictionary

Create a `Simulink.data.Dictionary` object representing the data dictionary `myDictionary_ex_API.sldd` and assign the object to variable `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')

myDictionaryObj =

  Dictionary with properties:

          DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 0
           NumberOfEntries: 4
```

Make a change to `myDictionary_ex_API.sldd` by adding an entry named `myNewEntry` with value `237`. View the `HasUnsavedChanges` property of `myDictionaryObj` to confirm a change was made.

```
addEntry(dDataSectObj,'myNewEntry',237);
myDictionaryObj

myDictionaryObj =

  Dictionary with properties:

          DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 1
           NumberOfEntries: 5
```

Discard all changes to `myDictionary_ex_API.sldd`. The `HasUnsavedChanges` property of `myDictionaryObj` indicates changes were discarded.

```
discardChanges(myDictionaryObj)
myDictionaryObj

myDictionaryObj =

  Dictionary with properties:

          DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 0
           NumberOfEntries: 4
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to discard changes to data dictionaries. See "View and Revert Changes to Dictionary Entries" for more information.

See Also

Simulink.data.Dictionary | saveChanges

More About

- "What Is a Data Dictionary?"

Introduced in R2015a

filepath

Class: Simulink.data.Dictionary
Package: Simulink.data

Full path and file name of data dictionary

Syntax

```
dictionaryFilePath = filepath(dictionaryObj)
```

Description

`dictionaryFilePath = filepath(dictionaryObj)` returns the full path and file name of the data dictionary `dictionaryObj`, a Simulink.data.Dictionary object.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Return Path of Data Dictionary File

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Return the full path of `myDictionary_ex_API.sldd` and assign it to variable `myDictionaryFilePath`.

```
myDictionaryFilePath = filepath(myDictionaryObj)
```

```
myDictionaryFilePath =

C:\Users\jsmith\myDictionary_ex_API.sldd
```

- "Store Data in Dictionary Programmatically"

See Also

Simulink.data.Dictionary

Introduced in R2015a

getSection

Class: Simulink.data.Dictionary
Package: Simulink.data

Return `Simulink.data.dictionary.Section` object to represent data dictionary section

Syntax

```
sectionObj = getSection(dictionaryObj,sectionName)
```

Description

`sectionObj = getSection(dictionaryObj,sectionName)` returns a Simulink.data.dictionary.Section object representing one section, `sectionName`, of a data dictionary `dictionaryObj`, a Simulink.data.Dictionary object.

Input Arguments

`dictionaryObj` — Data dictionary containing target section

`Simulink.data.Dictionary` object

Data dictionary containing target section, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

`sectionName` — Name of target data dictionary section

string

Name of target data dictionary section, specified as a string.

Example: `'Design Data'`

Example: `'Configurations'`

Data Types: `char`

Examples

Create New Data Dictionary Section Object

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
dDataSectObj = getSection(myDictionaryObj,'Design Data')
```

```
dDataSectObj =

  Section with properties:

    Name: 'Design Data'
```

- "Store Data in Dictionary Programmatically"

See Also

Simulink.data.Dictionary | Simulink.data.dictionary.Section

Introduced in R2015a

hide

Class: Simulink.data.Dictionary
Package: Simulink.data

Remove data dictionary from Model Explorer

Syntax

```
hide(dictionaryObj)
```

Description

`hide(dictionaryObj)` removes the data dictionary `dictionaryObj` from the **Model Hierarchy** pane of Model Explorer. The target dictionary no longer appears as a node in the model hierarchy tree. Use this function when you are finished working with a data dictionary and want to reduce clutter in the Model Explorer.

Tips

- To add a data dictionary as a node in the model hierarchy tree in Model Explorer, use the show function or use the interface to open and view the dictionary in Model Explorer.
- The `hide` function does not affect the content of the target dictionary.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Hide Data Dictionary from Model Explorer

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Open Model Explorer and display the new data dictionary as the selected tree node in the **Model Hierarchy** pane.

```
show(myDictionaryObj)
```

With Model Explorer open, at the MATLAB command prompt, call the `hide` function to observe the removal of `myDictionary_ex_API.sldd` from the model hierarchy tree.

```
hide(myDictionaryObj)
```

- "Store Data in Dictionary Programmatically"

Alternatives

You can remove a data dictionary from the **Model Hierarchy** pane of Model Explorer by right-clicking the dictionary tree node and selecting **Close**.

See Also

show | Simulink.data.Dictionary

Introduced in R2015a

importEnumTypes

Class: Simulink.data.Dictionary
Package: Simulink.data

Import enumerated type definitions to data dictionary

Syntax

```
importedTypes = importEnumTypes(dictionaryObj,targetTypes)
[importedTypes,importFailures] = importEnumTypes(dictionaryObj,
targetTypes)
```

Description

`importedTypes = importEnumTypes(dictionaryObj,targetTypes)` imports to the data dictionary `dictionaryObj` the definitions of one or more enumerated types `targetTypes`. `importEnumTypes` does not import MATLAB variables created using enumerated types but instead, in support of those variables, imports the definitions of the types. The target data dictionary stores the definition of a successfully imported type as an entry. This syntax returns a list of the names of successfully imported types.

`[importedTypes,importFailures] = importEnumTypes(dictionaryObj, targetTypes)` additionally returns a list of any target types that were not successfully imported. You can inspect the list to determine the reason for each failure.

Tips

- Before you can import an enumerated data type definition to a data dictionary, you must clear the base workspace of any variables created using the target type.
- You can define an enumerated type using a `classdef` block in a MATLAB file or a P-file. `importEnumTypes` imports type definitions directly from these files if you specify the names of the types to import using the input argument `targetTypes` and if the files defining the types are on your MATLAB path.
- To avoid conflicting definitions for imported types, `importEnumTypes` renders MATLAB files or P-files ineffective by appending `.save` to their names. The `.save`

extensions cause variables to rely on the definitions in the target data dictionary and not on the definitions in the files. You can remove the `.save` extensions to restore the files to their original state.

- You can use `importEnumTypes` to import enumerated types defined using the `Simulink.defineIntEnumType` function. Because such types are not defined using MATLAB files or P-files, `importEnumTypes` does not rename any files.
- Use the function `Simulink.findVars` to generate a list of the enumerated types that are used by a model. Then, use the list with `importEnumTypes` to import the definitions of the types to a data dictionary. See "Enumerations in Data Dictionary" for more information.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

`targetTypes` — Enumerated type definitions to import

cell array of strings

Enumerated type definitions to import, specified as a cell array of strings. If any target types are defined using `classdef` blocks in MATLAB files or P-files, the files must be available on your MATLAB path so that `importEnumTypes` can disable them.

Example: `{'myEnumType'}`

Example: `{'myFirstEnumType','mySecondEnumType','myThirdEnumType'}`

Data Types: `cell`

Output Arguments

`importedTypes` — Target types successfully imported

array of structures

Target enumerated type definitions successfully imported, returned as an array of structures. Each structure in the array represents one imported type. The `className` field of each structure identifies a type by name and the `renamedFiles` field identifies any renamed MATLAB files or P-files.

`importFailures` — Target types not imported

array of structures

Enumerated type definitions targeted but not imported, returned as an array of structures. Each structure in the array represents one type not imported. The `className` field of each structure identifies a type by name and the `reason` field explains the failure.

Examples

Import Enumerated Data to Data Dictionary

Create a data dictionary `myNewDictionary.sldd` in your current working folder and a `Simulink.data.Dictionary` object representing the new data dictionary. Assign the object to the variable `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd');
```

Run the script in the MATLAB file `myDataEnum_ex_API.m`. The file defines an enumerated type named `InstrumentTypes` using the `Simulink.defineIntEnumType` function and creates three variables based on the new type. Then, import the new variables from the base workspace to `myDictionary_ex_API.sldd`.

```
myDataEnum_ex_API
importFromBaseWorkspace(myDictionaryObj,'varList',...
{'firstEnumVariable','secondEnumVariable','thirdEnumVariable'});
```

Clear the imported variables from the base workspace. Before you can import an enumerated data type definition to the target data dictionary, you must clear the base workspace of any variables created using the target type.

```
clear firstEnumVariable
clear secondEnumVariable
clear thirdEnumVariable
```

Import the data type definition to `myDictionary_ex_API.sldd`.

```
importEnumTypes(myDictionaryObj,{'InstrumentTypes'})

ans =

       className: 'InstrumentTypes'
    renamedFiles: {}
```

- “Enumerations in Data Dictionary”
- “Store Data in Dictionary Programmatically”

See Also

importFromBaseWorkspace | Simulink.data.Dictionary

Introduced in R2015a

importFromBaseWorkspace

Class: Simulink.data.Dictionary
Package: Simulink.data

Import base workspace variables to data dictionary

Syntax

```
importedVars = importFromBaseWorkspace(dictionaryObj)
importedVars = importFromBaseWorkspace(dictionaryObj,Name,Value)
[importedVars,existingVars] = importFromBaseWorkspace(___)
```

Description

`importedVars = importFromBaseWorkspace(dictionaryObj)` imports all variables from the MATLAB base workspace to the data dictionary `dictionaryObj` without overwriting existing entries in the dictionary. If any base workspace variables are already in the dictionary, the function present a warning and a list.

This syntax returns a list of names of the successfully imported variables. A variable is considered successfully imported only if `importFromBaseWorkspace` assigns the value of the variable to the corresponding entry in the target data dictionary.

`importedVars = importFromBaseWorkspace(dictionaryObj,Name,Value)` imports base workspace variables to a data dictionary, with additional options specified by one or more `Name,Value` pair arguments.

`[importedVars,existingVars] = importFromBaseWorkspace(___)` additionally returns a list of variables that were not overwritten. Use this syntax if `existingVarsAction` is set to `'none'`, the default value, which prevents existing dictionary entries from being overwritten.

Tips

- `importFromBaseWorkspace` can import MATLAB variables created from enumerated data types but cannot import the definitions of the enumerated types.

Use the importEnumTypes function to import enumerated data type definitions to a data dictionary. If you import variables of enumerated data types to a data dictionary but do not import the enumerated type definitions, the dictionary is less portable and might not function properly if used by someone else.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'clearWorkspaceVars'` — Flag to clear base workspace of imported variables

`false` (default) | `true`

Flag to clear the base workspace of any successfully imported variables, specified as the comma-separated pair consisting of `'clearWorkspaceVars'` and `true` or `false`.

Example: `'clearWorkspaceVars',true`

Data Types: `logical`

`'existingVarsAction'` — Action to take for existing dictionary variables

`'none'` (default) | `'error'` | `'overwrite'`

Action to take for existing dictionary variables, specified as the comma-separated pair consisting of `'existingVarsAction'` and `'none'`, `'error'`, or `'overwrite'`.

If you specify `'none'`, `importFromBaseWorkspace` attempts to import target variables but does not import or make any changes to variables that are already in the data dictionary.

If you specify `'error'`, `importFromBaseWorkspace` returns an error, without importing any variables, if any target variables are already in the data dictionary.

If you specify `'overwrite'`, `importFromBaseWorkspace` imports all target variables and overwrites any variables that are already in the data dictionary.

Example: `'existingVarsAction','error'`

Data Types: `char`

`'varList'` — Variables to import

cell array of strings

Names of specific base workspace variables to import, specified as the comma-separated pair consisting of `'varList'` and a cell array of strings. If you want to import only one variable, specify the name inside a cell array. If you do not specify `'varList'`, `importFromBaseWorkspace` imports all variables from the MATLAB base workspace.

Example: `'varList',{'a','myVariable','fuelFlow'}`

Example: `'varList',{'fuelFlow'}`

Data Types: `cell`

Output Arguments

`importedVars` — Successfully imported variables

cell array of strings

Names of successfully imported variables, returned as a cell array of strings. A variable is considered successfully imported only if `importFromBaseWorkspace` assigns the value of the variable to the corresponding entry in the target data dictionary.

`existingVars` — Variables that were not imported

cell array of strings

Names of target variables that were not imported due to their existence in the target data dictionary, returned as a cell array of strings. `existingVars` has content only if `'existingVarsAction'` is set to `'none'` which is also the default. In that case `importFromBaseWorkspace` imports only variables that are not already in the target data dictionary.

Examples

Import All Base Workspace Variables to Data Dictionary

In the MATLAB base workspace, create variables to import.

```
a = 'String Variable';
myVariable = true;
fuelFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import all base workspace variables to the data dictionary and return a list of successfully imported variables. If any base workspace variables are already in `myDictionary_ex_API.sldd`, `importFromBaseWorkspace` presents a warning and a list of the affected variables.

```
importFromBaseWorkspace(myDictionaryObj);
```

```
Warning: The following variables were not imported because
they already exist in the dictionary:
   fuelFlow
```

Specify Variables to Import to Data Dictionary from Base Workspace

In the MATLAB base workspace, create variables to import.

```
b = 'String Variable';
mySecondVariable = true;
airFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import only the new base workspace variables to the data dictionary.

```
importFromBaseWorkspace(myDictionaryObj,'varList',...
```

```
{'b','mySecondVariable','airFlow'});
```

Import Variables from Base Workspace and Overwrite Conflicts

In the MATLAB base workspace, create a variable to import.

```
fuelFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`. `myDictionary_ex_API.sldd` already contains an entry called `fuelFlow`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the variable `fuelFlow` and overwrite the corresponding entry in `myDictionary_ex_API.sldd`.

```
importFromBaseWorkspace(myDictionaryObj,'varList',{'fuelFlow'},...
'existingVarsAction','overwrite');
```

`importFromBaseWorkspace` assigns the value of the base workspace variable `fuelFlow` to the value of the corresponding entry in `myDictionary_ex_API.sldd`.

Return Variables Not Imported to Data Dictionary from Base Workspace

Return a list of variables that are not imported from the MATLAB base workspace because they are already in the target data dictionary.

In the MATLAB base workspace, create variables to import.

```
fuelFlow = 324;
myNewVariable = 'This is a string.'
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`. `myDictionary_ex_API.sldd` already contains an entry called `fuelFlow`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the variables `fuelFlow` and `myNewVariable` to the data dictionary. Specify names for the output arguments of `importFromBaseWorkspace` to return the names of successfully and unsuccessfully imported variables.

```
[importedVars,existingVars] = importFromBaseWorkspace(myDictionaryObj,...
```

```
'varList',{'fuelFlow','myNewVariable'})

importedVars =

    'myNewVariable'


existingVars =

    'fuelFlow'
```

`importFromBaseWorkspace` does not import the variable `fuelflow` because it is already in the target data dictionary.

- "Store Data in Dictionary Programmatically"

Alternatives

- When you use the Simulink Editor to link a model to a data dictionary, you can choose to import model variables from the base workspace. See "Migrate Single Model to Use Dictionary" for more information.
- You can also use the Model Explorer window to drag-and-drop variables from the base workspace into a data dictionary.

See Also

importEnumTypes | Simulink.data.Dictionary

Introduced in R2015a

listEntry

Class: Simulink.data.Dictionary
Package: Simulink.data

List data dictionary entries

Syntax

```
listEntry(dictionaryObj)
listEntry(dictionaryObj,Name,Value)
```

Description

`listEntry(dictionaryObj)` displays in the MATLAB Command Window a table of information about all the entries in the data dictionary `dictionaryObj`, a Simulink.data.Dictionary object. The displayed information includes the name of each entry, the name of the section containing each entry, the status of each entry, the date and time each entry was last modified, the last user name to modify each entry, and the class of the value each entry contains. By default, the function sorts the list of entries alphabetically by entry name.

`listEntry(dictionaryObj,Name,Value)` displays the entries in a data dictionary with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Ascending'` — Sort order of list

`true` (default) | `false`

Sort order of the list of data dictionary entries, specified as the comma-separated pair consisting of `'Ascending'` and `true` or `false`. If you specify `false`, `listEntry` sorts the list in descending order.

Example: `'Ascending',false`

Data Types: `logical`

`'Class'` — Criteria to filter list by class

string

Criteria to filter the list of data dictionary entries by class, specified as the comma-separated pair consisting of `'Class'` and a string identifying a valid class. The function lists only entries whose values are of the specified class.

Example: `'Class','Simulink.Parameter'`

Data Types: `char`

`'LastModifiedBy'` — Criteria to filter list by user name of last modifier

string

Criteria to filter the list of data dictionary entries by the user name of the last user to modify each entry, specified as the comma-separated pair consisting of `'LastModifiedBy'` and a string identifying the specified user name. The function lists only entries that were last modified by the specified user name.

Example: `'LastModifiedBy','jsmith'`

Data Types: `char`

`'Limit'` — Maximum number of entries to list

integer

Maximum number of entries to list, specified as the comma-separated pair consisting of `'Limit'` and an integer. The function lists up to the specified number of entries starting from the top of the sorted and filtered list.

Example: `'Limit',9`

Data Types: `double`

`'Name'` — Criteria to filter list by entry name

string

Criteria to filter the list of data dictionary entries by entry name, specified as the comma-separated pair consisting of `'Name'` and a string defining the filter criteria. You can use an asterisk character, `*`, as a wildcard to represent any number of characters. The function lists only entries whose names match the filter criteria.

Example: `'Name','fuelFlow'`

Example: `'Name','fuel*'`

Data Types: `char`

`'Section'` — Criteria to filter list by data dictionary section

string

Criteria to filter the list of data dictionary entries by section, specified as the comma-separated pair consisting of `'Section'` and a string identifying the target section. The function lists only entries that are contained in the target section.

Example: `'Section','Design Data'`

`'SortBy'` — Flag to sort list by specific property

`'Name'` (default) | `'Section'` | `'LastModified'` | `'LastModifiedBy'`

Flag to sort the list of data dictionary entries by a specific property, specified as the comma-separated pair consisting of `'SortBy'` and a string identifying a property in the list of entries. Valid properties include `'Name'`, `'Section'`, `'LastModified'`, and `'LastModifiedBy'`.

Example: `'SortBy','LastModifiedBy'`

Examples

List All Entries in Data Dictionary

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the data dictionary.

```
listEntry(myDictionaryObj)
```

Sort List of Data Dictionary Entries in Descending Order

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the data dictionary and sort the list in descending order by entry name.

```
listEntry(myDictionaryObj,'Ascending',false)
```

Filter List of Data Dictionary Entries by Name

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List only the entries in the data dictionary whose names begin with `max`.

```
listEntry(myDictionaryObj,'Name','max*')
```

Sort List of Data Dictionary Entries by Time of Modification

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the dictionary and sort the list by the date and time each entry was last modified.

```
listEntry(myDictionaryObj,'SortBy','LastModified')
```

- “Store Data in Dictionary Programmatically”

See Also

evalin | Simulink.data.Dictionary | Simulink.data.dictionary.Entry

More About

- “What Is a Data Dictionary?”

Introduced in R2015a

removeDataSource

Class: Simulink.data.Dictionary
Package: Simulink.data

Remove reference data dictionary from parent data dictionary

Syntax

```
removeDataSource(dictionaryObj,refDictionaryFile)
```

Description

`removeDataSource(dictionaryObj,refDictionaryFile)` removes a referenced data dictionary, `refDictionaryFile`, from a parent dictionary `dictionaryObj`, a Simulink.data.Dictionary object.

The parent dictionary no longer contains the entries that are defined in the referenced dictionary.

Input Arguments

`dictionaryObj` — Parent data dictionary

`Simulink.data.Dictionary` object

Parent data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

`refDictionaryFile` — File name of referenced data dictionary

string

File name of referenced data dictionary, specified as a string that includes the `.sldd` extension. The data dictionary file must be on your MATLAB path.

Example: 'myRefDictionary_ex_API.sldd'

Data Types: `char`

Examples

Remove Referenced Data Dictionary from Parent Data Dictionary

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`. The `DataSources` property of `myDictionaryObj` indicates `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')
```

```
myDictionaryObj =

  Dictionary with properties:

         DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 0
           NumberOfEntries: 4
```

Remove `myRefDictionary_ex_API.sldd` from `myDictionary_ex_API.sldd`.

```
removeDataSource(myDictionaryObj,'myRefDictionary_ex_API.sldd');
```

View the properties of the `Simulink.data.Dictionary` object `myDictionaryObj`, which represents the parent data dictionary. The `DataSources` property confirms the removal of `myRefDictionary_ex_API.sldd`.

```
myDictionaryObj
```

```
myDictionaryObj =

  Dictionary with properties:

         DataSources: {0x1 cell}
    HasUnsavedChanges: 1
           NumberOfEntries: 3
```

- "Store Data in Dictionary Programmatically"

Alternatives

You can use Model Explorer to manage reference dictionaries. See “Partition Dictionary Data Using Referenced Dictionaries” for more information.

See Also

addDataSource | Simulink.data.Dictionary

Introduced in R2015a

saveChanges

Class: Simulink.data.Dictionary
Package: Simulink.data

Save changes to data dictionary

Syntax

```
saveChanges(dictionaryObj)
```

Description

`saveChanges(dictionaryObj)` saves all changes made to a data dictionary `dictionaryObj`, a Simulink.data.Dictionary object. `saveChanges` also saves changes made to referenced data dictionaries. The previous states of the target dictionary and its referenced dictionaries are permanently lost.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Save Changes to Data Dictionary

Create a new data dictionary `myNewDictionary.sldd` and represent the Design Data section with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd')
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

```
myDictionaryObj =

  data dictionary with properties:

          DataSources: {0x1 cell}
    HasUnsavedChanges: 0
           NumberOfEntries: 0
```

Change `myNewDictionary.sldd` by adding an entry named `myNewEntry` with value `237`. View the `HasUnsavedChanges` property of `myDictionaryObj` to confirm a change was made.

```
addEntry(dDataSectObj,'myNewEntry',237);
myDictionaryObj
```

```
myDictionaryObj =

  Dictionary with properties:

          DataSources: {0x1 cell}
    HasUnsavedChanges: 1
           NumberOfEntries: 1
```

Save all changes to `myNewDictionary.sldd`. The `HasUnsavedChanges` property of `myDictionaryObj` indicates changes were saved.

```
saveChanges(myDictionaryObj)
myDictionaryObj
```

```
myDictionaryObj =

  Dictionary with properties:

          DataSources: {0x1 cell}
    HasUnsavedChanges: 0
           NumberOfEntries: 1
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to save changes to a data dictionary by right-clicking on the dictionary tree node in the **Model Hierarchy** pane and selecting **Save Changes**.

See Also

discardChanges | Simulink.data.Dictionary

Introduced in R2015a

show

Class: Simulink.data.Dictionary
Package: Simulink.data

Show data dictionary in Model Explorer

Syntax

```
show(dictionaryObj)
show(dictionaryObj,openModelExplorer)
```

Description

`show(dictionaryObj)` opens Model Explorer and displays the data dictionary `dictionaryObj` as the selected tree node in the **Model Hierarchy** pane.

`show(dictionaryObj,openModelExplorer)` enables you to add the target dictionary to the **Model Hierarchy** pane without opening Model Explorer.

Tips

- Use the `hide` function to remove a data dictionary from the tree in the **Model Hierarchy** pane of Model Explorer. The dictionary does not appear in the hierarchy again until you use the `show` function or you open and view the dictionary in the Model Explorer using the interface.

Input Arguments

`dictionaryObj` — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

`openModelExplorer` — Flag to open Model Explorer

`true` (default) | `false`

Flag to open Model Explorer, specified as `true` or `false`.

Data Types: `logical`

Examples

Show Data Dictionary in Model Explorer

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Open Model Explorer and display `myDictionary_ex_API` as the selected node of the model hierarchy tree in the **Model Hierarchy** pane.

```
show(myDictionaryObj)
```

Add Data Dictionary to Model Hierarchy Tree

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Add `myDictionary_ex_API.sldd` to the model hierarchy tree without opening Model Explorer.

```
show(myDictionaryObj,false)
```

You can confirm the addition of `myDictionary_ex_API` to the model hierarchy tree by manually opening Model Explorer.

- "Store Data in Dictionary Programmatically"

See Also

hide | Simulink.data.Dictionary

Introduced in R2015a

Simulink.data.dictionary.Entry class

Package: Simulink.data.dictionary

Configure data dictionary entry

Description

An object of the `Simulink.data.dictionary.Entry` class represents one entry of a data dictionary. The object allows you to perform operations such as assign the entry a value or change the name of the entry.

Before you can create a new `Simulink.data.dictionary.Entry` object, you must create a Simulink.data.dictionary.Section object representing the data dictionary section that contains the target entry. However, once created, the `Simulink.data.dictionary.Entry` object exists independently of the `Simulink.data.dictionary.Section` object. Use the function getSection to create a `Simulink.data.dictionary.Section` object.

Construction

The functions addEntry, getEntry, and find create `Simulink.data.dictionary.Entry` objects.

Properties

`DataSource` — File name of containing data dictionary

string

File name of containing data dictionary, specified as a string. Changes you make to this property affect the represented data dictionary entry.

Example: `'myDictionary.sldd'`

Data Types: `char`

`LastModified` — Date and time of last modification

string

Date and time of last modification to entry, returned in Coordinated Universal Time (UTC) as a string. This property is read only.

`LastModifiedBy` — Name of last user to modify entry

string

Name of last user to modify entry, returned as a string. This property is read only.

`Name` — Name of entry

string

Name of entry, specified as a string. Changes you make to this property affect the represented data dictionary entry.

Data Types: `char`

`Status` — State of entry

`'New' | 'Modified' | 'Unchanged' | 'Deleted'`

State of entry, returned as `'New'`, `'Modified'`, `'Unchanged'`, or `'Deleted'`. The state is valid since the last data dictionary save. If the state is `'Deleted'`, the represented entry was deleted from its data dictionary. This property is read only.

Methods

deleteEntry	Delete data dictionary entry
discardChanges	Discard changes to data dictionary entry
find	Search in array of data dictionary entries
getValue	Return value of data dictionary entry
setValue	Set value of data dictionary entry
showChanges	Display changes made to data dictionary entry

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Add Entry to Data Dictionary and Modify its Value

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Add an entry `myEntry` with value `27` to the Design Data section of `myDictionary_ex_API.sldd`. Assign the returned `Simulink.data.dictionary.Entry` object to variable `e`.

```
e = addEntry(dDataSectObj,'myEntry',27)
```

```
e =

  Entry with properties:

             Name: 'myEntry'
            Value: 27
       DataSource: 'myDictionary_ex_API.sldd'
     LastModified: '2014-Aug-26 18:42:08.439709'
   LastModifiedBy: 'jsmith'
           Status: 'New'
```

Change the value of `myEntry` from `27` to the string `'My New Value'`.

```
setValue(e,'My New Value')
e
```

```
e =

  Entry with properties:

             Name: 'myEntry'
            Value: 'My New Value'
       DataSource: 'myDictionary_ex_API.sldd'
     LastModified: '2014-Aug-26 18:45:58.336598'
   LastModifiedBy: 'jsmith'
```

```
            Status: 'New'
```

Return Value of Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Return the value of the entry `fuelFlow` and assign the value to the variable `fuelFlowValue`.

```
fuelFlowValue = getValue(fuelFlowObj)
```

```
fuelFlowValue =

   237
```

Move Entry Within Data Dictionary Hierarchy

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`. `myDictionary_ex_API.sldd` references the data dictionary `myRefDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Create a `Simulink.data.dictionary.Entry` object representing the entry `fuelFlow`, which resides in `myDictionary_ex_API.sldd`. Assign the object to variable `e`.

```
e = getEntry(dDataSectObj,'fuelFlow')
```

```
e =

  Entry with properties:

             Name: 'fuelFlow'
            Value: 237
       DataSource: 'myDictionary_ex_API.sldd'
     LastModified: '2014-Sep-05 13:12:06.099278'
   LastModifiedBy: 'jsmith'
```

```
            Status: 'Unchanged'
```

Migrate the entry `fuelFlow` to the reference data dictionary `myRefDictionary_ex_API.sldd` by modifying the `DataSource` property of `e`.

```
e.DataSource = 'myRefDictionary_ex_API.sldd'

e =

  Entry with properties:

              Name: 'fuelFlow'
             Value: 237
        DataSource: 'myRefDictionary_ex_API.sldd'
      LastModified: '2014-Sep-05 13:12:06.099278'
    LastModifiedBy: 'jsmith'
            Status: 'Modified'
```

Because `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`, both dictionaries belong to the same dictionary hierarchy, allowing you to migrate the entry `fuelFlow` between them.

- "Store Data in Dictionary Programmatically"

See Also

Simulink.data.dictionary.Section | Simulink.data.Dictionary | getEntry

More About

- "What Is a Data Dictionary?"

Introduced in R2015a

deleteEntry

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Delete data dictionary entry

Syntax

```
deleteEntry(entryObj)
```

Description

`deleteEntry(entryObj)` deletes the data dictionary entry represented by `entryObj`, a Simulink.data.dictionary.Entry object. The represented entry no longer exists in the data dictionary that defined it.

The function sets the `Status` properties of any `Simulink.data.dictionary.Entry` objects representing the deleted entry to `'Deleted'`. You can access only the `Status` properties of the objects.

Input Arguments

`entryObj` — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the getEntry function.

Examples

Delete Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Delete the entry `fuelFlow` from the data dictionary `myDictionary_ex_API.sldd`. `myDictionary_ex_API.sldd` no longer contains the `fuelFlow` entry.

```
deleteEntry(fuelFlowObj)
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to view the contents of a data dictionary and delete entries.

See Also

addEntry | Simulink.data.dictionary.Entry

Introduced in R2015a

discardChanges

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Discard changes to data dictionary entry

Syntax

```
discardChanges(entryObj)
```

Description

`discardChanges(entryObj)` discards all changes made to the data dictionary entry `entryObj`, a Simulink.data.dictionary.Entry object, since the last time the containing data dictionary was saved using the saveChanges function. The changes to the entry are permanently lost.

Tips

- You can use the discardChanges function or the saveChanges function with an entire data dictionary, discarding or saving changes to all entries in the dictionary at once. However, only the `discardChanges` function can additionally operate on individual entries. You cannot use the `saveChanges` function to save changes to individual entries.

Input Arguments

`entryObj` — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the getEntry function.

Examples

Discard Changes to Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Change the entry `fuelFlow` by assigning it the new value `493`. Confirm a change was made by viewing the `Status` property of `fuelFlowObj`.

```
setValue(fuelFlowObj,493);
fuelFlowObj
```

```
fuelFlowObj =

  Entry with properties:

             Name: 'fuelFlow'
            Value: 493
       DataSource: 'myDictionary_ex_API.sldd'
     LastModified: '2014-Sep-05 13:14:30.661978'
   LastModifiedBy: 'jsmith'
           Status: 'Modified'
```

Discard all changes to the entry `fuelFlow`. The `Status` property of `fuelFlowObj` shows that changes were discarded.

```
discardChanges(fuelFlowObj)
fuelFlowObj
```

```
fuelFlowObj =

  Entry with properties:

             Name: 'fuelFlow'
            Value: 237
       DataSource: 'myDictionary_ex_API.sldd'
     LastModified: '2014-Sep-05 13:12:06.099278'
   LastModifiedBy: 'jsmith'
```

```
            Status: 'Unchanged'
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer and the Comparison Tool to discard changes to data dictionary entries. See “View and Revert Changes to Dictionary Entries” for more information.

See Also

saveChanges | Simulink.data.dictionary.Entry

Introduced in R2015a

find

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Search in array of data dictionary entries

Syntax

```
foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN)
foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN,
options)
```

Description

`foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN)` searches the array of data dictionary entries `targetEntries` using search criteria `PName1,PValue1,...,PNameN,PValueN`, and returns an array of entries matching the criteria. This syntax matches the search criteria with the properties of the target entries, which are `Simulink.data.dictionary.Entry` objects, but not with the properties of their values. See Simulink.data.dictionary.Entry for a list of data dictionary entry properties.

`foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN, options)` searches for data dictionary entries using additional search options. For example, you can match the search criteria with the values of the target entries.

Input Arguments

`targetEntries` — Data dictionary entries to search

array of `Simulink.data.dictionary.Entry` objects

Data dictionary entries to search, specified as an array `Simulink.data.dictionary.Entry` objects. Before you use this function, represent the target entries with `Simulink.data.dictionary.Entry` objects by using, for example, the getEntry function.

Example: `[myEntryObj1,myEntryObj2,myEntryObj3]`

`PName1,PValue1,...,PNameN,PValueN` — Search criteria

name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the target data dictionary entries. For a list of the properties of a data dictionary entry, see Simulink.data.dictionary.Entry. If you specify more than one name-value pair, the returned entries meet all of the criteria.

If you include the `'-value'` option to search in the values of the target entries, the search criteria apply to the values of the entries rather than to the entries themselves.

Example: `'LastModifiedBy','jsmith'`

Example: `'DataSource','myRefDictionary_ex_API.sldd'`

`options` — Additional search options

supported option codes

Additional search options, specified as one or more of the following supported option codes.

`'-value'`	This option causes `find` to search only in the values of the target data dictionary entries. Specify this option before any other search criteria or `options` arguments.
`'-and'`, `'-or'`, `'-xor'`, or `'-not'` logical operators	These options modify or combine multiple search criteria or other option codes.
`'-property',propertyName`	This name-value pair causes `find` to search for entries or values that have the property `propertyName` regardless of the value of the property. Specify `propertyName` as a string.
`'-class',className`	This name-value pair causes `find` to search for entries or values that are objects of the class `className`. Specify `className` as a string.
`'-isa',className`	This name-value pair causes `find` to search for entries or values that are objects of the class or of any subclass derived from the class `className`. Specify `className` as a string.

`'-regexp'`	This option allows you to use regular expressions in your search criteria. This option affects only search criteria that follow `'-regexp'`.

Example: `'-value'`

Example: `'-value','-property','CoderInfo'`

Example: `'-value','-class','Simulink.Parameter'`

Output Arguments

`foundEntries` — Data dictionary entries matching search criteria

array of `Simulink.data.dictionary.Entry` objects

Data dictionary entries matching the specified search criteria, returned as an array of `Simulink.data.dictionary.Entry` objects.

Examples

Search Data Dictionary Entry Values for Specific Class

Search in an array of data dictionary entries `myEntryObjs` for entries whose values are objects of the class `Simulink.Parameter`.

```
foundEntries = find(myEntryObjs,'-value','-class','Simulink.Parameter')
```

Search Data Dictionary Entries for Modifying User

Search in an array of data dictionary entries `myEntryObjs` for entries that were last modified by the user `jsmith`.

```
foundEntries = find(myEntryObjs,'LastModifiedBy','jsmith')
```

Search Data Dictionary Entries Using Multiple Criteria

Search in an array of data dictionary entries `myEntryObjs` for entries that were last modified by the user `jsmith` or whose names begin with `fuel`.

```
foundEntries = find(myEntryObjs,'LastModifiedBy','jsmith','-or',...
```

```
'-regexp','Name','fuel*')
```

Search Data Dictionary Entries Using Regular Expressions

Search in an array of data dictionary entries `myEntryObjs` for entries whose names begin with `Press`.

```
foundEntries = find(myEntryObjs,'-regexp','Name','Press*')
```

Search Data Dictionary Entries for Specific Value

Search in an array of data dictionary entries `myEntryObjs` for entries whose values are `273`. If you find more than one entry, store the entries in an array called `foundEntries`.

```
foundEntries = [];
for i = 1:length(myEntryObjs)
    if getValue(myEntryObjs(i)) == 237
        foundEntries = [foundEntries myEntryObjs(i)];
    end
end
```

Search Data Dictionary Entry Values for Specific Property

Search in an array of data dictionary entries `myEntryObjs` for entries whose values have a property `DataType`.

```
foundEntries = find(myEntryObjs,'-value','-property','DataType')
```

- “Store Data in Dictionary Programmatically”

See Also

find | Simulink.data.dictionary.Entry

Introduced in R2015a

getValue

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Return value of data dictionary entry

Syntax

```
entryValue = getValue(entryObj)
```

Description

`entryValue = getValue(entryObj)` returns the value of the data dictionary entry `entryObj`, a Simulink.data.dictionary.Entry object.

Input Arguments

`entryObj` — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the getEntry function.

Examples

Return Value of Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

```
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Return the value of the entry `fuelFlow` and assign the value to variable `fuelFlowValue`.

```
fuelFlowValue = getValue(fuelFlowObj)
```

```
fuelFlowValue =

   237
```

- "Store Data in Dictionary Programmatically"

See Also

setValue | Simulink.data.dictionary.Entry

Introduced in R2015a

setValue

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Set value of data dictionary entry

Syntax

```
setValue(entryObj,newValue)
```

Description

`setValue(entryObj,newValue)` assigns the value `newValue` to the data dictionary entry `entryObj`, a Simulink.data.dictionary.Entry object.

Input Arguments

`entryObj` — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the getEntry function.

`newValue` — Value to assign to data dictionary entry

MATLAB expression

Value to assign to data dictionary entry, specified as a MATLAB expression. The expression must return a value that is supported by the data dictionary section that contains the entry.

Example: `27.5`

Example: `myBaseWorkspaceVariable`

Example: `Simulink.Parameter`

Examples

Set Value of Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Set the value of the entry `fuelFlow` to `493`. Then, view the `Value` property of `fuelFlowObj` to observe the change.

```
setValue(fuelFlowObj,493)
fuelFlowObj
```

```
fuelFlowObj =

  Entry with properties:

              Name: 'fuelFlow'
             Value: 493
        DataSource: 'myDictionary_ex_API.sldd'
      LastModified: '2014-Sep-05 13:37:22.161124'
    LastModifiedBy: 'jsmith'
            Status: 'Modified'
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to view and change the values of data dictionary entries.

See Also

getValue | Simulink.data.dictionary.Entry

Introduced in R2015a

showChanges

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Display changes made to data dictionary entry

Syntax

```
showChanges(entryObj)
```

Description

`showChanges(entryObj)` opens the Comparison Tool to show changes made to the data dictionary entry `entryObj`, a Simulink.data.dictionary.Entry object. The Comparison Tool displays the properties of `entryObj` as they were when the data dictionary was last saved and as they were when the `showChanges` function was called.

Input Arguments

`entryObj` — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the getEntry function.

Examples

View Unsaved Changes to Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Make a change to the entry `fuelFlow` by assigning it the new value `494`.

```
setValue(fuelFlowObj,494);
```

Observe the unsaved change to the entry `fuelFlow`. The Comparison Tool opens and compares side by side the current state of the entry with its most recently saved state.

```
showChanges(fuelFlowObj)
```

- "Store Data in Dictionary Programmatically"

Alternatives

You can use Model Explorer and the Comparison Tool to view changes to data dictionary entries. See "View and Revert Changes to Dictionary Entries" for more information.

See Also

discardChanges | Simulink.data.dictionary.Entry

Introduced in R2015a

Simulink.data.dictionary.EnumTypeDefinition class

Package: Simulink.data.dictionary

Store enumerated type definition in data dictionary

Description

An object of the `Simulink.data.dictionary.EnumTypeDefinition` class defines an enumerated data type in a data dictionary. You store the object in a data dictionary entry so models linked to the dictionary can use the enumerated type definition.

In the MATLAB base workspace, objects of this class retain information about an enumerated type but do not define the type for use by other variables or by models.

Construction

When you use the function importEnumTypes to import the definitions of enumerated types to a data dictionary, Simulink creates a `Simulink.data.dictionary.EnumTypeDefinition` object in the dictionary for each imported definition. The dictionary stores each object in an individual entry.

The constructor `Simulink.data.dictionary.EnumTypeDefinition` creates an instance of this class with default property values and a single enumeration member that has underlying integer value `0`.

Properties

`AddClassNameToEnumNames` — Flag to control enumeration identifiers in generated code

`false` (default) | `true`

Flag to prefix enumerations with the class name in generated code, specified as `true` or `false`.

If you specify `true`, when you generate code the identifier of each enumeration member begins with the name of the enumeration class. For example, an enumeration class

`LEDcolor` with enumeration members `GREEN` and `RED` defines the enumeration members in generated code as `LEDcolor_GREEN` and `LEDcolor_RED`.

Data Types: `logical`

`DataScope` — Flag to control data type definition in generated code

`'Auto'` (default) | `'Imported'` | `'Exported'`

Flag to control data type definition in generated code, specified as `'Auto'`, `'Imported'`, or `'Exported'`. The table describes the behavior of generated code for each value.

Value	Action
`Auto` (default)	If you do not specify the property `Headerfile`, export the data type definition to *`model`*`_types.h`, where *`model`* is the model name. If you specify `Headerfile`, import the data type definition from the specified header file.
`Exported`	Export the data type definition to a separate header file. If you do not specify the property `Headerfile`, the header file name defaults to *`type`*`.h`, where *`type`* is the data type name.
`Imported`	Import the data type definition from a separate header file. If you do not specify the property `Headerfile`, the header file name defaults to *`type`*`.h`, where *`type`* is the data type name.

`DefaultValue` — Default enumeration member

`''` (default) | string

Default enumeration member, specified as a string. Specify `DefaultValue` as the name of an enumeration member you have already defined.

When you create a `Simulink.data.dictionary.EnumTypeDefinition` object, `DefaultValue` is an empty string, `''`, and Simulink uses the first enumeration member as the default member.

Example: `'enumMember1'`

`Description` — Description of enumerated data type in generated code

`''` (default) | string

Description of the enumerated data type, specified as a string. Use this property to explain the purpose of the type in generated code.

Example: `'Two possible colors of LED indicator: GREEN and RED.'`

Data Types: `char`

`HeaderFile` — Name of header file defining enumerated data type in generated code

`''` (default) | string

Name of the header file that defines the enumerated data type in generated code, specified as a string. Use a `.h` extension to specify the file name.

If you do not specify `HeaderFile`, generated code uses a default header file name that depends on the value of the `DataScope` property .

Example: `'myTypeIncludeFile.h'`

Data Types: `char`

`StorageType` — Data type of underlying integer values

`''` (default) | string

Data type of the integer values underlying the enumeration members, specified as a string. Generated code stores the underlying integer values using the data type you specify.

You can specify one of these supported integer types:

- `'int8'`
- `'int16'`
- `'int32'`
- `'uint8'`

- `'uint16'`

To store the underlying integer values in generated code using the native integer type of the target hardware, specify `StorageType` as an empty string, `''`, which is the default value.

Example: `'int16'`

`''`

Methods

appendEnumeral	Add enumeration member to enumerated data type definition in data dictionary
removeEnumeral	Remove enumeration member from enumerated data type definition in data dictionary

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member `enum1` with underlying integer value `0`.

```
myColors = Simulink.data.dictionary.EnumTypeDefinition
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      enum1
```

Add some enumeration members to the definition of the type.

```
appendEnumeral(myColors,'Orange',1,'')
appendEnumeral(myColors,'Black',2,'')
appendEnumeral(myColors,'Cyan',3,'')
myColors
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      enum1
      Orange
      Black
      Cyan
```

Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index `1`.

```
removeEnumeral(myColors,1)
myColors
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      Orange
      Black
      Cyan
```

Customize the enumerated type by configuring the properties of the object representing it.

```
myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the object that defines the enumerated type `myColors` to the dictionary.

```
importFromBaseWorkspace(myDictionaryObj,'varList',{'myColors'});
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to add and modify enumerated data types stored in a data dictionary.

See Also

Simulink.data.Dictionary

More About

- "Use Enumerated Data in Simulink Models"

Introduced in R2015a

appendEnumeral

Class: Simulink.data.dictionary.EnumTypeDefinition
Package: Simulink.data.dictionary

Add enumeration member to enumerated data type definition in data dictionary

Syntax

```
appendEnumeral(typeObj,memberName,memberValue,memberDesc)
```

Description

`appendEnumeral(typeObj,memberName,memberValue,memberDesc)` adds an enumeration member to the enumerated type definition stored by `typeObj`, a Simulink.data.dictionary.EnumTypeDefinition object.

Input Arguments

`typeObj` — Target enumerated type definition

`Simulink.data.dictionary.EnumTypeDefinition` object

Target enumerated type definition, specified as a `Simulink.data.dictionary.EnumTypeDefinition` object.

`memberName` — Name of new enumeration member

string

Name of the new enumeration member, specified as a string.

Example: `'myNewEnumMember'`

Data Types: `char`

`memberValue` — Integer value underlying new enumeration member

integer

Integer value underlying the new enumeration member, specified as an integer.

The definition of the enumeration class determines the integer data type used in generated code to store the underlying values of enumeration members.

Example: `3`

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `double`

`memberDesc` — Description of new enumeration member

string

Description of the new enumeration member, specified as a string.

If you do not want to supply a description for the enumeration member, use an empty string.

Example: `'Enumeration member number 1.'`

Example: `''`

Data Types: `char`

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member `enum1` with underlying integer value `0`.

```
myColors = Simulink.data.dictionary.EnumTypeDefinition
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      enum1
```

Add some enumeration members to the definition of the type.

```
appendEnumeral(myColors,'Orange',1,'')
appendEnumeral(myColors,'Black',2,'')
appendEnumeral(myColors,'Cyan',3,'')
myColors
```

```
myColors =
```

```
   Simulink.data.dictionary.EnumTypeDefinition
      enum1
      Orange
      Black
      Cyan
```

Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index `1`.

```
removeEnumeral(myColors,1)
myColors
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      Orange
      Black
      Cyan
```

Customize the enumerated type by configuring the properties of the object representing it.

```
myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the object that defines the enumerated type `myColors` to the dictionary.

```
importFromBaseWorkspace(myDictionaryObj,'varList',{'myColors'});
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to add enumeration members to the enumerated data type represented by a `Simulink.data.dictionary.EnumTypeDefinition` object.

See Also

Simulink.data.dictionary.EnumTypeDefinition |
Simulink.data.dictionary.EnumTypeDefinition.removeEnumeral

More About

- "Use Enumerated Data in Simulink Models"

Introduced in R2015a

removeEnumeral

Class: Simulink.data.dictionary.EnumTypeDefinition
Package: Simulink.data.dictionary

Remove enumeration member from enumerated data type definition in data dictionary

Syntax

```
removeEnumeral(typeObj,memberNum)
```

Description

`removeEnumeral(typeObj,memberNum)` removes an enumeration member from the enumerated type definition stored by `typeObj`, a Simulink.data.dictionary.EnumTypeDefinition object.

Input Arguments

`typeObj` — Target enumerated type definition

`Simulink.data.dictionary.EnumTypeDefinition` object

Target enumerated type definition, specified as a `Simulink.data.dictionary.EnumTypeDefinition` object.

`memberNum` — Index of target enumeration member

integer

Index of target enumeration member, specified as an integer.

The first enumeration member in an enumerated type definition has index `1`. For example, suppose an enumerated type `BasicColors` has this definition:

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      Orange
      Black
```

```
Cyan
```

To remove the enumeration member `Black`, specify `memberNum` as `2`. To remove the enumeration member `Cyan`, specify `3`.

Do not specify `memberNum` using the integer value underlying an enumeration member. The integer value underlying the member is not equivalent to the index of the member.

Example: `3`

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `double`

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member `enum1` with underlying integer value `0`.

```
myColors = Simulink.data.dictionary.EnumTypeDefinition
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      enum1
```

Add some enumeration members to the definition of the type.

```
appendEnumeral(myColors,'Orange',1,'')
appendEnumeral(myColors,'Black',2,'')
appendEnumeral(myColors,'Cyan',3,'')
myColors
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      enum1
      Orange
      Black
      Cyan
```

Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index `1`.

```
removeEnumeral(myColors,1)
myColors
```

```
myColors =

   Simulink.data.dictionary.EnumTypeDefinition
      Orange
      Black
      Cyan
```

Customize the enumerated type by configuring the properties of the object representing it.

```
myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the object that defines the enumerated type `myColors` to the dictionary.

```
importFromBaseWorkspace(myDictionaryObj,'varList',{'myColors'});
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to remove enumeration members from the enumerated data type represented by a `Simulink.data.dictionary.EnumTypeDefinition` object.

See Also

Simulink.data.dictionary.EnumTypeDefinition |
Simulink.data.dictionary.EnumTypeDefinition.appendEnumeral

More About

- “Use Enumerated Data in Simulink Models”

Introduced in R2015a

Simulink.data.dictionary.Section class

Package: Simulink.data.dictionary

Configure data dictionary section

Description

An object of the `Simulink.data.dictionary.Section` class represents one section of a data dictionary, such as Design Data or Configurations. The object allows you to perform operations on the section such as add or delete entries and import data from files.

Before you can create a `Simulink.data.dictionary.Section` object, you must create a Simulink.data.Dictionary object representing the target data dictionary. Once created, the `Simulink.data.dictionary.Section` object exists independently of the `Simulink.data.Dictionary` object.

Construction

The function getSection creates a `Simulink.data.dictionary.Section` object.

Properties

`Name` — Name of data dictionary section

string

Name of data dictionary section, returned as a string. This property is read only.

Methods

addEntry	Add new entry to data dictionary section
assignin	Assign value to data dictionary entry
deleteEntry	Delete data dictionary entry

evalin	Evaluate MATLAB expression in data dictionary section
exist	Check existence of data dictionary entry
exportToFile	Export data dictionary entries from section to MAT-file or MATLAB file
find	Search in data dictionary section
getEntry	Create `Simulink.data.dictionary.Entry` object to represent data dictionary entry
importFromFile	Import variables from MAT-file or MATLAB file to data dictionary section

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create New Data Dictionary Section Object

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
dDataSectObj = getSection(myDictionaryObj,'Design Data')
```

```
dDataSectObj =

  Section with properties:

    Name: 'Design Data'
```

- “Store Data in Dictionary Programmatically”

See Also

Simulink.data.Dictionary | getSection

More About

- "What Is a Data Dictionary?"

Introduced in R2015a

addEntry

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Add new entry to data dictionary section

Syntax

```
addEntry(sectionObj,entryName,entryValue)
entryObj = addEntry(sectionObj,entryName,entryValue)
```

Description

`addEntry(sectionObj,entryName,entryValue)` adds an entry, with name `entryName` and value `entryValue`, to the data dictionary section `sectionObj`, a Simulink.data.dictionary.Section object.

`entryObj = addEntry(sectionObj,entryName,entryValue)` returns a Simulink.data.dictionary.Entry object representing the newly added data dictionary entry.

Tips

- `addEntry` returns an error if the entry name you specify with `entryName` is already the name of an entry in the target data dictionary section or in the same section of any referenced dictionaries.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`entryName` — Name of new data dictionary entry

string

Name of new data dictionary entry, specified as a string.

Example: `'myNewEntry'`

Data Types: `char`

`entryValue` — Value of new data dictionary entry

MATLAB expression

Value of new data dictionary entry, specified as a MATLAB expression that returns any valid data dictionary content.

Example: `27.5`

Example: `myBaseWorkspaceVariable`

Example: `Simulink.Parameter`

Examples

Add Entry to Design Data Section of Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Add an entry to the Design Data section of `myDictionary_ex_API.sldd` an entry `myNewEntry` with value `237`.

```
addEntry(dDataSectObj,'myNewEntry',237)
```

Add New Simulink.Parameter Object to Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Add an entry to the Design Data section of `myDictionary_ex_API.sldd`. Name the new entry `myNewParam` and assign a `Simulink.Parameter` object to the value.

```
addEntry(dDataSectObj,'myNewParam',Simulink.Parameter)
```

The expression `Simulink.Parameter` constructs a new `Simulink.Parameter` object, and the `addEntry` function assigns the object to the value of the new data dictionary entry `myNewParam`.

- "Store Data in Dictionary Programmatically"

Alternatives

You can use Model Explorer to add entries to a data dictionary in the same way you can use it to add variables to a model workspace or the base workspace.

See Also

assignin | Simulink.data.dictionary.Entry | Simulink.data.dictionary.Section

Introduced in R2015a

assignin

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Assign value to data dictionary entry

Syntax

```
assignin(sectionObj,entryName,entryValue)
```

Description

`assignin(sectionObj,entryName,entryValue)` assigns the value `entryValue` to the data dictionary entry `entryName` in the data dictionary section `sectionObj`, a Simulink.data.dictionary.Section object. If an entry with the specified name is not in the target section, `assignin` creates the entry with the specified name and value.

If an entry with the name specified by input argument `entryName` is not defined in the target data dictionary section but is defined in a referenced dictionary, `assignin` does not create a new entry in the target section but operates on the entry in the referenced dictionary.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`entryName` — Name of target data dictionary entry

string

Name of target data dictionary entry, specified as a string. If a matching entry does not already exist, the functions creates a new entry using the specified name.

Example: `'myEntry'`

Data Types: `char`

`entryValue` — Value to assign to data dictionary entry

MATLAB expression

Value to assign to data dictionary entry, specified as a MATLAB expression that returns any valid data dictionary content.

Example: `27.5`

Example: `myBaseWorkspaceVariable`

Example: `Simulink.Parameter`

Examples

Assign Value to Data Dictionary Entry

Assign a value to a data dictionary entry by operating on a `Simulink.data.dictionary.Section` object.

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Assign the value `237` to an entry `myAssignedEntry` in the data dictionary `myDictionary_ex_API.sldd`. If an entry named `myAssignedEntry` is not in `myDictionary_ex_API.sldd`, create it.

```
assignin(dDataSectObj,'myAssignedEntry',237)
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to view and change the values of data dictionary entries.

See Also

setValue | Simulink.data.dictionary.Section

Introduced in R2015a

deleteEntry

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Delete data dictionary entry

Syntax

```
deleteEntry(sectionObj,entryName)
deleteEntry(sectionObj,entryName,'DataSource',dictionaryName)
```

Description

`deleteEntry(sectionObj,entryName)` deletes a data dictionary entry `entryName` from the data dictionary section `sectionObj`, a Simulink.data.dictionary.Section object. If there are multiple entries with the specified name in a hierarchy of reference dictionaries, the function deletes all the entries. If you represent a data dictionary entry with one or more Simulink.data.dictionary.Entry objects and later delete the entry using the `deleteEntry` function, the objects remain with their `Status` property set to `'Deleted'`.

`deleteEntry(sectionObj,entryName,'DataSource',dictionaryName)` deletes an entry that is defined in the data dictionary `DictionaryName`. Use this syntax to uniquely identify an entry that is defined more than once in a hierarchy of referenced data dictionaries.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`entryName` — Name of target data dictionary entry

string

Name of target data dictionary entry, specified as a string.

Example: `'myEntry'`

Data Types: `char`

`dictionaryName` — Name of data dictionary that defines target entry

string

File name of data dictionary that defines the target entry, specified as a string including the `.sldd` extension.

Example: `'mySubDictionary_ex_API.sldd'`

Data Types: `char`

Examples

Delete Entry from Data Dictionary Section

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`. The Design Data section of `myDictionary_ex_API.sldd` already contains an entry named `fuelFlow`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Delete the entry `fuelFlow` from the data dictionary `myDictionary_ex_API.sldd`. `myDictionary_ex_API.sldd` no longer contains the `fuelFlow` entry.

```
deleteEntry(dDataSectObj,'fuelFlow')
```

Delete Entry from Reference Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Delete the entry `myRefEntry` from the data dictionary `myRefDictionary_ex_API.sldd`. `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`, and `myRefDictionary_ex_API.sldd` defines an entry`myRefEntry`.

```
deleteEntry(dDataSectObj,'myRefEntry','DataSource',...
'myRefDictionary_ex_API.sldd')
```

- "Store Data in Dictionary Programmatically"

Alternatives

You can use the Model Explorer window to delete entries from a data dictionary in the same way you can delete variables from a model workspace or the base workspace.

See Also

addEntry | Simulink.data.dictionary.Entry | Simulink.data.dictionary.Section

Introduced in R2015a

evalin

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Evaluate MATLAB expression in data dictionary section

Syntax

```
returnValue = evalin(sectionObj,expression)
```

Description

`returnValue = evalin(sectionObj,expression)` evaluates a MATLAB expression in the data dictionary section `sectionObj` and returns the values returned by `expression`.

Tips

- `evalin` allows you to treat a data dictionary section as a MATLAB workspace. You can think of entries contained in the section as workspace variables you can manipulate with MATLAB expressions.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`expression` — MATLAB expression to evaluate

string

MATLAB expression to evaluate, specified as a string.

Example: `'a = 5.3'`

Example: `'whos'`

Example: `'CurrentSpeed.Value = 290.73'`

Data Types: `char`

Examples

List All Entries in Data Dictionary Section

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Execute the `whos` command in the Design Data section of `myDictionary_ex_API.sldd`.

```
evalin(dDataSectObj,'whos')
```

```
  Name                Size            Bytes  Class                Attributes

  fuelFlow            1x1                 8  double
  myRefEntry          1x1                 1  logical
  parameterGain37     1x1               112  Simulink.Parameter
```

- "Store Data in Dictionary Programmatically"

See Also

Simulink.data.dictionary.Section | `Simulink.data.evalinGlobal`

Introduced in R2015a

exist

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Check existence of data dictionary entry

Syntax

```
doesExist = exist(sectionObj,entryName)
```

Description

`doesExist = exist(sectionObj,entryName)` determines if the data dictionary section `sectionObj` contains an entry by the name of `entryName` and returns an indication.

Tips

- `exist` also determines if a matching entry exists in the same section of any referenced data dictionaries. For example, if `sectionObj` represents the Design Data section of a data dictionary `myDictionary_ex_API.sldd`, `exist` searches the Design Data section of `myDictionary_ex_API.sldd` and the Design Data sections of any dictionaries referenced by `myDictionary_ex_API.sldd`.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`entryName` — Name of target entry

string

Name of target entry, specified as a string.

Example: `'myEntry'`

Data Types: `char`

Output Arguments

`doesExist` — Indication of entry existence

`0 | 1`

Indication of entry existence, returned as `0` if false and `1` if true.

Examples

Determine if Data Dictionary Entry Exists

Determine if an entry exists in a data dictionary by searching for the name of the entry

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Determine if an entry `fuelFlow` exists in the Design Data section of `myDictionary_ex_API.sldd`.

```
exist(dDataSectObj,'fuelFlow')
```

```
ans =

     1
```

Determine if an entry `myEntry` exists in the Design Data section of `myDictionary_ex_API.sldd`.

```
exist(dDataSectObj,'myEntry')
```

```
ans =
```

0

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to search a data dictionary for an entry.

See Also

find | Simulink.data.dictionary.Section | `Simulink.data.existsInGlobal`

Introduced in R2015a

exportToFile

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Export data dictionary entries from section to MAT-file or MATLAB file

Syntax

```
exportToFile(sectionObj,fileName)
```

Description

`exportToFile(sectionObj,fileName)` exports to a MAT or MATLAB file all the values of the entries contained in the data dictionary section `sectionObj`, a Simulink.data.dictionary.Section object. `exportToFile` exports the values of all entries, including those defined in referenced dictionaries.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`fileName` — Name of MAT or MATLAB file

string

Name of target MAT or MATLAB file, specified as a string. `exportToFile` supplies a file extension `.mat` if you do not specify an extension.

Example: `'myNewFile.mat'`

Example: `'myNewFile.m'`

Data Types: `char`

Examples

Export Data Dictionary Entries to MAT or MATLAB Files

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`. Represent the Configurations section of `myDictionary_ex_API.sldd` with an object named `configSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
configSectObj = getSection(myDictionaryObj,'Configurations');
```

Export the entries from the Design Data section of `myDictionary_ex_API.sldd` to a MATLAB file in your current working folder.

```
exportToFile(dDataSectObj,'myDictionaryDesignData.m')
```

Export the entries from the Configurations section of `myDictionary_ex_API.sldd` to a MAT-file in your current working folder.

```
exportToFile(configSectObj,'myDictionaryConfigurations.mat')
```

```
Exported 1 entries from scope 'Configurations'
to MAT-file myDictionaryConfigurations.mat.
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to export data dictionary entries to a file. See “Export Design Data from Dictionary” for more information.

See Also

importFromFile | Simulink.data.dictionary.Section

Introduced in R2015a

find

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Search in data dictionary section

Syntax

```
foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN)
foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN,
options)
```

Description

`foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN)` searches the data dictionary section `sectionObj` using search criteria `PName1,PValue1,...,PNameN,PValueN`, and returns an array of matching entries that were found in the target section. This syntax matches the search criteria with the properties of the entries in the target section but not with the properties of their values. See Simulink.data.dictionary.Entry for a list of data dictionary entry properties.

`foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN, options)` searches for data dictionary entries using additional search options. For example, you can match the search criteria with the values of the entries in the target section.

Input Arguments

`sectionObj` — Data dictionary section to search

`Simulink.data.dictionary.Section` object

Data dictionary section to search, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`PName1,PValue1,...,PNameN,PValueN` — Search criteria

name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the entries in the target data dictionary section. For a list of the properties of a data dictionary entry, see Simulink.data.dictionary.Entry. If you specify more than one name-value pair, the returned entries meet all of the criteria.

If you include the `'-value'` option to search in the values of the entries, the search criteria apply to the values of the entries rather than to the entries themselves.

Example: `'LastModifiedBy','jsmith'`

Example: `'DataSource','myRefDictionary_ex_API.sldd'`

`options` — Additional search options

supported option codes

Additional search options, specified as one or more of the following supported option codes.

`'-value'`	This option causes `find` to search only in the values of the entries in the target data dictionary section. Specify this option before any other search criteria or `options` arguments.
`'-and'`, `'-or'`, `'-xor'`, `'-not'` logical operators	These options modify or combine multiple search criteria or other option codes.
`'-property',propertyName`	This name-value pair causes `find` to search for entries or values that have the property `propertyName` regardless of the value of the property. Specify `propertyName` as a string.
`'-class',className`	This name-value pair causes `find` to search for entries or values that are objects of the class `className`. Specify `className` as a string.
`'-isa',className`	This name-value pair causes `find` to search for entries or values that are objects of the class or of any subclass derived from the class `className`. Specify `className` as a string.
`'-regexp'`	This option allows you to use regular expressions in your search criteria. This option affects only search criteria that follow `'-regexp'`.

Example: `'-value'`

Example: `'-value','-property','CoderInfo'`

Example: `'-value','-class','Simulink.Parameter'`

Output Arguments

`foundEntries` — Data dictionary entries matching search criteria

array of `Simulink.data.dictionary.Entry` objects

Data dictionary entries matching the specified search criteria, returned as an array of Simulink.data.dictionary.Entry objects.

Examples

Return Array of All Entries in Data Dictionary Section

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Return all of the entries stored in the Design Data section of the data dictionary `myDictionary_ex_API.sldd`.

```
allEntries = find(dDataSectObj)
```

Search Data Dictionary Section for Specific Class

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries whose values are objects of the `Simulink.Parameter` class.

```
foundEntries = find(dDataSectObj,'-value','-class','Simulink.Parameter')
```

Search Data Dictionary Section for Modifying User

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries that were last modified by the user `jsmith`.

```
foundEntries = find(dDataSectObj,'LastModifiedBy','jsmith')
```

Search Data Dictionary Section Using Multiple Criteria

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries that were last modified by the user `jsmith` or whose names begin with `fuel`.

```
foundEntries = find(dDataSectObj,'LastModifiedBy','jsmith','-or',...
'-regexp','Name','fuel*')
```

Search Data Dictionary Section Using Regular Expressions

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries whose names begin with `fuel`.

```
foundEntries = find(dDataSectObj,'-regexp','Name','fuel*')
```

Search Data Dictionary Section for Specific Value

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Return all of the entries stored in the Design Data section of the data dictionary `myDictionary_ex_API.sldd`.

```
allEntries = find(dDataSectObj);
```

Find the entries with value `237`. If you find more than one entry, store the entries in an array called `foundEntries`.

```
foundEntries = [];
for i = 1:length(allEntries)
    if getValue(allEntries(i)) == 237
        foundEntries = [foundEntries allEntries(i)];
    end
end
```

Search Data Dictionary Section for Specific Property

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries whose values have a property `DataType`.

```
foundEntries = find(dDataSectObj,'-value','-property','DataType')
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to search a data dictionary for entries using arbitrary criteria.

See Also

Simulink.data.dictionary.Section | Simulink.data.dictionary.Entry | exist | find

Introduced in R2015a

getEntry

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Create `Simulink.data.dictionary.Entry` object to represent data dictionary entry

Syntax

```
entryObj = getEntry(sectionObj,entryName)
entryObj = getEntry(sectionObj,
entryName,'DataSource',dictionaryName)
```

Description

`entryObj = getEntry(sectionObj,entryName)` returns an array of Simulink.data.dictionary.Entry objects representing data dictionary entries `entryName` found in the data dictionary section `sectionObj`, a Simulinkdata.dictionary.Section object. `getEntry` returns multiple objects if multiple entries have the specified name in a reference hierarchy of data dictionaries.

`entryObj = getEntry(sectionObj,
entryName,'DataSource',dictionaryName)` returns an object representing a data dictionary entry that is defined in the data dictionary `dictionaryName`. Use this syntax to uniquely identify an entry that is defined more than once in a hierarchy of referenced data dictionaries.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`entryName` — Name of target data dictionary entry

string

Name of target data dictionary entry, specified as a string.

Example: `'myEntry'`

Data Types: `char`

`dictionaryName` — Name of data dictionary containing target entry

string

File name of data dictionary containing the target entry, specified as a string including the `.sldd` extension.

Example: `'mySubDictionary_ex_API.sldd'`

Data Types: `char`

Output Arguments

`entryObj` — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, returned as one or more `Simulink.data.dictionary.Entry` objects.

Examples

Set Value of Data Dictionary Entry

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Set the value of the entry `fuelFlow` to `493`.

```
setValue(fuelFlowObj,493)
```

Set Value of Entry in Reference Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Represent the data dictionary entry `myRefEntry` with a `Simulink.data.dictionary.Entry` object named `refEntryObj`. `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`, and `myRefDictionary_ex_API.sldd` defines an entry `myRefEntry`.

```
refEntryObj = getEntry(dDataSectObj,'myRefEntry','DataSource',...
'myRefDictionary_ex_API.sldd');
```

Set the value of the entry `myRefEntry` to `493`.

```
setValue(refEntryObj,493)
```

- “Store Data in Dictionary Programmatically”

See Also

addEntry | getValue | Simulink.data.dictionary.Entry | Simulink.data.dictionary.Section

Introduced in R2015a

importFromFile

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Import variables from MAT-file or MATLAB file to data dictionary section

Syntax

```
importedVars = importFromFile(sectionObj,fileName)
importedVars = importFromFile(sectionObj,
fileName,'existingVarsAction',existAction)
[importedVars,existingVars] = importFromFile( ___ )
```

Description

`importedVars = importFromFile(sectionObj,fileName)` imports variables defined in the MAT-file or MATLAB file `fileName` to the data dictionary section `sectionObj` without overwriting any variables that are already in the target section. If any variables are already in the target section, the function displays a warning and a list in the MATLAB Command Window. This syntax returns a list of variables that were successfully imported. A variable is considered successfully imported only if `importFromFile` assigns the value of the variable to the corresponding entry in the target data dictionary.

`importedVars = importFromFile(sectionObj, fileName,'existingVarsAction',existAction)` imports variables that are already in the target section by taking a specified action `existAction`. For example, you can choose to use the variable values in the target file to overwrite the corresponding values in the target section.

`[importedVars,existingVars] = importFromFile( ___ )` returns a list of variables in the target section that were not overwritten. Use this syntax if `existingVarsAction` is set to `'none'`, the default value, which prevents existing dictionary entries from being overwritten.

Tips

- `importFromFile` can import MATLAB variables created from enumerated data types but cannot import the definitions of the enumerated types. Use the importEnumTypes function to import enumerated data type definitions to a data dictionary. If you import variables of enumerated data types to a data dictionary but do not import the enumerated type definitions, the dictionary is less portable and might not function properly if used by someone else.

Input Arguments

`sectionObj` — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

`fileName` — Name of MAT or MATLAB file

string

Name of target MAT or MATLAB file, specified as a string. `importFromFile` automatically supplies a file extension `.mat` if you do not specify an extension.

Example: `'myFile.mat'`

Example: `'myFile.m'`

Data Types: `char`

`existAction` — Action to take for existing dictionary variables

`'none'` (default) | `'overwrite'` | `'error'`

Action to take for existing dictionary variables, specified as `'none'`, `'overwrite'`, or `'error'`.

If you specify `'none'`, `importFromFile` attempts to import target variables but does not import or make any changes to variables that are already in the data dictionary section.

If you specify `'overwrite'`, `importFromFile` imports all target variables and overwrites any variables that are already in the data dictionary section.

If you specify `'error'`, `importFromFile` returns an error, without importing any variables, if any target variables are already in the data dictionary section.

Example: `'overwrite'`

Data Types: `char`

Output Arguments

`importedVars` — Successfully imported variables

cell array of strings

Names of successfully imported variables, returned as a cell array of strings. A variable is considered successfully imported only if `importFromFile` assigns its value to the corresponding entry in the target data dictionary.

`existingVars` — Variables that were not imported

cell array of strings

Names of target variables that were not imported due to their existence in the target data dictionary, returned as a cell array of strings. `existingVars` has content only if `existAction` is set to `'none'`, which is also the default. In that case `importFromFile` imports only variables that are not already in the target data dictionary.

Examples

Import to Data Dictionary from File

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Import all variables contained in the file `myData_ex_API.m` to the data dictionary and return a list of successfully imported variables. If any variables are already in `myDictionary_ex_API.sldd`, `importFromFile` returns a warning and a list of the affected variables.

```
importFromFile(dDataSectObj,'myData_ex_API.m')
```

```
Warning: The following variables were not imported because
they already exist in the dictionary:
   fuelFlow

ans =

    'myFirstEntry'
    'mySecondEntry'
    'myThirdEntry'
```

Import Variables from File and Overwrite Conflicts

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Import all variables contained in the file `myData_ex_API.m` to the data dictionary, overwrite any variables that are already in the dictionary, and return a list of successfully imported variables.

```
importFromFile(dDataSectObj,'myData_ex_API.m','existingVarsAction','overwrite')
```

```
ans =

    'fuelFlow'
    'myFirstEntry'
    'mySecondEntry'
    'myThirdEntry'
```

Return Variables Not Imported to Data Dictionary from File

Return a list of variables that are not imported from a file because they are already in the target data dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Import all variables contained in the file `myData_ex_API.m` to the data dictionary. Specify names for the output arguments of `importFromFile` to return the names of successfully and unsuccessfully imported variables.

```
[importedVars,existingVars] = importFromFile(dDataSectObj,'myData_ex_API.m')

importedVars =

    'myFirstEntry'
    'mySecondEntry'
    'myThirdEntry'


existingVars =

    'fuelFlow'
```

`importFromFile` does not import the variable `fuelflow` because it is already in the target data dictionary.

- "Store Data in Dictionary Programmatically"

Alternatives

You can use the Model Explorer to import variables to a data dictionary from a file. See "Import Data to Dictionary from File" for more information.

See Also

exportToFile | importEnumTypes | Simulink.data.dictionary.Section

Introduced in R2015a

Simulink.DualScaledParameter

Specify name, value, units, and other properties of Simulink dual-scaled parameter

Description

This class extends the `Simulink.Parameter` class so that you can define an object that stores two scaled values of the same physical value. For example, for temperature measurement, you can store a Fahrenheit scale and a Celsius scale with conversion defined by a computational method that you provide. Given one scaled value, the `Simulink.DualScaledParameter` computes the other scaled value using the computational method.

A dual-scaled parameter has:

- A calibration value. The value that you prefer to use.
- A main value. The real-world value that Simulink uses.
- An internal stored integer value. The value that is used in the embedded code.

You can use `Simulink.DualScaledParameter` objects in your model for both simulation and code generation. The parameter computes the internal value before code generation via the computational method. This offline computation results in leaner generated code.

If you provide the calibration value, the parameter computes the main value using the computational method. This method can be a first-order rational function.

$$y = \frac{ax + b}{cx + d}$$

- `x` is the calibration value.
- `y` is the main value.
- `a` and `b` are the coefficients of the CalToMain compute numerator.
- `c` and `d` are the coefficients of the CalToMain compute denominator.

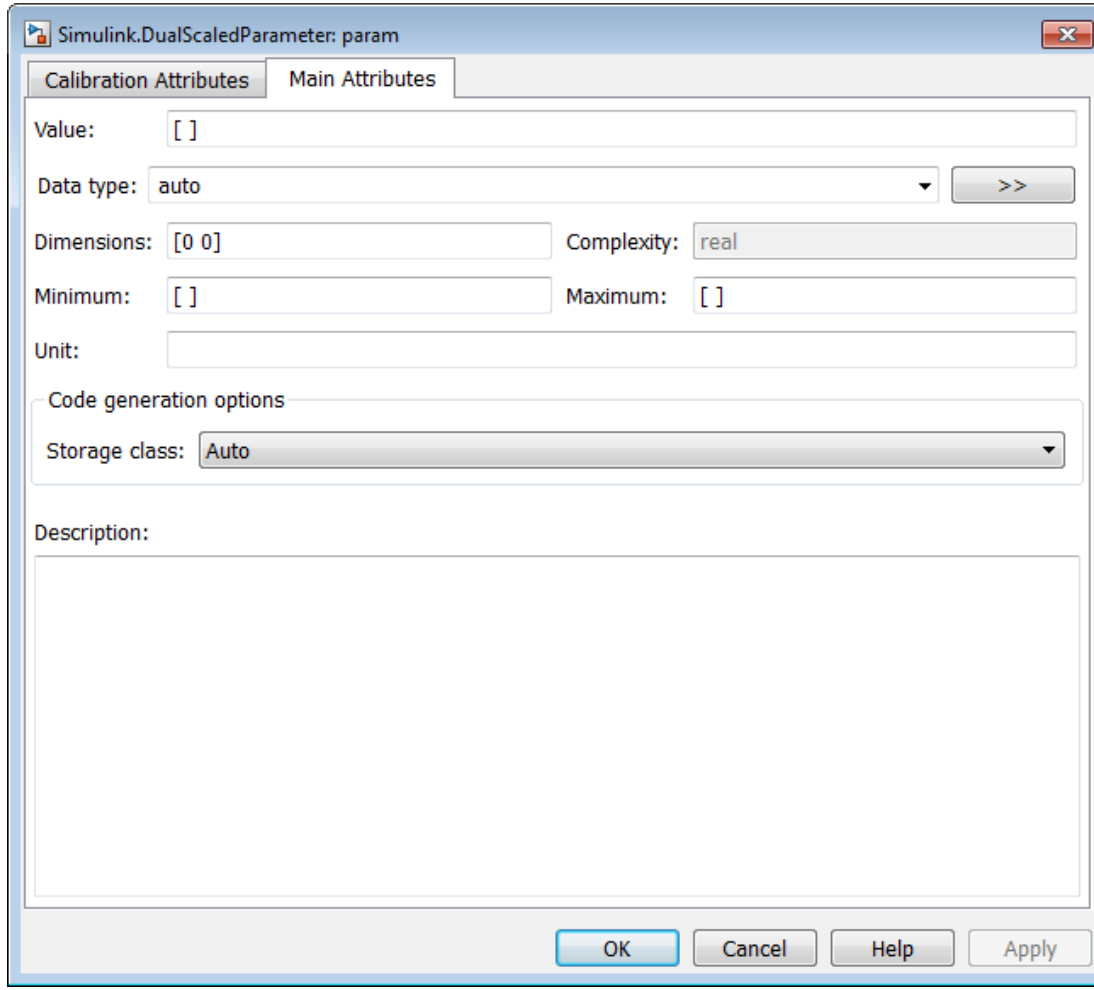
If you provide the calibration minimum and maximum values, the parameter computes minimum and maximum values of the main value. Simulink performs range checking of parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum and maximum values and data type.

You can use the Simulink.DualScaledParameter dialog box to define a `Simulink.DualScaledParameter` object. To open the dialog box:

1. In the Model Explorer, select the base workspace or a model workspace and select **Add** > **Add Custom**.
2. In the Model Explorer — Select Object dialog box, set **Object class** to `Simulink.DualScaledParameter`.

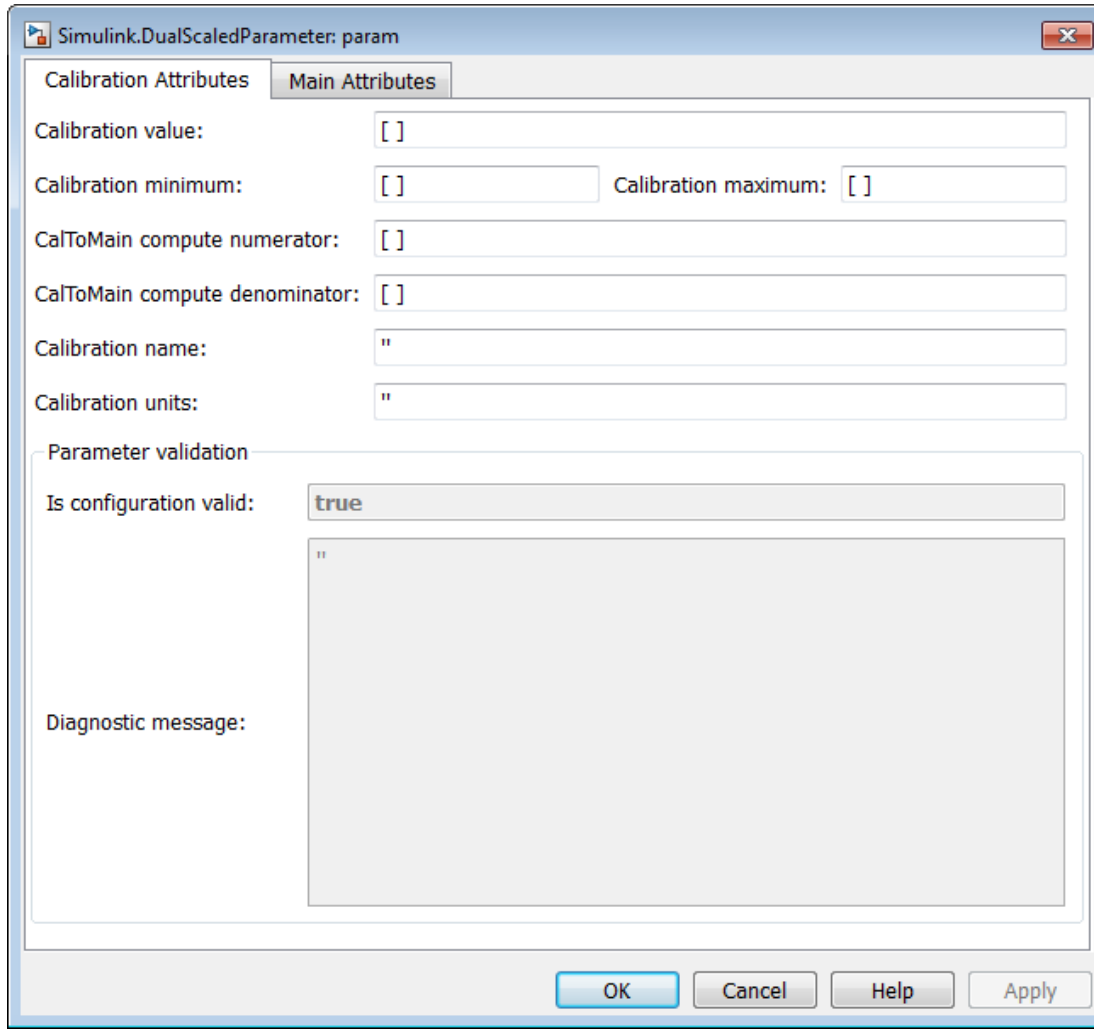
Property Dialog Box

Main Attributes Tab



This tab shows the properties inherited from the`Simulink.Parameter` class. For more information, see `Simulink.Parameter`.

Calibration Attributes Tab



Calibration value

Calibration value of the parameter. The value that you prefer to use. The default value is [ ] (unspecified). Specify a finite, real, double value.

Before specifying **Calibration value**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computational method. The parameter uses the computational method and the calibration value to calculate the main value that Simulink uses.

Calibration minimum

Minimum value for the calibration parameter. The default value is `[ ]` (unspecified). Specify a finite, real, double scalar value.

Before specifying **Calibration minimum**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computational method. The parameter uses the computational method and the calibration minimum value to calculate the minimum or maximum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration minimum sets the main minimum value. If it is decreasing, setting the calibration minimum sets the main maximum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

Calibration maximum

Maximum value for the calibration parameter. The default value is `[ ]` (unspecified). Specify a finite, real, double scalar value.

Before specifying **Calibration maximum**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computational method. The parameter uses the computational method and the calibration maximum value to calculate the corresponding maximum or minimum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration maximum sets the main maximum value. If it is decreasing, setting the calibration maximum sets the main minimum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

CalToMain compute numerator

Specify the numerator coefficients `a` and `b` of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is `[]` (unspecified). Specify finite, real, double scalar values for `a` and `b`. For example, `[1 1]` or, for reciprocal scaling, `1`.

Once you have applied CalToMain compute numerator, you cannot change it.

CalToMain compute denominator

Specify the denominator coefficients `c` and `d` of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is `[]` (unspecified). Specify finite, real, double scalar values for `c` and `d`. For example, `[1 1]`.

Once you have applied CalToMain compute denominator, you cannot change it.

Calibration name

Specify the name of the calibration parameter. The default value is `''`. Specify a string value, for example, `'T1'`.

Calibration units

Specify the measurement units for this calibration value. This field is intended for use in documenting this parameter. The default value is `''`. Specify a string value, for example, `'Fahrenheit'`.

Is configuration valid

Simulink indicates whether the configuration is valid. The default value is `true`. If Simulink detects an issue with the configuration, it sets this field to `false` and provides information in the **Diagnostic message** field. You cannot set this field.

Diagnostic message

If you specify invalid parameter settings, Simulink displays a message in this field. Use the diagnostic information to help you fix an invalid configuration issue. You cannot set this field.

Properties

Name	Access	Description
`CalibrationValue`	`RW`	Calibration value of this parameter. (Calibration value)
`CalibrationMin`	`RW`	Calibration minimum value of this parameter. (Calibration minimum)
`CalibrationMax`	`RW`	Calibration maximum value of this parameter. (Calibration maximum)
`CalToMainCompuNumerator`	`RW`	Numerator coefficients of the computational method. (CalToMain compute numerator) Once you have applied `CalToMainCompuNumerator`, you cannot change it.
`CalToMainCompuDenominator`	`RW`	Denominator coefficients of the computational method. (CalToMain compute denominator) Once you have applied `CalToMainCompuDenominator`, you cannot change it.
`CalibrationName`	`RW`	Name of the calibration parameter. (Calibration name)
`CalibrationDocUnits`	`RW`	Measurement units for this calibration parameter's value. (Calibration units)
`IsConfigurationValid`	`RO`	Information about validity of configuration. (Is configuration valid)
`DiagnosticMessage`	`RO`	If the configuration is invalid, diagnostic information to help you fix the issue. (Diagnostic message)

Example

Create and Update a Dual-Scaled Parameter

Create a `Simulink.DualScaledParameter` object that stores a temperature as both Fahrenheit and Celsius.

Create a `Simulink.DualScaledParameter` object.

```
Temp = Simulink.DualScaledParameter;
```

Set the computational method that converts between Fahrenheit and Celsius.

```
Temp.CalToMainCompuNumerator = [1 -32];
Temp.CalToMainCompuDenominator = [1.8];
```

Set the value of the temperature that you want to see in Fahrenheit.

```
Temp.CalibrationValue = 212
```

```
Temp =

  DualScaledParameter with properties:

            CalibrationValue: 212
              CalibrationMin: []
              CalibrationMax: []
     CalToMainCompuNumerator: [1 -32]
   CalToMainCompuDenominator: 1.8000
             CalibrationName: ''
         CalibrationDocUnits: ''
        IsConfigurationValid: 1
           DiagnosticMessage: ''
                       Value: 100
                   CoderInfo: [1x1 Simulink.CoderInfo]
                 Description: ''
                    DataType: 'auto'
                         Min: []
                         Max: []
                        Unit: ''
                  Complexity: 'real'
                  Dimensions: [1 1]
```

The `Simulink.DualScaledParameter` calculates `Temp.Value` which is the value that Simulink uses. `Temp.CalibrationValue` is `212` (degrees Fahrenheit), so `Temp.Value` is `100` (degrees Celsius).

Name the value and specify the units.

```
Temp.CalibrationName = 'TempF';
Temp.CalibrationDocUnits = 'Fahrenheit';
```

Set calibration minimum and maximum values.

```
Temp.CalibrationMin = 0;
Temp.CalibrationMax = 300;
```

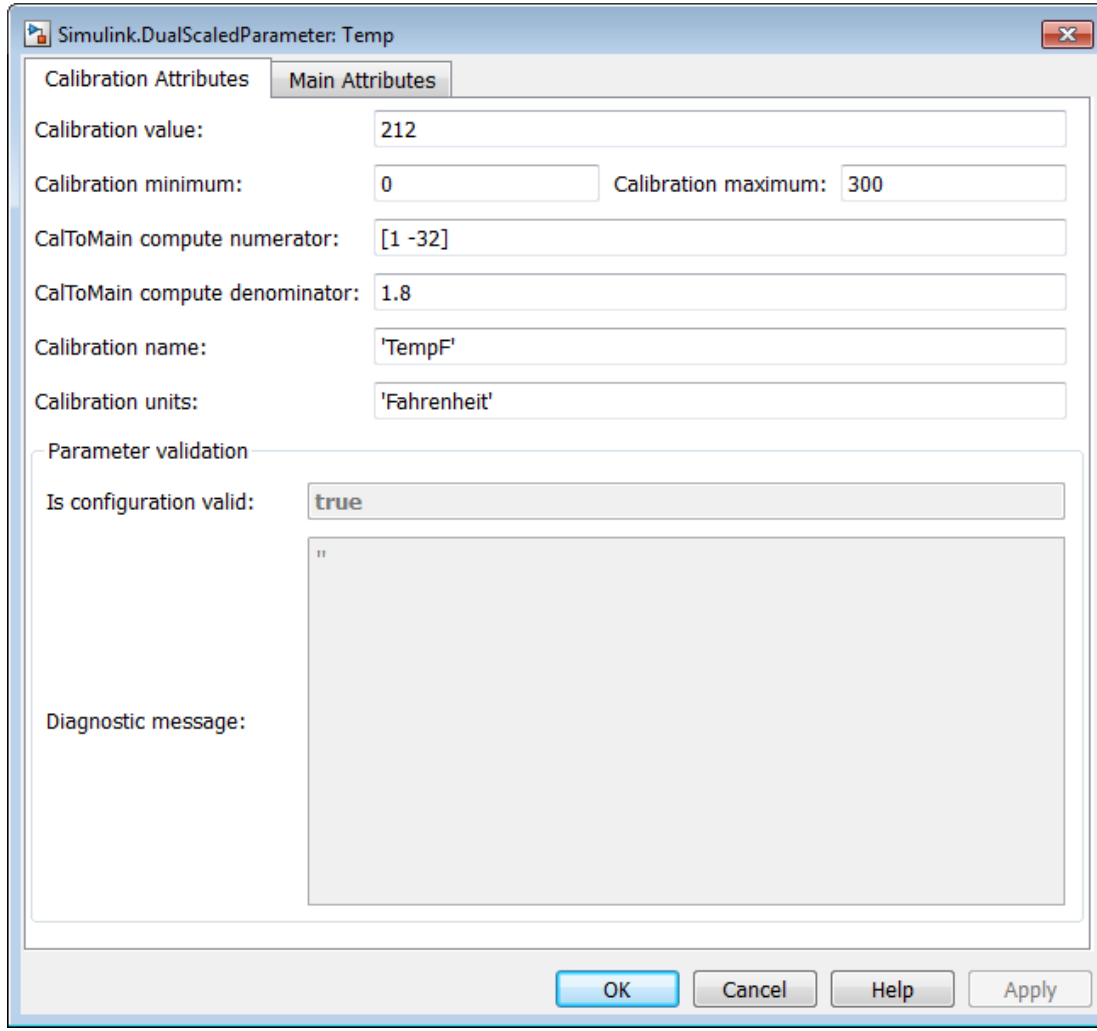
If you specify a calibration value outside this allowable range, Simulink generates a warning.

Specify the units that Simulink uses.

```
Temp.Unit = 'degC';
```

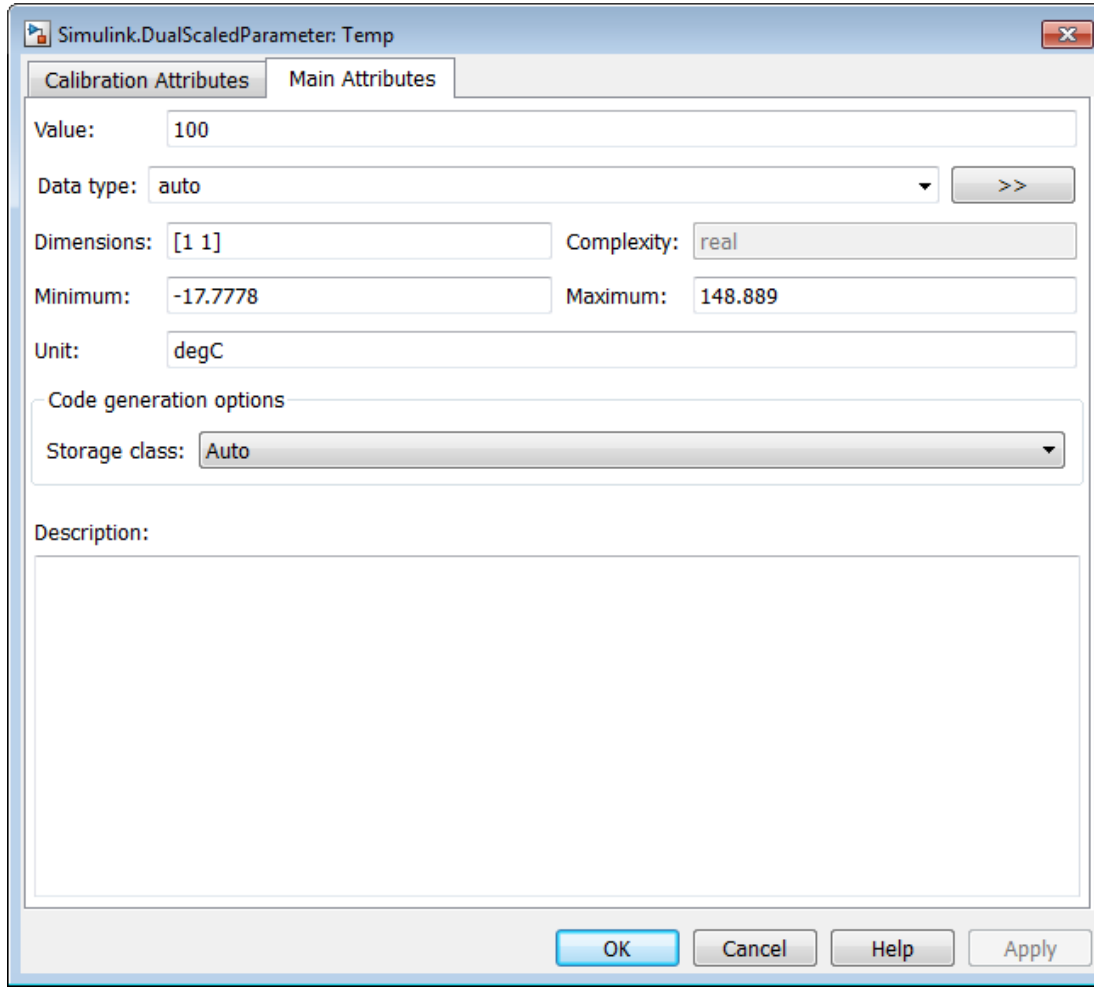
Open the Simulink.DualScaledParameter dialog box.

```
open Temp
```



The **Calibration Attributes** tab displays the calibration value and the computational method that you specified.

In the dialog box, click the **Main Attributes** tab.



This tab displays information about the value used by Simulink.

More About

- "Fixed Point"

See Also

AUTOSAR.DualScaledParameter | `Simulink.Parameter`

Related Examples

- "Configure AUTOSAR Data for Measurement and Calibration"

Introduced in R2013b

Simulink.Mask class

Package: Simulink

Control masks programmatically

Description

Use an instance of `Simulink.Mask` class to perform the following operations:

- Create, copy, and delete masks.
- Create, edit, and delete mask parameters.
- Determine the block that owns the mask.
- Obtain workspace variables defined for a mask.

Properties

`Type`

Specifies the mask type of the associated block.

Type: `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'|'port'`

Default: `'none'`

`PortRotate`

Specifies the port rotation policy for the masked block.

Type: `enum`

Values: `'default'|'physical'`

Default: `'default'`

`IconUnits`

Specifies the units for the drawing commands.

Type: `enum`

Values: `'pixel'|'autoscale'|'normalized'`

Default: `'autoscale'`

Methods

addParameter	Add a parameter to a mask
copy	Copy a mask from one block to another
create	Create a mask on a Simulink block
delete	Unmask a block and delete the mask from memory
get	Get a block mask as a mask object
addDialogControl	Add dialog control elements to mask dialog box
getDialogControl	Search for a specific dialog control on the mask
getOwner	Determine the block that owns a mask
getParameter	Get a mask parameter using its name
getWorkspaceVariables	Get all the variables defined in the mask workspace for a masked block
numParameters	Determine the number of parameters in a mask
removeDialogControl	Remove dialog control element from mask dialog box
removeParameter	Remove parameter from mask dialog box
removeAllParameters	Remove all existing parameters from a mask
set	Set the properties of an existing mask

How To

- "Control Masks Programmatically"
- "Block Masks"

addParameter

Class: Simulink.Mask
Package: Simulink

Add a parameter to a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.addParameter(Name,Value)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.addParameter(Name,Value)` appends a parameter to the mask. If you do not specify name–value pairs as arguments with this command, Simulink creates an unnamed mask parameter with control type set to `edit`.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Type'`

Type of control that is used to specify the value of this parameter.

Default: edit

`'TypeOptions'`

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

Default: empty

`'Name'`

The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Default: empty

`'Prompt'`

Text that identifies the parameter on the Mask Parameters dialog box.

Default: empty

`'Value'`

The default value of the mask parameter in the Mask Parameters dialog box.

Default: Type specific; depends on the `Type` of the parameter

`'Evaluate'`

Option to specify whether parameter must be evaluated.

Default: `'on'`

`'Tunable'`

Option to specify whether parameter is tunable.

Default: `'on'`

`'Enabled'`

Option to specify whether user can set parameter value.

Default: `'on'`

`'Visible'`

Option to set whether mask parameter is hidden or visible to the user.

Default: `'on'`

`'Callback'`

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog box and clicks **Apply**.

Default: empty

`'TabName'`

The name of the tab in the Mask Parameters dialog box where the parameter appears.

Default: empty

`'Container'`

Option to specifies a container for the child dialog control. The permitted values are `'panel'`, `'group'`, and `'tab'`.

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Add a parameter to the mask without specifying name–value pairs for parameter attributes.

   ```
   p.addParameter;
   ```

3. Add a mask parameter of type popup that cannot be evaluated and appears on a tab named **Properties** in the Mask Parameters dialog box.

   ```
   p.addParameter('Type','popup','TypeOptions',...
   {'Red' 'Blue' 'Green'},'Evaluate','off','TabName','Properties');
   ```

See Also

Simulink.Mask | "Block Masks"

Simulink.Mask.copy

Class: Simulink.Mask
Package: Simulink

Copy a mask from one block to another

Syntax

```
pSource = Simulink.Mask.get(srcBlockName)
pDest = Simulink.Mask.create(destBlockName)
pDest.copy(pSource)
```

Description

`pSource = Simulink.Mask.get(srcBlockName)` gets the mask on the source block specified by `blockName` as a mask object.

`pDest = Simulink.Mask.create(destBlockName)` creates an empty mask on the destination block specified by `destBlockName`.

`pDest.copy(pSource)` overwrites the destination mask with the source mask.

Input Arguments

`srcBlockName`

The handle to the source block or the path to the source block inside the model.

Note: The source block should be masked.

`destBlockName`

The handle to the destination block or the path to the destination block inside the model.

Note: The destination block should have an empty mask. Otherwise, the copied mask will overwrite the non-empty mask.

Examples

1. Create an empty mask on the destination block using the block's path.

```
pDest = Simulink.Mask.create('myModel/Subsystem');
```

2. Get source mask as an object using the source block's path.

```
pSource = Simulink.Mask.get('myModel/Abs');
```

3. Make the destination mask a copy of the source mask.

```
pDest.copy(pSource);
```

See Also

Simulink.Mask | "Block Masks"

Simulink.Mask.create

Class: Simulink.Mask
Package: Simulink

Create a mask on a Simulink block

Syntax

```
p = Simulink.Mask.create(blockName)
```

Description

`p = Simulink.Mask.create(blockName)` creates an empty mask on the block specified by `blockName`. If the specified block is already masked, an error message appears.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Examples

1 Create a mask using a block's handle.

> **Note:** In the model, select the block to be masked.

```
p = Simulink.Mask.create(gcbh);
```

2 Create a mask using the block's path.

```
p = Simulink.Mask.create('myModel/Subsystem');
```

See Also

Simulink.Mask | “Block Masks”

delete

Class: Simulink.Mask
Package: Simulink

Unmask a block and delete the mask from memory

Syntax

```
p = Simulink.Mask.get(blockName)
p.delete
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.delete` unmasks the block and deletes the mask from memory.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Unmask the block using the mask object and delete the mask from memory.

   ```
   p.delete;
   ```

See Also

Simulink.Mask | "Block Masks"

Simulink.Mask.get

Class: Simulink.Mask
Package: Simulink

Get a block mask as a mask object

Syntax

```
p = Simulink.Mask.get(blockName)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object. If the specified block is not masked, a null value returns.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Examples

1. Get mask as an object using a masked block's handle.

   Note: In the model, select the masked block.

   ```
   p = Simulink.Mask.get(gcbh);
   ```

2. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

See Also

Simulink.Mask | “Block Masks”

addDialogControl

Class: Simulink.Mask
Package: Simulink

Add dialog control elements to mask dialog box

Syntax

```
successIndicator = maskObj.addDialogControl(controlType,
controlIdentifier)
successIndicator = maskObj.addDialogControl(Name,Value)
```

Description

`successIndicator = maskObj.addDialogControl(controlType, controlIdentifier)` adds dialog control elements like text, hyperlinks, or tabs to mask dialog box. First get the mask object and assign it to the variable `maskObj`

`successIndicator = maskObj.addDialogControl(Name,Value)` specifies the Name and Value arguments for an element on the mask dialog box. You can specify multiple Name-Value pairs.

Input Arguments

`controlType` — Value type of dialog control element

`string`

Type of dialog control element, specified

- `'panel'`
- `'group'`
- `'tabcontainer'`
- `'tab'`

- `'collapsiblepanel'`
- `'text'`
- `'image'`
- `'hyperlink'`
- `'pushbutton'`

`controlIdentifer` — Unique identifier for the element
string

Specifies the programmatic identifier for the element of mask dialog box. Use a name that is unique and does not have space between words. For more information, see "Variable Names".

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`) and is case-sensitive. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Type'`

Type of control that is used to specify the value of this dialog control element. Type is a required argument. The permitted values are `'panel'`, `'group'`, `'tabcontainer'`, `'tab'`, `'collapsiblepanel'`, `'text'`, `'image'`, `'hyperlink'`, and `'pushbutton'`.

Default: empty

`'Name'`

The identifier of the dialog control element. Name is a required argument. This field is available for all dialog control types.

Default: empty

`'Prompt'`

Text that is displayed in the dialog control element on the Mask dialog box. This field is available for all except for panel and image dialog control types.

Default: empty

`'Enabled'`

Option to specify whether you can set value for the dialog control element. This field is available for all dialog control types.

Default: `'on'`

`'Visible'`

Option to set whether the dialog control element is hidden or visible to the user. This field is available for all dialog control types.

Default: `'on'`

`'Callback'`

Container for MATLAB code that executes when you edit the dialog control element and click **Apply**. This field is available only for the hyperlink and pushbutton dialog control types.

Default: empty

`'Row'`

Option to set whether the dialog control is placed in the new row or the same row. This field is available for all dialog control types.

Default: empty

`'Filepath'`

Contains the path to an image file. This field is available for image, and pushbutton dialog control types.

Default: empty

`'Container'`

Option to specifies a container for the child dialog control. The permitted values are the names of `'panel'`, `'group'`, and `'tab'` dialog controls.

Examples

Add Dialog Control Elements to Mask Dialog Box

Get mask object and add dialog control element to it.

```
% Get mask object on model Engine

maskObj = Simulink.Mask.get('Engine/Gain');

% Add hyperlink to mask dialog box

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'Mathworks Home Page';
maskLink.Callback = 'web(''www.mathworks.com'')'

% Alternative method to add hyperlink

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'www.mathworks.com';

% Add text to mask dialog box

maskText = maskObj.addDialogControl('text','text_tag');
maskText.Prompt = 'Enable range checking';

% Add button to mask dialog box

maskButton = maskObj.addDialogControl('pushbutton','button_tag');
maskButton.Prompt = 'Compute';
```

Add Dialog Control Elements to Mask Dialog Box Tabs

Create tabs on the mask dialog box and add elements to these tabs.

```
% Get mask object on a block named 'GainBlock'

maskObj = Simulink.Mask.get('GainBlock/Gain');

% Create a tab container

maskObj.addDialogControl('tabcontainer','allTabs');
tabs = maskObj.getDialogControl('allTabs');
```

```
% Create tabs and name them

maskTab1 = tabs.addDialogControl('tab','First');
maskTab1.Prompt = 'First tab';

maskTab2 = tabs.addDialogControl('tab','Second');
maskTab2.Prompt = 'Second tab';

% Add elements to one of the tabs

firstTab = tabs.getDialogControl('First');
firstTab.addDialogControl('text','textOnFirst');
firstTab.getDialogControl('textOnFirst').Prompt = 'Tab one';
```

Add Dialog Control Element Using Name-Value Pair

Add dialog control element and specify values for it

```
% Get mask object on model Engine

maskObj = Simulink.Mask.get('Engine/Gain');

% Add a dialog box and specify values for it

maskDialog = maskObj.addDialogControl('Type','text',...
'Prompt','hello','Visible','off');
```

See Also

Simulink.Mask | “Block Masks” | Control Masks Programmatically

Introduced in R2014a

getDialogControl

Class: Simulink.Mask
Package: Simulink

Search for a specific dialog control on the mask

Syntax

```
[control, phandle] = handle.getDialogControl(cname)
```

Description

`[control, phandle] = handle.getDialogControl(cname)` , search for a specific child dialog control recursively on the mask dialog.

Input Arguments

`cname`

Name of the dialog control being searched on the mask dialog.

Default:

Output Arguments

`control`

Target dialog control being searched on the mask dialog.

`phandle`

Parent of the dialog control being searched mask dialog.

Examples

Find a dialog control

Find a text dialog control on the mask dialog box. `maskObj` is the handle to the mask object. The `getDialogControl` method returns the handle to the dialog control (`hdlgctrl`) and handle to the parent dialog control (`phandle`).

```
[hdlgctrl, phandle] = maskObj.getDialogControl('txt_var')
```

See Also

Simulink.Mask | "Block Masks"

getOwner

Class: Simulink.Mask
Package: Simulink

Determine the block that owns a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.getOwner
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.getOwner` returns the interface to the block that owns the mask.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Get the interface to the block that owns the mask.

   ```
   p.getOwner;
   ```

See Also

Simulink.Mask | "Block Masks"

getParameter

Class: Simulink.Mask
Package: Simulink

Get a mask parameter using its name

Syntax

```
p = Simulink.Mask.get(blockName)
param = p.getParameter(paramName)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`param = p.getParameter(paramName)` returns 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 | “Block Masks”

getWorkspaceVariables

Class: Simulink.Mask
Package: Simulink

Get all the variables defined in the mask workspace for a masked block

Syntax

```
p = Simulink.Mask.get(blockName)
vars = p.getWorkspaceVariables
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`vars = p.getWorkspaceVariables` returns as a structure all the variables defined in the mask workspace for the masked block.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Get all the variables defined in the mask workspace for the masked block.

   ```
   vars = p.getWorkspaceVariables;
   ```

See Also

Simulink.Mask | “Block Masks”

numParameters

Class: Simulink.Mask
Package: Simulink

Determine the number of parameters in a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.numParameters
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.numParameters` returns the number of parameters in the mask.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Get the number of parameters in the mask.

   ```
   p.numParameters;
   ```

See Also

Simulink.Mask | "Block Masks"

removeDialogControl

Class: Simulink.Mask
Package: Simulink

Remove dialog control element from mask dialog box

Syntax

```
successIndicator = maskVariable.removeDialogControl(
controlIdentifier)
```

Description

`successIndicator = maskVariable.removeDialogControl(controlIdentifier)` removes dialog control element, specified by `controlIndentifier`, like text, hyperlinks, or tabs from a mask dialog box. First get the mask object and assign it to the variable `maskVariable`.

Successful removal of a dialog control element returns a Boolean value of `1`.

Input Arguments

`controlIdentifer` — Unique identifier for the element

string

Programmatic identifier for the dialog control element of mask dialog box, specified as a string.

Examples

Remove Dialog Control Element from Mask Dialog Box

```
% Get mask object on the Gain block in the model Engine.
```

```
maskObj = Simulink.Mask.get('Engine/Gain');

% Remove element named AllTab from mask dialog box.

p = maskObj.removeDialogControl('AllTab');
```

See Also

Simulink.Mask | “Block Masks”

Introduced in R2013b

removeParameter

Class: Simulink.Mask
Package: Simulink

Remove parameter from mask dialog box

Syntax

```
successIndicator = maskVariable.removeParameter(controlIdentifier)
```

Description

`successIndicator = maskVariable.removeParameter(controlIdentifier)` removes parameter, specified by `controlIdentifier`, like edit, check box, popup from an existing mask dialog box. First get the mask object and assign it to the variable `maskVariable`.

Successful removal of a parameter returns a Boolean value of `1`.

Input Arguments

`controlIdentifer` — Unique identifier for the parameter

string

Programmatic identifier for the parameter of mask dialog box, specified as a string.

Examples

Remove Parameter from Mask Dialog Box

```
% Get mask object on the Gain block in the model Engine.

maskObj = Simulink.Mask.get('Engine/Gain');

% Remove parameter named checkbox1 from mask dialog box.
```

```
p = maskObj.removeParameter('checkbox1');
```

Note: You can also use the index number as the `controlIdentifier`.

See Also

Simulink.Mask | "Block Masks"

Introduced in R2012b

removeAllParameters

Class: Simulink.Mask
Package: Simulink

Remove all existing parameters from a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.removeAllParameters
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.removeAllParameters` deletes all existing parameters from the mask.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Delete all existing parameters from the mask.

   ```
   p.removeAllParameters;
   ```

See Also

Simulink.Mask | "Block Masks"

set

Class: Simulink.Mask
Package: Simulink

Set the properties of an existing mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.set(Name,Value)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.set(Name,Value)` sets mask properties that you specify using name–value pairs as arguments.

Input Arguments

`blockName`

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Type'`

Text used as title for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

`'Description'`

Text used as summary for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

`'Help'`

Text used as body text for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

`'Initialization'`

MATLAB code that initializes the mask.

Default: empty

`'SelfModifiable'`

Option to set whether the mask can modify itself during simulation.

Default: `'off'`

`'Display'`

MATLAB code that draws the mask icon.

Default: empty

`'IconFrame'`

Option to specify whether the mask icon appears inside a visible block frame.

Default: `'on'`

`'MaskIconOpaque'`

Option to set the mask icon as opaque or transparent.

Default: `'opaque'`

`'RunInitForIconRedraw'`

Option to specify whether Simulink should run mask initialization before executing the mask icon commands.

Default: `'off'`

`'IconRotate'`

Option to specify icon rotation.

Default: `'none'`

`'PortRotate'`

Option to specify port rotation.

Default: `'default'`

`'IconUnits'`

Option to specify whether mask icon is autoscaled, normalized, or scaled in pixels.

Default: `'autoscale'`

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Modify the mask so that its mask icon is transparent and its documentation summarizes what it does.

   ```
   p.set('IconOpaque','off','Type','Random number generator','Description',...
   'This block generates random numbers.');
   ```

See Also

Simulink.Mask | "Block Masks"

Simulink.MaskParameter class

Package: Simulink

Control mask parameters programmatically

Description

Use an instance of `Simulink.MaskParameter` to set the properties of mask parameters.

Properties

`Type`

Specifies the mask parameter type.

Type: `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`, `radio`, `Datatypestr`, 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'`

Hidden

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`

`Alias`

Specifies the alternate name for mask parameter.

Type: `string`

Default: `Empty String`

Methods

set	Set properties of mask parameters

How To

- "Control Masks Programmatically"
- "Block Masks"

set

Class: Simulink.MaskParameter
Package: Simulink

Set properties of mask parameters

Syntax

```
Simulink.MaskParameter.set(Name,Value)
```

Description

`Simulink.MaskParameter.set(Name,Value)` sets the properties of a mask parameter.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Type'`

Type of control that is used to specify the value of this parameter.

Default: edit

`'TypeOptions'`

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

Default: empty

`'Name'`

The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Default: empty

`'Prompt'`

Text that identifies the parameter on the Mask Parameters dialog.

Default: empty

`'Value'`

The default value of the mask parameter in the Mask Parameters dialog.

Default: Type specific; depends on the `Type` of the parameter

`'Evaluate'`

Option to specify whether parameter must be evaluated.

Default: `'on'`

`'Tunable'`

Option to specify whether parameter is tunable.

Default: `'on'`

`'Enabled'`

Option to specify whether user can set parameter value.

Default: `'on'`

`'Visible'`

Option to set whether mask parameter is hidden or visible to the user.

Default: `'on'`

`'Callback'`

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog and clicks **Apply**.

Default: empty

`'TabName'`

The name of the tab in the Mask Parameters dialog where the parameter appears.

Default: empty

Examples

1. Get mask as an object using a masked block's path.

   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Get a mask parameter.

   ```
   a = p.Parameters(1);
   ```

3. Edit mask parameter so it is of type popup, cannot be evaluated, and appears on a tab named **Properties** in the Mask Parameters dialog.

   ```
   a.set('Type','popup','TypeOptions',{'Red' 'Blue' 'Green'},...
   'Evaluate','off','TabName','Properties');
   ```

See Also

Simulink.Mask | Simulink.MaskParameter | "Block Masks"

Simulink.dialog.Control class

Package: Simulink.dialog

Create instances of dialog control

Description

Use an instance of `Simulink.dialog.Control` class to create, delete, or search dialog controls.

Properties

`Name`

Uniquely identifies the dialog control element and is a required field.

Type: `string`

See Also

`Simulink.dialog.Button` | `Simulink.dialog.Image` | `Simulink.dialog.Text` | `Simulink.dialog.parameter.Control` | `Simulink.dialog.Hyperlink` | `Simulink.dialog.Container` | "Block Masks"

Simulink.dialog.Container class

Package: Simulink.dialog

Create instances of a container dialog control

Description

Use an instance of `Simulink.dialog.Container` class to add container type dialog control.

Properties

`Name`

Uniquely identifies the container dialog control and is a required filed.

Type: `string`

`Enabled`

Indicates whether container is active on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

`Visible`

Indicates whether container is displayed on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

`DialogControls`

Specifies the child dialog controls contained in the container.

Type: `Simulink.dialog.Control`

Default: `Empty array`

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Group` | `Simulink.dialog.Tab` | `Simulink.dialog.TabContainer` | `Simulink.dialog.Panel` | `Simulink.dialog.Control` | "Block Masks"

addDialogControl

Class: Simulink.dialog.Container
Package: Simulink.dialog

Add dialog control elements to mask dialog box

Syntax

```
successIndicator = maskObj.addDialogControl(controlType,
controlIdentifier)
successIndicator = maskObj.addDialogControl(Name,Value)
```

Description

`successIndicator = maskObj.addDialogControl(controlType, controlIdentifier)` adds dialog control elements like text, hyperlinks, or tabs to mask dialog box. First get the mask object and assign it to the variable `maskObj`

`successIndicator = maskObj.addDialogControl(Name,Value)` specifies the Name and Value arguments for an element on the mask dialog box. You can specify multiple Name-Value pairs.

Input Arguments

`controlType` — Value type of dialog control element

`string`

Type of dialog control element, specified

- `'panel'`
- `'group'`
- `'tabcontainer'`
- `'tab'`
- `'collapsiblepanel'`
- `'text'`

- `'image'`
- `'hyperlink'`
- `'pushbutton'`

`controlIdentifer` — Unique identifier for the element
string

Specifies the programmatic identifier for the element of mask dialog box. Use a name that is unique and does not have space between words. For more information, see "Variable Names".

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`) and is case-sensitive. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Type'`

Type of control that is used to specify the value of this dialog control element. Type is a required argument. The permitted values are `'panel'`, `'group'`, `'tabcontainer'`, `'tab'`, `'collapsiblepanel'`, `'text'`, `'image'`, `'hyperlink'`, and `'pushbutton'`. If the parent dialog control type is `'tabcontainer'`, the child dialog control must be `'tab'`.

`'Name'`

The identifier of the dialog control element. Name is a required argument. This field is available for all dialog control types.

`'Prompt'`

Text that is displayed in the dialog control element on the Mask dialog box. This field is available for all except for panel and image dialog control types.

Default: empty

`'Enabled'`

Option to specify whether you can set value for the dialog control element. This field is available for all dialog control types.

Default: `'on'`

`'Visible'`

Option to set whether the dialog control element is hidden or visible to the user. This field is available for all dialog control types.

Default: `'on'`

`'Callback'`

Container for MATLAB code that executes when you edit the dialog control element and click **Apply**. This field is available only for the hyperlink and pushbutton dialog control types.

Default: empty

`'Row'`

Option to set whether the dialog control is placed in the new row or the same row. This field is available for all dialog control types.

Default: empty

`'FilePath'`

Contains the path to an image file. This field is available for image, and pushbutton dialog control types.

Default: empty

`'Container'`

Option to specifies a container for the child dialog control. The permitted values are the names of `'panel'`, `'group'`, and `'tab'` dialog controls.

Examples

Add Dialog Control Elements to Mask Dialog Box

Get mask object and add dialog control element to it.

```
% Get mask object on model Engine

maskObj = Simulink.Mask.get('Engine/Gain');

% Add hyperlink to mask dialog box

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'Mathworks Home Page';
maskLink.Callback = 'web(''www.mathworks.com'')'

% Alternative method to add hyperlink

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'www.mathworks.com';

% Add text to mask dialog box

maskText = maskObj.addDialogControl('text','text_tag');
maskText.Prompt = 'Enable range checking';

% Add button to mask dialog box

maskButton = maskObj.addDialogControl('pushbutton','button_tag');
maskButton.Prompt = 'Compute';
```

Add Dialog Control Elements to Mask Dialog Box Tabs

Create tabs on the mask dialog box and add elements to these tabs.

```
% Get mask object on a block named 'GainBlock'

maskObj = Simulink.Mask.get('GainBlock/Gain');

% Create a tab container

maskObj.addDialogControl('tabcontainer','allTabs');
tabs = maskObj.getDialogControl('allTabs');

% Create tabs and name them

maskTab1 = tabs.addDialogControl('tab','First');
maskTab1.Prompt = 'First tab';

maskTab2 = tabs.addDialogControl('tab','Second');
maskTab2.Prompt = 'Second tab';
```

```
% Add elements to one of the tabs

firstTab = tabs.getDialogControl('First');
firstTab.addDialogControl('text','textOnFirst');
firstTab.getDialogControl('textOnFirst').Prompt = 'Tab one';
```

Add Dialog Control Element Using Name-Value Pair

Add dialog control element and specify values for it

```
% Get mask object on model Engine

maskObj = Simulink.Mask.get('Engine/Gain');

% Add a dialog box and specify values for it

maskDialog = maskObj.addDialogControl('Type','text',...
'Prompt','hello','Visible','off');
```

See Also

`Simulink.dialog.Container` | "Block Masks"

Introduced in R2014a

removeDialogControl

Class: Simulink.dialog.Container
Package: Simulink.dialog

Remove dialog control element from mask dialog box

Syntax

```
successIndicator = maskVariable.removeDialogControl(
controlIdentifier)
```

Description

`successIndicator = maskVariable.removeDialogControl(controlIdentifier)` removes dialog control element, specified by `controlIndentifier`, like text, hyperlinks, or tabs from a mask dialog box. First get the mask object and assign it to the variable `maskVariable`.

Successful removal of a dialog control element returns a Boolean value of `1`.

Input Arguments

`controlIdentifer` — Unique identifier for the element

string

Programmatic identifier for the dialog control element of mask dialog box, specified as a string.

Examples

Remove Dialog Control Element from Mask Dialog Box

```
% Get mask object on the Gain block in the model Engine.
```

```
maskObj = Simulink.Mask.get('Engine/Gain');

% Remove element named AllTab from mask dialog box.

maskTab = maskObj.removeDialogControl('AllTab');
```

See Also

`Simulink.dialog.Container` | "Block Masks"

Introduced in R2013b

getDialogControl

Class: Simulink.dialog.Container
Package: Simulink.dialog

Search for a specific dialog control on the mask

Syntax

```
[control, phandle] = handle.getDialogControl(controlIdentifier)
```

Description

`[control, phandle] = handle.getDialogControl(controlIdentifier)`, search for a specific child dialog control recursively on the mask dialog box.

Input Arguments

`controlIdentifier`

Name of the dialog control being searched on the mask dialog box.

Default:

Output Arguments

`control`

Target dialog control being searched on the mask dialog box.

`phandle`

Parent of the dialog control being searched mask dialog box.

Examples

Find a dialog control

Find a text dialog control on the mask dialog box. `maskObj` is the handle to the mask object. The `getDialogControl` method returns the handle to the dialog control (`hdlgctrl`) and handle to the parent dialog control (`phandle`).

```
[hdlgctrl, phandle] = maskObj.getDialogControl('txt_var')
```

See Also

`Simulink.dialog.Container` | "Block Masks"

Simulink.dialog.Panel class

Package: Simulink.dialog

Create an instance of a panel dialog control

Description

Use an instance of `Simulink.dialog.Panel` class to create an instance of panel dialog control.

Properties

`Name`

Uniquely identifies the panel dialog control and is a required field.

Type: `string`

`Row`

Specifies whether panel is placed on the current row or on a new row.

Type: `enumerated string`

Values: `'current'|'new'`

Default: `'new'`

`Enabled`

Specifies whether panel is active on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`Visible`

Specifies whether panel is displayed on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`DialogControls`

Specifies the child dialog controls contained in the panel.

Type: `Simulink.dialog.Control`

Default: `Empty array`

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Group` | `Simulink.dialog.TabContainer` | `Simulink.dialog.Container` | `Simulink.dialog.Tab` | `Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.Group class

Package: Simulink.dialog

Create an instance of a group dialog control

Description

Use an instance of `Simulink.dialog.Group` class to create an instance of group dialog control.

Properties

`Name`

Uniquely identifies the group dialog control and is a required field.

Type: `string`

`Prompt`

Specifies the text displayed on the group.

Type: `string`

Default: `Empty String`

`Row`

Specifies whether group is placed on the current row or on a new row.

Type: `enumerated string`

Values: `'current'|'new'`

Default: `'new'`

`Enabled`

Specifies whether group is active on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`Visible`

Specifies whether group is displayed on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`DialogControls`

Specifies the child dialog controls contained in the group.

Type: `Simulink.dialog.Control`

Default: `Empty array`

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Panel` | `Simulink.dialog.TabContainer` | `Simulink.dialog.Container` | `Simulink.dialog.Tab` | `Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.Tab class

Package: Simulink.dialog

Create an instance of a tab dialog control

Description

Use an instance of `Simulink.dialog.Tab` class to create an instance of tab dialog control.

Properties

`Name`

Uniquely identifies the tab dialog control and is a required field.

Type: `string`

`Prompt`

Specifies the text displayed on the tab.

Type: `string`

Default: `Empty String`

`Enabled`

Specifies whether tab is active on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`Visible`

Specifies whether tab is displayed on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`DialogControls`

Specifies the child dialog controls contained in the tab dialog control.

Type: `Simulink.dialog.Control`

Default: `Empty array`

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Group` | `Simulink.dialog.TabContainer` | `Simulink.dialog.Container` | `Simulink.dialog.Panel` | `Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.TabContainer class

Package: Simulink.dialog

Create an instance of a tab container dialog control

Description

Use an instance of `Simulink.dialog.TabContainer` class to create an instance of tab container dialog control. Tab container dialog box be used to group the tab dialog controls.

Properties

`Name`

Uniquely identifies the tab container dialog control and is a required field.

Type: `string`

`Row`

Specifies whether tab container is placed on the current row or on a new row.

Type: `enumerated string`

Values: `'current'|'new'`

Default: `'new'`

`Enabled`

Specifies whether tab container is active on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`Visible`

Specifies whether tab container is displayed on the mask dialog box.

Type: `boolean`

Values: `'on'|'off'`

Default: `'on'`

`DialogControls`

Specifies the child dialog controls contained in the group. `Simulink.dialog.TabContainer` class can only contain `Simulink.dialog.Tab` dialog control.

Type: `Simulink.dialog.Tab`

Default: `Empty array`

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Group` | `Simulink.dialog.Tab` | `Simulink.dialog.Container` | `Simulink.dialog.Panel` | `Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.Button class

Package: Simulink.dialog

Create a button dialog control

Description

Use an instance of `Simulink.dialog.Button` class to add a button dialog control.

Properties

`Name`

Uniquely identifies the dialog control and is a required field.

Type: `string`

`Prompt`

Specifies the text displayed on the button dialog control.

Type: `string`

Default: `empty`

`FilePath`

Specifies the path to the image file to be shown on the button dialog control.

Type: `string`

Default: `empty`

`Callback`

Specifies the MATLAB command (s) to be executed when the dialog control is invoked.

Type: `string`

Default: `empty`

`Row`

Specifies whether dialog control is placed on the current row or on a new row.

Type: `enumerated string`

Value: `'current'|'new'`

Default: `'current'`

`Enabled`

Indicates whether container is active on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

`Visible`

Indicates whether container is displayed on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

See Also

`Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.Hyperlink class

Package: Simulink.dialog

Create a hyperlink dialog control

Description

Use an instance of `Simulink.dialog.Hyperlink` class to add a hyperlink dialog control.

Properties

`Name`

Uniquely identifies the dialog control and is a required field.

Type: `string`

`Prompt`

Specifies the text displayed on the hyperlink.

Type: `string`

Default: `empty`

`Callback`

Specifies the MATLAB command (s) to be executed when the dialog control is invoked.

Type: `string`

Default: `empty`

`Row`

Specifies whether hyperlink is placed on the current row or on a new row.

Type: `enumerated string`

Value: `'current'|'new'`

Default: `'new'`

`Enabled`

Indicates whether hyperlink is active on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

`Visible`

Indicates whether hyperlink is displayed on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

See Also

`Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.Image class

Package: Simulink.dialog

Create an image dialog control

Description

Use an instance of `Simulink.dialog.Image` class to add an image dialog control.

Properties

`Name`

Uniquely identifies the dialog control and is a required field.

Type: `string`

`FilePath`

Specifies the path to the image file to be displayed on the dialog box.

Type: `string`

Default: `empty`

`Row`

Specifies whether dialog control is placed on the current row or on a new row.

Type: `enumerated string`

Value: `'current'|'new'`

Default: `'new'`

`Enabled`

Indicates whether image is active on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

`Visible`

Indicates whether image is displayed on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

See Also

`Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.Text class

Package: Simulink.dialog

Create a text dialog control

Description

Use an instance of `Simulink.dialog.Text` class to add a text dialog control.

Properties

`Name`

Uniquely identifies the dialog control element and is a required field.

Type: `string`

`Prompt`

Specifies the text displayed on the mask dialog box.

Type: `string`

Default: `empty`

`WordWrap`

Specifies whether to wrap long text to the next line on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

`Row`

Specifies whether dialog control is placed on the current row or on a new row.

Type: `enumerated string`

Value: `'current'|'new'`

Default: `'new'`

`Enabled`

Indicates whether dialog control is active on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

`Visible`

Indicates whether dialog control is displayed on the mask dialog box.

Type: `boolean`

Value: `'on'|'off'`

Default: `'on'`

See Also

`Simulink.dialog.Control` | "Block Masks"

Simulink.dialog.parameter.Control class

Package: Simulink.dialog.parameter

Create a parameter dialog control

Description

Use an instance of `Simulink.dialog.parameter.Control` class to add a parameter dialog control.

Properties

`Name`

Uniquely identifies the dialog control element. This is a required field and has the same value as its underlying parameter name.

Type: `string`

`Row`

Specifies whether the dialog control is placed on the current row or on a new row.

Type: `enumerated string`

Value: `'current'|'new'`

Default: `'new'`

See Also

`Simulink.dialog.Control` | "Block Masks"

addElement

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Add element to end of data set

Syntax

```
dataset = addElement(dataset,element)
dataset = addElement(dataset,element,name)
```

Description

`dataset = addElement(dataset,element)` adds an element to the `Simulink.SimulationData.Dataset` dataset.

`dataset = addElement(dataset,element,name)` adds an element to the `Simulink.SimulationData.Dataset` data set and gives the element the name that you specify with the `name` argument. If the object already has a name, the element instead uses the name you specify by using the `name` argument.

Input Arguments

`dataset` — Data set

`SimulationData.Dataset` object

The data set to which to add the element.

`element` — Element to add

`Simulink.SimulationData.Signal` object | `Simulink.SimulationData.DataStoreMemory` object

Element to add to the data set, specified as a `Simulink.SimulationData.Signal` object or `Simulink.SimulationData.DataStoreMemory` object.

`name` — Name for element

string

Name for element, specified as a string.

Output Arguments

`dataset` — Data set

string

The data set to which you add the element, returned as a string. The new element is added to the end of the data set.

Examples

Create a Data Set

Create a data set and add three elements to it.

```
time = 0.1*(0:100)';
ds = Simulink.SimulationData.Dataset;
element1 = Simulink.SimulationData.Signal;
element1.Name = 'A';
element1.Values = timeseries(sin(time),time);
ds = addElement(ds,element1);
element2 = Simulink.SimulationData.Signal;
element2.Name = 'B';
element2.Values = timeseries(2*sin(time),time);
ds = addElement(ds,element2);
element3 = Simulink.SimulationData.Signal;
element3.Name = 'C';
element3.Values = timeseries(3*sin(time),time);
ds = addElement(ds,element3)
```

```
ds =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

  Characteristics:
              Name: 'logsout'
```

```
    Total Elements: 3

  Elements:
    1: 'A'
    2: 'B'
    3: 'C'
```

- “Migrate Scripts That Use ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use ModelDataLogs API”

See Also

Simulink.SimulationData.Dataset.concat | Simulink.SimulationData.Dataset.find | Simulink.SimulationData.Dataset.get | Simulink.SimulationData.Dataset.getElementNames | Simulink.SimulationData.Dataset.numElements | Simulink.SimulationData.Dataset.setElement | Simulink.SimulationData.Dataset | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Signal`

Introduced in R2011a

concat

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Concatenate dataset to another dataset

Syntax

```
dataset1 = concat(dataset1,dataset2)
```

Description

`dataset1 = concat(dataset1,dataset2)` concatenates the elements of `dataset2` to `dataset1`.

Input Arguments

`dataset1` — Dataset to concatenate to

data set

Dataset to concatenate to with `dataset2`, returned as a cell array.

`dataset2` — Dataset to concatenate

data set

Data set to concatenate to `dataset1`, specified as a cell array.

Output Arguments

`dataset1` — Concatenated dataset

data set

Concatenated dataset from `dataset1` and `dataset2`.

Examples

Concatenate ds1 to ds

Convert output from two To Workspace blocks to `Dataset` format and concatenate them.

```
mdl = 'myvdp';
open_system(mdl);
sim(mdl)
ds = Simulink.SimulationData.Dataset(simout);
ds1 = Simulink.SimulationData.Dataset(simout1);
dsfinal = concat(ds,ds1);
```

- “Migrate Scripts That Use ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use ModelDataLogs API”

See Also

Simulink.SimulationData.Dataset.addElement | Simulink.SimulationData.Dataset.find | Simulink.SimulationData.Dataset.get | Simulink.SimulationData.Dataset.getElementNames | Simulink.SimulationData.Dataset.numElements | Simulink.SimulationData.Dataset.setElement | Simulink.SimulationData.Dataset | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Signal`

Introduced in R2015a

get

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Get element or collection of elements from dataset

Syntax

```
element = get(dataset,index)
element = get(dataset,name)
element = get(dataset,{name})
```

Description

`element = get(dataset,index)` returns the element corresponding to the `index`. The `getElement` method uses the same syntax and behavior as the `get` method.

`element = get(dataset,name)` returns the element whose name matches `name`. When `name` is in a cell array, return the index of the element whose name matches `name`.

`element = get(dataset,{name})` returns a single element if only one element name matches, a `SimulationData.Dataset` if multiple elements with this name exist.

Input Arguments

`dataset` — Data set

`SimulationData.Dataset` object

The data set from which to get the element.

`index` — Index value of element to get

scalar numeric

Index value of element to get. The index reflects the index value of a data set element.

`name` — Name for data set element

character array | cell array

Name for a data set element, specified as:

- A character array reflecting the name of the data set element
- A cell array containing one string. To return a `SimulationData.Dataset object` that can contain one element, use this format. Consider this form when writing scripts.

Output Arguments

`element` — Element

element | `SimulationData.Dataset` object | empty object

The element that the `get` method finds.

- If `index` is the first argument after the data set, the method returns the element at the `index`.
- If `name` is the first argument after the data set:
  - If the method finds one element, it returns the element.
  - If the method finds more than one element, return a Dataset that contains the elements.
  - If the method does not find an element, it returns an empty object.

Examples

Access Dataset Elements

Access dataset elements in the top model of the `ex_bus_logging` model. The signal logging dataset is `topOut`.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_mdlref_counter_bus')));
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','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 Dataset Elements with Index

Access the element at index if the first argument is a numeric value.

```
el = logsout.get(1);
```

Access Dataset Elements with Characters

Access the element whose name matches `name`.

```
el = logsout.get('name');
```

Access Dataset Elements with Cell Array

Return a dataset if the first argument is a cell array with a string as the first element.

```
ds = logsout.get({'my_name'});
```

- “Migrate Scripts That Use ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use ModelDataLogs API”

See Also

Simulink.SimulationData.Dataset.addElement | Simulink.SimulationData.Dataset.concat | Simulink.SimulationData.Dataset.find | Simulink.SimulationData.Dataset.getElementNames | Simulink.SimulationData.Dataset.numElements | Simulink.SimulationData.Dataset.setElement | Simulink.SimulationData.Dataset | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Signal`

Introduced in R2011a

getElementNames

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Return names of all elements in dataset

Syntax

```
element_list = getElementNames(dataset)
```

Description

`element_list = getElementNames(dataset)` returns the names of all of the elements in the `Simulink.SimulationData.Dataset` object.

Input Arguments

`dataset` — Data set

`SimulationData.Dataset` object

The data set from which to the element name.

Output Arguments

`element_list` — Data set

cell array

Data set, returned as a cell array of the strings containing names of all of the elements of the dataset.

Examples

Return Names of Elements

Return the names of the elements for the `topOut` data set (the signal logging data).

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_mdlref_counter_bus')));
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_bus_logging')));
sim('ex_bus_logging')
el_names = topOut.getElementNames

el_names =

    'COUNTERBUS'
    'OUTPUTBUS'
    'INCREMENTBUS'
    'inner_bus'
```

- “Migrate Scripts That Use ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use ModelDataLogs API”

See Also

Simulink.SimulationData.Dataset.addElement | Simulink.SimulationData.Dataset.concat | Simulink.SimulationData.Dataset.find | Simulink.SimulationData.Dataset.get | Simulink.SimulationData.Dataset.numElements | Simulink.SimulationData.Dataset.setElement | Simulink.SimulationData.Dataset | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Signal`

Introduced in R2011a

find

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Get element or collection of elements from dataset

Syntax

```
[datasetOut,retIndex]=find(datasetIn,Name,Value,…)

[datasetOut,retIndex]=find(datasetIn,Name,Value,'-logicaloperator',…
Name,Value,…)

[datasetOut,retIndex]=find(datasetIn,'-regexp',Name,Value,…)
```

Description

`[datasetOut,retIndex]=find(datasetIn,Name,Value,…)` returns a `Simulink.SimulationData.Dataset` object and indices of the elements whose property values match the specified property names and values. Specify optional comma-separated pairs of `Name,Value` properties. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair properties in any order as `Name1,Value1,...,NameN,ValueN`.

`[datasetOut,retIndex]=find(datasetIn,Name,Value,'-logicaloperator',…Name,Value,…)` applies the logical operator to the matching property value. You can combine multiple logical operators. Logical operator can be one of:

- `-or`
- `-and`

If you do not specify an operation, the method assumes `-and`.

`[datasetOut,retIndex]=find(datasetIn,'-regexp',Name,Value,…)` matches elements using regular expressions as if the value of the property is passed to the `regexp` function as:

`regexp(element.Name,Value)`
The method applies regular expression matching to the name-value pairs that appear after `-regexp`. If there is no `-regexp`, the method matches elements as if the value of the property is passed as:

`isequal(element.Name,Value)`

For more information on `-regexp`, see "`-regexp` With Multiple Block Paths" on page 5-703.

`-regexp` With Multiple Block Paths

`-regexp` works with properties of type char. To specify multiple block paths, you can use `Simulink.SimulationData.BlockPath` and `Simulink.BlockPath`. For example, when a signal is logged in a referenced model, you can use `Simulink.SimulationData.BlockPath` to specify multiple block paths.

The method returns elements that contain a **BlockPath** property where one or more of the individual block paths match the specified `Value` path when you use:

- `-regexp` with the **BlockPath** `Name` property.
- `Value` as a string or scalar object of type `Simulink.SimulationData.BlockPath` with one block path

Input Arguments

`datasetIn` — `SimulationData.Dataset`
`SimulationData.Dataset` object

`SimulationData.Dataset` object in which to search for matching elements.

`Name` — Name of property
string

Name of property to find in the element.

`Value` — Value of property
string | double | `Simulink.SimulationData.BlockPath`

Value of property to find in the element.

Output Arguments

`datasetOut` — `SimulationData.Dataset` data set
`SimulationData.Dataset`

`SimulationData.Dataset` object that contains the elements that match the specified criteria. If there is no matching `SimulationData.Dataset` object, the returned `SimulationData.Dataset` object contains no elements.

`retIndex` — Indices
vector

Indices of the elements `datasetIn` that match the specified criteria.

Examples

Find Block Path

Find a specific block path (specified by string) and port index.

```
dsOut = find(dsIn, 'BlockPath', 'vdp/x1', 'PortIndex', 1)
```

Find Elements

Find elements that have either name or propagated name as `InValve`.

```
dsOut = find(dsIn, 'Name', 'InValve', '-or', 'PropagatedName', 'InValve')
dsOut = find(dsIn, '-regex','Name', 'In*', '-or', …
                   '-regex','PropagatedName', 'In*')
```

Find and Change Element

Find and replace all elements containing `specified_name` with a `new_name`.

```
[dsOut,idxInDs] = find(ds, 'specified_name');
for idx=1: length(idxInDs)
  % process each element
   elm = get(dsOut, idx);
   elm.Name= 'New_Name'
   dsIn = setElement(dsIn, idxInDs(idx), elm);
```

```
end
```

Find Signals in subSys Using -regexp

Find all signals logged in a `subSys` using `-regexp`.

```
dsOut = find(dsIn, '-regexp', 'BlockPath', 'mdl/subSys/.*')
```

Find Signals in Referenced Model

Find all signals logged in the Model block.

```
dsOut = find(dsIn, '-regexp', 'BlockPath', 'refmdl/ModelBlk')
```

- "Migrate Scripts That Use ModelDataLogs API"
- "Export Signal Data Using Signal Logging"
- "Log Data Stores"
- "Convert Logged Data to Dataset Format"
- "Migrate Scripts That Use ModelDataLogs API"
- "Access Logged Data from Persistent Storage"

See Also

Simulink.SimulationData.Dataset.addElement | Simulink.SimulationData.Dataset.concat | Simulink.SimulationData.Dataset.get | Simulink.SimulationData.Dataset.getElementNames | Simulink.SimulationData.Dataset.numElements | Simulink.SimulationData.Dataset.setElement | Simulink.SimulationData.Dataset | `findobj` | `regexp` | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Signal`

Introduced in R2015b

numElements

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Get number of elements in data set

Syntax

```
length = numElements(dataset)
```

Description

`length = numElements(dataset)` gets the number of elements in the top-level dataset. To get the number of elements of a nested data set, use `numElements` with the nested data set.

Input Arguments

`dataset` — Data set

`SimulationData.Dataset` object

The data set from which to get the number of elements.

Output Arguments

`length` — Number of elements

double

Number of elements, returned as a double.

Examples

Get Number of Elements

Get the number of elements in the signal logging data set for the `ex_bus_logging`.

```
length = topOut.numElements()
```

- “Migrate Scripts That Use ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use ModelDataLogs API”

See Also

Simulink.SimulationData.Dataset.addElement | Simulink.SimulationData.Dataset.concat | Simulink.SimulationData.Dataset.find | Simulink.SimulationData.Dataset.get | Simulink.SimulationData.Dataset.getElementNames | Simulink.SimulationData.Dataset.setElement | Simulink.SimulationData.Dataset | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Signal`

Introduced in R2011a

setElement

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Change element stored at specified index

Syntax

```
dataset = setElement(dataset,index,... element)
dataset = setElement(index,element, name)
```

Description

`dataset = setElement(dataset,index,... element)` changes the element stored at the specified index, for an existing index. If `index` is one greater than the number of elements in the data set, the function adds the element at the end of the data set.

`dataset = setElement(index,element, name)` changes the element stored at the specified index and gives it the name that you specify. You can use `name` to identify an element that does not have a name. If the signal already has a name, the element instead uses the name you specify by using the `name` argument.

Input Arguments

`dataset` — Data set

`SimulationData.Dataset` object

The data set for which to set the element.

`index` — Index

scalar

Index for the added element, specified as a scalar numeric value. The value must be between 1 and the number of elements plus 1.

`element` — Element to replace existing element

`Simulink.SimulationData.Signal` object | `Simulink.SimulationData.DataStoreMemory` object

Element to replace existing element or to add to the data set, specified as a `Simulink.SimulationData.Signal` object or `Simulink.SimulationData.DataStoreMemory` object.

`name` — Element name

string

Element name, returned as a string.

Output Arguments

`dataset` — Data set

string

Data set in which you change or add an element, specified as a string.

Examples

Set Element Name

Set element name.

```
ds = Simulink.SimulationData.Dataset
element1 = Simulink.SimulationData.Signal
element1.Name = 'A'
ds = ds.addElement(element1)
element2 = Simulink.SimulationData.Signal
element2.Name = 'B'
elementNew = Simulink.SimulationData.Signal
ds = ds.setElement(2,elementNew,'B1')
ds
```

```
ds =

  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData
```

```
Characteristics:
            Name: 'topOut'
  Total Elements: 2

Elements:
  1: 'A'
  2: 'B1'

Use getElement to access elements by index, name or
block path.

Methods, Superclasses
```

- “Migrate Scripts That Use ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Migrate Scripts That Use ModelDataLogs API”

See Also

Simulink.SimulationData.Dataset.addElement | Simulink.SimulationData.Dataset.concat | Simulink.SimulationData.Dataset.find | Simulink.SimulationData.Dataset.get | Simulink.SimulationData.Dataset.getElementNames | Simulink.SimulationData.Dataset.numElements | Simulink.SimulationData.Dataset | `Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.DataStoreMemory` | `Simulink.SimulationData.Signal`

Introduced in R2011a

coder.BuildConfig class

Package: coder

Build context during code generation

Description

The code generation software creates an object of this class to facilitate access to the *build context*. The build context encapsulates the settings used by the code generation software including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

Use `coder.BuildConfig` methods in the methods that you write for the `coder.ExternalDependency` class.

Construction

The code generation software creates objects of this class.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods

This example shows how to use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods. In this example, you use:

- `coder.BuildConfig.isMatlabHostTarget` to verify that the code generation target is the MATLAB host. If the host is not MATLAB report an error.
- `coder.BuildConfig.getStdLibInfo` to get the link-time and run-time library file extensions. Use this information to update the build information.

Write a class definition file for an external library that contains the function `adder`.

```
%=============================================================
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=============================================================

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(ctx)
            if  ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end

        function updateBuildInfo(buildInfo, ctx)
            [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

            % Header files
            hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
```

```
    buildInfo.addIncludePaths(hdrFilePath);

    % Link files
    linkFiles = strcat('adder', linkLibExt);
    linkPath = hdrFilePath;
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    % Non-build files
    nbFiles = 'adder';
    nbFiles = strcat(nbFiles, execLibExt);
    buildInfo.addNonBuildFiles(nbFiles,'','');
end

%API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % running in generated code, call library function
        coder.cinclude('adder.h');

        % Because MATLAB Coder generated adder, use the
        % housekeeping functions before and after calling
        % adder with coder.ceval.
        % Call initialize function before calling adder for the
        % first time.

        coder.ceval('adder_initialize');
        c = 0;
        c = coder.ceval('adder', a, b);


        % Call the terminate function after
        % calling adder for the last time.

        coder.ceval('adder_terminate');
    end
end
```

```
    end
end
```

See Also

coder.ExternalDependency | `coder.target`

Introduced in R2013b

coder.ExternalDependency class

Package: coder

Interface to external code

Description

`coder.ExternalDependency` is an abstract class for encapsulating the interface between external code and MATLAB code intended for code generation. You define classes that derive from `coder.ExternalDependency` to encapsulate the interface to external libraries, object files, and C/C++ source code. This encapsulation allows you to separate the details of the interface from your MATLAB code. The derived class contains information about external file locations, build information, and the programming interface to external functions.

To define a class, `myclass`, make the following line the first line of your class definition file:

```
classdef myclass < coder.ExternalDependency
```

You must define all of the methods listed in "Methods" on page 5-716. These methods are static and are not compiled. When you write these methods, use `coder.BuildConfig` methods to access build information.

You also define methods that call the external code. These methods are compiled. For each external function that you want to call, write a method to define the programming interface to the function. In the method, use `coder.ceval` to call the external function. Suppose you define the following method for a class named `AdderAPI`:

```
function c = adder(a, b)
    coder.cinclude('adder.h');
    c = 0;
    c = coder.ceval('adder', a, b);
end
```

This method defines the interface to a function `adder` which has two inputs `a` and `b`. In your MATLAB code, call `adder` this way:

```
y = AdderAPI.adder(x1, x2);
```

Methods

Examples

Encapsulate the interface to an external C dynamic linked library

This example shows how to encapsulate the interface to an external C dynamic linked library using `coder.ExternalDependency`.

Write a function `adder` that returns the sum of its inputs.

```
function c = adder(a,b)
    %#codegen
    c = a + b;
end
```

Generate a library that contains `adder`.

```
codegen('adder','-args', {-2,5}, '-config:dll', '-report');
```

Write the class definition file `AdderAPI.m` to encapsulate the library interface.

```
%=============================================================
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=============================================================

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(ctx)
            if  ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
```

```
end

function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

    % Header files
    hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
    buildInfo.addIncludePaths(hdrFilePath);

    % Link files
    linkFiles = strcat('adder', linkLibExt);
    linkPath = hdrFilePath;
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    % Non-build files
    nbFiles = 'adder';
    nbFiles = strcat(nbFiles, execLibExt);
    buildInfo.addNonBuildFiles(nbFiles,'','');
end

%API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % running in generated code, call library function
        coder.cinclude('adder.h');

        % Because MATLAB Coder generated adder, use the
        % housekeeping functions before and after calling
        % adder with coder.ceval.
        % Call initialize function before calling adder for the
        % first time.

        coder.ceval('adder_initialize');
        c = 0;
        c = coder.ceval('adder', a, b);
```

```
                % Call the terminate function after
                % calling adder for the last time.

                coder.ceval('adder_terminate');
            end
        end
    end
end
```

Write a function `adder_main` that calls the external library function `adder`.

```
function y = adder_main(x1, x2)
    %#codegen
    y = AdderAPI.adder(x1, x2);
end
```

Generate a MEX function for `adder_main`. The MEX Function exercises the `coder.ExternalDependency` methods.

```
codegen('adder_main', '-args', {7,9}, '-report')
```

Copy the library to the current folder using the file extension for your platform.

For Windows, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.dll'));
```

For Linux, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.so'));
```

Run the MEX function and verify the result.

```
adder_main_mex(2,3)
```

See Also

coder.BuildConfig | `coder.ceval` | `coder.cinclude` | `coder.updateBuildInfo`

More About

- “Encapsulating the Interface to External Code”
- “Best Practices for Using coder.ExternalDependency”

Introduced in R2013b

Scope Configuration Properties

Control scope appearance and behavior

Description

Scope configuration properties control the appearance and behavior of a Scope block. Create a scope configuration object with `get_param`, and then change property values using the object with dot notation.

```
open_system('vdp')
myScopeConfiguration = get_param('vdp/Scope','ScopeConfiguration')
myScopeConfiguration.NumInputPorts = '2'
```

Properties

`ActiveDisplay` — Display for setting display-specific properties

`'1'` (default) | positive integer string

Display for setting display-specific properties, specified as a positive integer string. The number of a display corresponds to its column-wise placement index.

Dependency: Setting this property selects the display for setting the properties `ShowGrid`, `ShowLegend`, `Title`, `PlotAsMagnitudePhase`, `YLabel`, and `YLimits`.

Block Configuration Property: **Active display**

`AxesScaling` — How to scale y-axes

`'Manual'` (default) | `'Auto'` | `'Updates'`

How to scale *y*-axes, specified as one of these values:

- `'Manual'` — Manually scale *y*-axes with the **Scale Y-axis Limits** button.
- `'Auto'` — Scale *y*-axes during and after simulation.
- `'Updates'` — Scale *y*-axes after specified number of block updates (time intervals).

Dependency: If this property is set to `'Updates'`, also specify the property `AxesScalingNumUpdates`.

Block Configuration Property: **Axes scaling**

`AxesScalingNumUpdates` — Number of updates before scaling y-axes

`'10'` (default) | positive integer string

Number of updates before scaling y-axes. Specified as a positive integer string.

Dependency: Activate this property by setting `AxesScaling` to `'Updates'`.

Block Configuration Property: **Number of updates**

`DataLogging` — Save scope data

`false` (default) | `true`

Save scope data to a variable in the MATLAB workspace, specified as one of these values:

- `false` — Inactivate logging and logging properties.
- `true` — Activate logging and logging properties.

This property does not apply to floating scopes and scope viewers.

Dependency: If this property is set to `true`, also specify the properties `DataLoggingVariableName` and `DataLoggingSaveFormat`.

Block Configuration Property: **Log data to workspace**

`DataLoggingVariableName` — Variable name for saving scope data

`'ScopeData'` (default) | character string

Variable name for saving scope data in the MATLAB workspace, specified as a character string. This property does not apply to floating scopes and scope viewers.

Dependency: Activate this property by setting `DataLogging` to `true`.

Block Configuration Property: **Variable name**

`DataLoggingSaveFormat` — Variable format for saving scope data

`'Dataset'` (default) | `'Structure With Time'` | `'Structure'` | `'Array'`

Variable format for saving scope data to the MATLAB workspace, specified as one of these values:

- `'Dataset'` — Save data as a dataset object. This format does not support variable-size data, MAT-file logging, or external mode archiving. See Simulink.SimulationData.Dataset.

- `'StructureWithTime'` — Save data as a structure with associated time information. This format does not support single- or multiport frame-based data, or multirate data.
- `'Structure'` — Save data as a structure. This format does not support multi-rate data.
- `'Array'` — Save data as an array with associated time information. This format does not support multiport sample-based data, single or multiport frame-based data, variable-size data, or multi-rate data.

This property does not apply to floating scopes and scope viewers.

Dependency: Activate this property by setting `DataLogging` to `true`.

Block Configuration Property: **Save format**

`DataLoggingLimitDataPoints` — Limit buffered data

`false` (default) | `true`

Limit buffered data before plotting and saving data, specified as one of these values:

- `false` — Save all the data. Setting this parameter to `false` can cause an out-of-memory error.
- `true` — Save signal data at the end of a simulation.

For simulations with **Stop time** set to `inf`, always set this parameter to `true`.

Dependency: If this property is set to `true`, also specify the number of data values to plot and save with the property `DataLoggingMaxPoints`.

Block Configuration Property: **Limit data points to last**

`DataLoggingMaxPoints` — Maximum number of data values

`'5000'` (default) | positive integer string

Maximum number of data values to plot and save, specified as a positive integer string. The data values are from the end of a simulation.

Dependency: Activate this property by setting `DataLoggingLimitDataPoints` to `true`. Specifying this property limits the data values a scope plots and the data values saved in the MATLAB variable specified in `DataLoggingVariableName`.

Block Configuration Property: Text box to the right of the **Limit data points to last** check box.

`DataLoggingDecimateData` — Reduce scope data

`false` (default) | `true`

Reduce scope data before plotting and saving, specified as one of these values:

- `false` — Do not reduce buffered data.
- `true` — Reduce buffered data.

Dependency: If this property is set to `true`, also specify `DataLoggingDecimation`.

Block Configuration Property: **Decimation**

`DataLoggingDecimation` — How signal data is reduced

`'1'` (default) | positive integer string

How signal data is reduced before plotting and saving, specified as a positive integer string. The scope buffers every `N`th data point, where `N` is the decimation factor you specify. A value of `1` buffers all data values.

Dependency: Activate this property by setting `DataLoggingDecimateData` to `true`.

Block Configuration Property: Text box to the right of the **Decimation** check box.

`FrameBasedProcessing` — Frame-based processing of signals

`false` (default for Time Scope block) | `true` (default for Scope block)

Frame-based processing of signals, specified as one of these values:

- `false` — Process signal values in a channel at each time interval (sample based).
- `true` — Process signal values in a channel as a group of values from multiple time intervals (frame based). Frame-based processing is available only with discrete input signals.

Block Configuration Property: **Input processing**

`LayoutDimensions` — Number of display rows and columns

`[1 1]` (default) | `[numberOfRows numberOfColumns]`

Number of display rows and columns, specified with as a two-element vector. The maximum layout dimension is four rows by four columns.

- If the number of displays is equal to the number of ports, signals from each port appear on separate displays.
- If the number of displays is less than the number of ports, signals from additional ports appear on the last *y*-axis.

Block Configuration Property: **Layout** button to the right of the **Number of input ports** text box

`MaximizeAxes` — Maximize size of signal plots

`'Auto'` (default) | `'On'` | `'Off'`

Maximize size of signal plots, specified as one of these values:

- `'Auto'` — If `Title` and `YLabel` are not specified, maximize all plots.
- `'On'` — Maximize all plots. Values in `Title` and `YLabel` are hidden.
- `'Off‘` — Do not maximize plots.

Each of the plots expands to fit the full display. Maximizing the size of signal plots removes the background area around the plots.

Block Configuration Property: **Maximize axes**

`MinimizeControls` — Hide menu and toolbar

`false` (default) | `true`

Hide menu and toolbar, specified by one of these values:

- `false` — Display menu and toolbar.
- `true` — Hide menu and toolbar.

If you dock the scope, this property is inactive.

`Name` — Title on a scope window

`'Scope'` (default for Scope block) | `'Time Scope'` (default for Time Scope block) | character string

Title on a scope window, specified with a character string.

`NumInputPorts` — Number of input ports

`'1'` (default) | positive integer string

Number of input ports on a Scope block, specified by a positive integer string. Maximum number of input ports is 96. This property does not apply to floating scopes and scope viewers.

Block Configuration Property: **Number of input ports**

`OpenAtSimulationStart` — Open scope window

`true` (default for Time Scope block) | `false` (default for Scope block)

Open scope window, specified as one of these values:

- `true` — Open Scope when simulation starts.
- `false` — Do not open a closed Scope at the start of a simulation.

Block Configuration Property: **Open at simulation start**

`PlotAsMagnitudePhase` — Magnitude and phase plots

`false` (default) | `true`

Magnitude and phase plots, specified by one of these values:

- `false` — Display signal plot.

  If the signal is complex, plot the real and imaginary parts on the same *y*-axis (display).
- `true` — Display magnitude and phase plots.

  If the signal is real, plot the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values.

Dependency: Set `ActiveDisplay` before setting this property.

Block Configuration Property: **Plot signals as magnitude and phase**

`Position` — Size and location of Scope

`[left bottom width height]`

Size and location of Scope window, specified as a four-element vector consisting of the left, bottom, width, and height positions, in pixels.

By default, a scope window appears in the center of your screen with a width of `560` pixels and height of `420` pixels.

`ShowGrid` — Vertical and horizontal grid lines

`true` (default) | `false`

Vertical and horizontal grid lines, specified as one of these values:

- `true` — Display grid lines.
- `false` — Hide grid lines.

Dependency: Set `ActiveDisplay` property before setting this property.

Block Configuration Property: **Show grid**

`SampleTime` — Time interval

`'-1'` for inherited | positive real string

Time interval between Scope block updates during a simulation, specified as a positive real string. This property does not apply to floating scopes and scope viewers.

Block Configuration Property: **Sample Time**

`ShowLegend` — Signal legend

`false` (default) | `true`

Signal legend, specified as one of these values:

- `false` — Hide legend.
- `true` — Show legend on active display.

Names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. See the `Scope` block reference for an example.

Dependency: Set `ActiveDisplay` property before setting this property.

Block Configuration Property: **Show legend**

`ShowTimeAxisLabel` — Display or hide x-axis labels

`true` (default for Time Scope block) | `false` (default for Scope block)

Display or hide *x*-axis labels, specified as one of these values:

- `true` — Display *x*-axis labels for the active display.
- `false` — Hide *x*-axis labels.

Dependency: Set `ActiveDisplay` property before setting this property. If this property is set to `true`, also set `TimeAxisLabels`. If `TimeAxisLabels` is set to `'None'`, this property is inactive.

Block Configuration Property: **Show time-axis label**

`TimeAxisLabels` — How x-axis labels display

`'All'` (default for Time Scope block) | `'Bottom'` (default for Scope block) | `'None'`

How *x*-axis labels display, specified as one of these values:

- `'All'` — Display *x*-axis labels on all *y*-axes.
- `'Bottom'` — Display *x*-axis label only on the bottom *y*-axis.
- `'None'` — Do not display labels and deactivate `ShowTimeAxisLabel` property.

Dependency: Set `ActiveDisplay` before specifying this property. Activate this property by setting `ShowTimeAxisLabel` to `true` and setting `Maximize axes` to `'Off'`.

Block Configuration Property: **Time-axis labels**

`TimeDisplayOffset` — X-axis range offset

`'0'` (default) | real number string | `[real number string, real number string]`

X-axis range offset, specified as a real number string or vector of real number strings. For input signals with multiple channels, enter a scaler or vector of offsets.

- Scaler — Offset all channels of an input signal by the same value.
- Vector — Independently offset the channels.

Block Configuration Property: **Time display offset**

`TimeSpan` — Length of x-axis range to display

`'0'` (default) | positive real number string | `'Auto'`

Length of *x*-axis range to display, specified as one of these values:

- Positive real number — Value less than the total simulation time.
- `'Auto'` — Difference between the simulation start and stop times.

The block calculates the beginning and end times of the *x*-axis range using the `TimeDisplayOffset` and `TimeSpan` properties. For example, if you set `TimeDisplay` to `10` and the `TimeSpan` to `20`, the scope sets the *x*-axis range from `10` to `30`.

Block Configuration Property: **Time span**

`TimeSpanOverrunAction` — How to display data

`'Wrap'` (default) | `'Scroll'`

How to display data beyond the visible *x*-axis range, specified as one of these values:

- `'Wrap'` — Draw a full screen of data from left to right, clear the screen, and then restart drawing of data.
- `'Scroll'` — Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

Block Configuration Property: **Time span overrun action**

`TimeUnits` — Units to display on the x-axis

`'Metric'` (default for Time Scope block) | `'None'` (default for Scope block) | `'Seconds'`

Units to display on the *x*-axis, specified as one of these values:

- `'Metric'` — Display time units based on the length of the `TimeSpan` property.
- `'None'` — Display `Time` on the *x*-axis.
- `'Seconds'` — Display `Time (seconds)` on the *x*-axis.

Block Configuration Property: **Time units**

`Title` — Title for display

`'%<SignalLabel>'` (default) | character string

Title for a display, specified as a character string. The default value `%<SignalLabel>` uses the input signal name for the title.

Dependency: Set `ActiveDisplay` before setting this property.

Block Configuration Property: **Title**

`Visible` — Visibility of scope window

`true (default)` | `false`

Visibility of scope window, specified as one of these values:

- `true` — Scope window visible.
- `false` — Scope window hidden.

Block Configuration Property: No corresponding property

`YLabel` — Y-axis label

`''` (default) | character string

Y-axis label for active display, specified as a character string.

Dependency: Set `ActiveDisplay` before setting this property. If `PlotAsMagnitudePhase` is `true`, the value of `YLabel` is hidden and plots are labeled `Magnitude` and `Phase`.

Block Configuration Property: **Y-label**

`YLimits` — Minimum and maximum values of y-axis

`[-10 10]` (default) | `[ymin ymax]`

Minimum and maximum values of *y*-axis, specified as a two-element numeric vector.

Dependency: Set `ActiveDisplay` before setting this property. When `PlotAsMagnitudePhase` is `true`, this property specifies the *y*-axis limits for the magnitude plot. The *y*-axis limits of the phase plot are always `[-180 180]`.

Block Configuration Property: **Y-limits (Minimum)** and **Y-limits (Maximum)**

See Also

`Floating Scope` | `Scope`

Related Examples

- "Control Scopes Programmatically"

6

Model and Block Parameters

Model Parameters

In this section...

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 6-85.

Model Parameters in Alphabetical Order

Parameter	Description	Values
`AbsTol`	Specify the largest acceptable solver error, as the value of the measured state approaches zero.	string — `{'auto'}`

Parameter	Description	Values
	Set by **Absolute tolerance** on the **Solver** pane of the Configuration Parameters dialog box.	
`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 **All Parameters** tab 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'`
`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 **All Parameters** tab 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	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	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.	
`AssertControl`	Enable model verification blocks in the current model either globally or locally. Set by **Model Verification block enabling** on the **All Parameters** tab of the Configuration Parameters dialog box.	string — `{'UseLocalSettings'} | 'EnableAll' | 'DisableAll'`
`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'}`

Parameter	Description	Values
`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'}`
`BlockPriority-ViolationMsg`	Select the diagnostic action to take if Simulink software detects a block priority specification error. Set by **Block priority violation** on the **Solver Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'warning'} | 'error'`
`BlockReduction`	Enables block reduction optimization. Set by **Block reduction** on the **All Parameters** tab of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`BlockReductionOpt`	See `BlockReduction` parameter for more information.	
`BooleanDataType`	Enable Boolean mode.	string — `{'on'} | 'off'`

Parameter	Description	Values
	Set by **Implement logic signals as Boolean data (vs. double)** on the **All Parameters** tab of the Configuration Parameters dialog box.	
`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'}`
`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	string — `{'WarnAndRepair'} | 'ErrorWithoutRepair'`

Parameter	Description	Values
	Diagnostics > Connectivity pane of the Configuration Parameters dialog box.	
`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 **All Parameters** tab 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 **All Parameters** tab of the Configuration Parameters dialog box.	string — `'on' | {'off'}`

Parameter	Description	Values
`CheckForMatrix-Singularity`	See `CheckMatrixSingularityMsg` parameter for more information.	
`CheckMatrix-SingularityMsg`	Select the diagnostic action to take if the `Product` block detects a singular matrix while inverting one of its inputs in matrix multiplication mode. Set by **Division by singular matrix** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	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 **All Parameters** tab of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`CloseFcn`	Set the close callback function, which can be a command or a variable.	string — `{''}`

Parameter	Description	Values
	Set by **Model close function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	
`CompiledBusType`	Return information about whether the signal connected to a port is not a bus, or whether it is a virtual or nonvirtual bus. (Read-only) Get with the `get_param` command. Specify a port or line handle. See “View Information about Buses”.	Return values are `'NOT_BUS'`, `VIRTUAL_BUS`, and `NON_VIRTUAL_BUS`
`CompiledModelBlockNormalM`	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 **All Parameters** tab of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`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	string — `{'10*128*eps'}`

Parameter	Description	Values
	Configuration Parameters dialog box.	
`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 — `{''}`
`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	string — `'on' | {'off'}`

Parameter	Description	Values
	`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.	
`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	string — `'on' | {'off'}`

Parameter	Description	Values
	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.	
`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. Select these options in the **Reporting** tab of the Coverage Settings dialog box.	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): • `'-sRT=1'` — Show report • `'-sVT=0'` — Web view mode • `'-aTS=1'` — Include each test in the model summary • `'-bRG=1'` — Produce bar graphs in the model summary • `'-bTC=0'` — Use two color bar graphs (red, blue)

Parameter	Description	Values
		• `'-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 • `'-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**	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

Parameter	Description	Values
	unsupported blocks exist in model, respectively, on the **Options** pane of the Coverage Settings dialog box. Disable option `'e'` by selecting **Display coverage results using model coloring** on the **Results** pane of the Coverage Settings dialog box.	• `'t'` — Enable lookup table coverage • `'r'` — Enable signal range coverage • `'z'` — Enable signal size coverage • `'o'` — Enable coverage for Simulink Design Verifier blocks • `'i'` — Enable saturation on integer overflow coverage • `'b'` — Enable relational boundary coverage • `'s'` — Treat Simulink logic blocks as short-circuited • `'w'` — Warn when unsupported blocks exist in model • `'e'` — Eliminate model coloring for coverage results
`CovModelRefEnable`	If `CovModelRefEnable` is set to `on` or `all`, the Simulink Verification and Validation software generates coverage data for all referenced models. If `CovModelRefEnable` is set to `filtered`, coverage data is collected for all referenced models except those specified by the parameter `CovModelRefExcluded`. Set by **Coverage for referenced models** on	string — `'on' | {'off'} | 'all' | 'filtered'`

Parameter	Description	Values
	the **Coverage** pane of the Coverage Settings dialog box.	
`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 — `{''}`
`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	string — `{'/'}`

Parameter	Description	Values
	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**.	
`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	string — `{'covdata'}`

Parameter	Description	Values
	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** 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'`
`CovSFcnEnable`	Enables coverage for C/C++ `S-Function` blocks in your model. Enable this feature by checking **Coverage for C/C++ S-Functions** on the Coverage Settings dialog box. For more information, see "Model Coverage for C and C ++ S-Functions" in Simulink Verification and Validation documentation.	string — `'on' | {'off'}`
`Created`	Date and time model was created. Set by **Created on** on the **History** pane of the Model Properties dialog box.	string

Parameter	Description	Values
	See “Viewing and Editing the Model History Log” for more information.	
`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.	
`DataLoggingOverride`	A `Simulink.SimulationData.M` object that specifies the signal logging override settings for a model. See “Override Signal Logging Settings”.	`Simulink.SimulationData.ModelLoggin` — `{'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.	string — `{'1'}`

Parameter	Description	Values
	Set by **"Decimation"** on the **Data Import/Export** pane of the Configuration Parameters dialog box.	
`DefaultParameterBehavior`	Enable inlining of block parameters in generated code. Set by **Default parameter behavior** on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box. For more information, see Default parameter behavior.	string — `'Inlined' | {'Tunable'}`
`DefaultUnderspecifiedData`	Specify data type to use if Simulink cannot infer the type of a signal during data type propagation. Set by **"Default for underspecified data type"** on the **Optimization** pane of the Configuration Parameters dialog box.	string — `{'double'} | 'single'`
`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.	string — `{''}`

Parameter	Description	Values
	Set by **Model description** on the **Description** pane of the Model Properties dialog box.	
`Dirty`	If the parameter is `on`, the model has unsaved changes.	string — `'on' | {'off'}`
`DiscreteInherit-ContinuousMsg`	For internal use.	
`DisplayBdSearchResults`	For internal use.	
`DisplayBlockIO`	For internal use.	
`DisplayCallgraph-Dominators`	For internal use	
`DisplayCompileStats`	For internal use.	
`DisplayCondInputTree`	For internal use.	
`DisplayCondStIdTree`	For internal use.	
`DisplayErrorDirections`	For internal use.	
`DisplayInvisibleSources`	For internal use.	
`DisplaySortedLists`	For internal use.	
`DisplayVectorAnd-FunctionCounts`	For internal use.	
`DisplayVect-PropagationResults`	For internal use.	
`ExecutionContextIcon`	Show execution context bars on conditional subsystems that do not propagate execution context across the subsystem boundaries. In the Simulink Editor, set by **Execution Context Indicator** on the **Display** > **Signals & Ports** menu.	string — `'on' | {'off'}`
`ExplicitPartitioning`	Specifies whether or not to manually map tasks (explicit	string — `'on' | {'off'}`

Parameter	Description	Values
	mapping) or use the rate-based tasks.	
`ExpressionFolding`	Enables expression folding. Set by **Eliminate superfluous local variables (Expression folding)** on the **All Parameters** tab of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`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}`
`FastRestart`	Enable or disable fast restart mode.	string — `{'on'} | 'off'`

Parameter	Description	Values
	In the Simulink Editor toolbar, click the **Fast restart** button on or off.	
`FcnCallInpInside-ContextMsg`	Specifies diagnostic action to take when Simulink software must compute any function-call subsystem inputs directly or indirectly during execution of a call to a function-call subsystem. Set by **Context-dependent inputs** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	string — `{'EnableAllAsError'} | 'EnableAllAsWarning' | 'UseLocalSettings' | 'DisableAll'` **Note:** The `Use local settings` and `Disable all` settings are maintained for backward compatibility, but 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 overflow occurs during simulation.	string — `{'none'} | 'warning' | 'error'`

Parameter	Description	Values
	Set by **Detect overflow** on the **Type Conversion Diagnostics** pane of the Configuration Parameters dialog box.	
`FixptConstPrecisionLossMs`	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.	string — `{{'old_path_1', 'new_path_1'} ...`

Parameter	Description	Values
	See “Forwarding Tables” for more information.	`{'old_path_n', 'new_path_n'}}`
`ForwardingTableString`	For internal use.	
`GeneratePreprocessorCondi`	When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`). When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.	string — `{'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.	
`IgnoredZcDiagnostic`	Specify the diagnostic action to take for warnings related to zero crossing.	string — `'none' | {'warning'} | 'error'`
`InheritedTsInSrcMsg`	Message behavior when the sample time is inherited. Set by **Source block specifies -1 sample time** on the **Sample Time Diagnostics** pane of	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	the Configuration Parameters dialog box.	
`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'}`
`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)'`

Parameter	Description	Values
`InspectSignalLogs`	Enable Simulink software to display logged signals in the Simulation Data Inspector tool at the end of a simulation or whenever you pause the simulation. Set by **"Record logged workspace data in Simulation Data Inspector"** on the **Data Import/Export** pane of the Configuration Parameters dialog box.	string — `'on' | {'off'}`
`InstrumentedSignals`	Returns a `Simulink.HMI.InstrumentedSignals` object with the properties of model name and the number of signals that are marked for streaming. From this object, you can control signal streaming using the block path and output port index.	object — `Simulink.HMI.InstrumentedSignals`
`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'}`
`IntegerOverflowMsg`	Specify message behavior when an integer overflow occurs. Set by **"Wrap on overflow"** in the **Signals** section on the	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	Data Validity Diagnostics pane of the Configuration Parameters dialog box.	
`IntegerSaturationMsg`	Specify message behavior when an integer saturation occurs. Set by **"Saturate on overflow"** in the **Signals** section on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	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 when the model was last saved.	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't string — `'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.

Parameter	Description	Values
		Set to `user`, displays only links to the user libraries. 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** configuration parameter	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 workspace. Set by the **Initial state** check box on the **Data Import/Export** pane of the	string — `'on' | {'off'}`

Parameter	Description	Values
	Configuration Parameters dialog box.	
`Location`	For internal use.	
`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'}`
`LoggingFileName`	Use when you enable `LoggingToFile` parameter for logging to persistent storage. Specify the destination MAT-file for data logging.	string — `{'out.mat'}` Do not use a file name from one locale in a different locale.
`LoggingToFile`	Store logging data that uses `Dataset` format to persistent storage (MAT-file). Using a `Simulink.SimlationData.Da` object to access signal logging and states logging data loads data into the model workspace incrementally. Accessing data for other kinds of logging loads all of the data at once. Use this feature when logging large amounts of data that can cause memory issues. For details, see “Log Data Using Persistent Storage”.	string — `'on' | {'off'}`

Parameter	Description	Values
`MAModelExclusionFile`	Specifies the location of the Model Advisor exclusion file. Set by the **File Name** field on the Model Advisor Exclusion Editor dialog box.	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'}`
`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	string — `'none' | 'warning' | {'error'}`

Parameter	Description	Values
	`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.	
`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'}`
`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'}`

Parameter	Description	Values
`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	
`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 **All Parameters** tab 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.getMetad`	Structure. Fields can be strings, numeric matrices of type "double", or more structures.
`MinMaxOverflow-ArchiveData`	For internal use	
`MinMaxOverflow-ArchiveMode`	Logging type for fixed-point logging. Set by **Overwrite or merge model simulation results** in the Fixed-Point Tool.	string — `{'Overwrite'} | 'Merge'`
`MinMaxOverflowLogging`	Setting for fixed-point logging.	string — `{'UseLocalSettings'}`

Parameter	Description	Values
	Set by **Fixed-point instrumentation mode** in the Fixed-Point Tool.	`| 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`
`MinStep`	Minimum step size for the solver. Set by **Min step size** on the **Solver** pane of the Configuration Parameters dialog box.	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'`
`ModelBlockNormalModeVisib`	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.
`ModelBlockNormaModeVisibl`	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".
`ModelBrowserVisibility`	Show the Model Browser.	string — `'on' | {'off'}`

Parameter	Description	Values
	In the Simulink Editor, set by **Model Browser** on the **View** menu.	
`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`	This parameter is maintained for compatibility purposes only. Do not use this parameter. You can use the Model Advisor to identify models referenced in Accelerator mode for which Simulink ignores certain configuration parameters. 1 In the Simulink Editor, select **Analysis > Model Advisor**. 2 Select **By Task**. 3 Run the **Check diagnostic settings ignored during accelerated model reference simulation** check.	string — `{'none'} | 'warning' | 'error'` Simulink ignores this parameter if you set it to `warning` or `error`.

Parameter	Description	Values
	For more information, see “Certain Diagnostic Configuration Parameters Ignored for Models Referenced in Accelerator Mode”.	
`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'`
`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.	string — `{'none'} | 'warning' | 'error'`

Parameter	Description	Values
	Set by **Invalid root Inport/ Outport block connection** on the **Model Referencing Diagnostics** pane of the Configuration Parameters dialog box.	
`ModelReferenceMin-AlgLoopOccurrences`	Toggles the minimization of algebraic loop occurrences. Set by **Minimize algebraic loop occurrences** on the **Model Referencing** pane of the Configuration Parameters dialog box.	string — `'on' | {'off'}`
`ModelReferenceNum-InstancesAllowed`	Total number of model reference instances allowed per top model. Set by **Total number of instances allowed per top model** on the **Model Referencing** pane of the Configuration Parameters dialog box.	string — `'Zero' | 'Single' | {'Multi'}`
`ModelReferencePass-RootInputsByReference`	Toggles the passing of scalar root inputs by value. Set by "Pass fixed-size scalar root inputs by value for code generation"on the **Model Referencing** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`ModelReferenceSim-TargetVerbose`	This parameter is deprecated and has no effect. Use `AccelVerboseBuild` instead.	

Parameter	Description	Values
`ModelReferenceSymbol-NameMessage`	For referenced models, specifies diagnostic action to take when the **Maximum identifier length** does not provide enough space to make global identifiers unique across models.	string — `'none' | {'warning'} | 'error'`
`ModelReferenceTargetType`	For internal use.	
`ModelReferenceVersion-MismatchMessage`	Message shown when there is a model block version mismatch. Set by **Model block version mismatch** on the **Model Referencing Diagnostics** pane of the Configuration Parameters dialog box.	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
`ModifiedByFormat`	Format for the display of last modifier.	string — `{'%<Auto>'}`

Parameter	Description	Values
	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.	
`ModifiedComment`	Field for user comments.	string — `{''}`
`ModifiedDateFormat`	Format string used to generate the value of the `LastModifiedDate` parameter. Set by **Last saved on** on the **History** pane of the Model Properties dialog box. See “Viewing and Editing the Model History Log” for more information.	string — `{'%<Auto>'}`
`ModifiedHistory`	Area for keeping notes about the history of the model. Set by the **Model history** field on the **History** pane of the Model Properties dialog box. See “Viewing and Editing the Model History Log” for more information.	string — `{''}`

Parameter	Description	Values
	Can also be set by the **Model history** field on the **History** pane of the Model Explorer.	
`MultiTaskCondExecSysMsg`	Select the diagnostic action to take if Simulink software detects a subsystem that might cause data corruption or nondeterministic behavior. Set by **Multitask conditionally executed subsystem** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	string — `'none' | 'warning' | {'error'}`
`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'}`

Parameter	Description	Values
`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'`
`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}`
`NumStatesForStiffnessChec`	Threshold value of number of continuous states in model for stiffness calculation. If the number of continuous states in the model exceeds the `NumStatesForStiffnessChec` value, auto solver uses `ode15s`. For more information, see "Use Auto Solver to Select a Solver".	string — `{''}`
`ObjectParameters`	Names and attributes of model parameters.	structure
`Open`	For internal use.	
`OptimizeBlockIOStorage`	Enables signal storage reuse optimization.	string — `{'on'} | 'off'`

Parameter	Description	Values
	Set by **Signal storage reuse** on the **All Parameters** tab of the Configuration Parameters dialog box.	
`OutputOption`	Time step output options for variable-step solvers. Set by **Output options** parameter under **Configuration Parameters > Data Import/Export > Additional parameters**.	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'}`
`OutputTimes`	Output times set when Set by **Output options** parameter under **Configuration Parameters > Data Import/Export > Additional parameters** is set to `Produce additional output`. Set using the **Output times** parameter.	string — `{'[]'}` **Note:** If the value of **Output options** is `Produce additional output` or `Produce specified output only`, set to a value other than the default value of `'[]'`.
`PaperOrientation`	Printing paper orientation.	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	vector — `[left, bottom, width, height]`

Parameter	Description	Values
	exported as a graphic file in the units specified by `PaperUnits`.	
`PaperPositionMode`	Paper position mode. • `auto` When printing, Simulink software sizes the diagram to fit the printed page. When exporting a diagram as a graphic image, Simulink software sizes the exported image to be the same size as the diagram's normal size on screen. • `manual` When printing, Simulink software positions and sizes the diagram on the page as indicated by `PaperPosition`. When exporting a diagram as a graphic image, Simulink software sizes the exported graphic to have the height and width specified by `PaperPosition`. • `tiled` Enables tiled printing. See "Tiled Printing" for more information.	string — `{'auto'} | 'manual' | 'tiled'`
`PaperSize`	Size of `PaperType` in `PaperUnits`.	vector — `[width height]` (read only)

Parameter	Description	Values
`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'`
`ParallelModelReferenceErr`	Specify if you want the Simulink software to perform a consistency check on the parallel pool before starting a parallel build. If you set the parameter to `on`, the client and the remote workers must meet the following criteria for the parallel build to initiate: • The parallel pool is open. • The pool is `spmd` compatible. • The platform is consistent between workers and client. • The workers have a Simulink Real-Time license. • A common compiler exists across workers and client. If you set the parameter to `off`, the software displays a warning	string — `{'on'} | 'off'`

Parameter	Description	Values
	for the first condition that fails and then performs a sequential build.	
`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'}`
`ParameterPrecision-LossMsg`	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	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	Configuration Parameters dialog box.	
`ParameterTunabilityLossMs`	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.	
`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	string — `'on' | {'off'}`

Parameter	Description	Values
	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.	
`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 — `{''}`
`PostSaveFcn`	Function invoked just after this model is saved to disk. Not executed for blocks inside library links. Set by **Model post-save function** on the **Callbacks** pane of the Model Properties dialog box. See "Create Model Callbacks" for more information.	string — `{''}`
`PreLoadFcn`	Preload callback. Set by **Model pre-load function** on the **Callbacks**	string — `{''}`

Parameter	Description	Values
	pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	
`PreSaveFcn`	Function invoked just before this model is saved to disk. Not executed for blocks inside library links, except when you are breaking the link, e.g., with `save_system( A, B, 'BreakUserLinks', 'on')`. Set by **Model pre-save function** on the **Callbacks** pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.	string — `{''}`
`ProdBitPerChar`	Describes the length in bits of the C `char` data type supported by the hardware board to be used by this model. Set by **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{8}`
`ProdBitPerInt`	Describes the length in bits of the C `int` data type supported by the hardware board to be used by this model. Set by **Number of bits: int** on the **Hardware Implementation** pane of the	integer — `{32}`

Parameter	Description	Values
	Configuration Parameters dialog box.	
`ProdBitPerLong`	Describes the length in bits of the C `long` data type supported by the hardware board to be used by this model. Set by **Number of bits: long** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`
`ProdBitPerLongLong`	Describes the length in bits of the C `long` data type supported by the hardware board to be used by this model. Set by **"Number of bits: long long"** on the **Hardware Implementation** pane of the Configuration Parameters dialog box. The value of this parameter must be greater than or equal to the value of `ProdBitPerLong`.	integer — `{64}`
`ProdBitPerShort`	Describes the length in bits of the C `short` data type supported by the hardware board to be used by this model. Set by **Number of bits: short** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{16}`

Parameter	Description	Values
`ProdEndianess`	Describes the significance of the first byte of a data word of the hardware board to be used by this model. Set by **Byte ordering** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	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.	string — `{'on'} | 'off'`
`ProdHWDeviceType`	Predefined hardware device to specify the C language constraints for your microprocessor. Set by **"Device vendor"** and **Device type** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	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** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `'Floor' | 'Zero' | {'Undefined'}`

Parameter	Description	Values
`ProdLargestAtomicFloat`	Specify the largest floating-point data type that can be atomically loaded and stored on the hardware board. Set by **"Largest atomic size: floating-point"** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `'Float' | 'Double' | {'None'}`
`ProdLargestAtomicInteger`	Specify the largest integer data type that can be atomically loaded and stored on the hardware board. Set this parameter to `'LongLong'` only if the production hardware supports the C `long long` data type and you have set `ProdLongLongMode` to `'on'`. Set by **"Largest atomic size: integer"** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'Char'} | 'Short' | 'Int' | 'Long' | 'LongLong'`
`ProdLongLongMode`	Specify that your C compiler supports the C `long long` data type. Most C99 compilers support `long long`. Set by **"Support long long"** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `'on' | {'off'}`

Parameter	Description	Values
`ProdShiftRightIntArith`	Describes whether the C compiler that creates production code for this model implements a signed integer right shift as an arithmetic right shift. Set by **Shift right on a signed integer as arithmetic shift** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	string — `{'on'} | 'off'`
`ProdWordSize`	Describes the word length in bits of the hardware board to be used by this model. Set by **Number of bits: native** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.	integer — `{32}`
`Profile`	Enables the simulation profiler for this model. In the Simulink Editor, set by **Show Profiler Report** on the **Analysis** menu.	string — `'on' | {'off'}`
`PropagateSignalLabelsOutO`	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.	string — `{'on'} | 'off'`

Parameter	Description	Values
	See “Propagate all signal labels out of the model” for more information.	
`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`
		string — `'on' | {'off'}`

Parameter	Description	Values

	`CovCumulativeVarName`
	
	`CovHTMLOptions`
	
	`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** parameter under parameter under **Configuration Parameters** > **Data Import/**	string — `{'1'}`

Parameter	Description	Values
	Export > **Additional parameters**.	
`RelTol`	Relative error tolerance. Set by **Relative tolerance** on the **Solver** pane of the Configuration Parameters dialog box.	string — `{'1e-3'}`
`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`.	string — `'none' | 'warning' | {'error'}`

Parameter	Description	Values
	Set by **"rt" prefix for identifiers** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	
`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`

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.	
`SavedSinceLoaded`	Indicates whether the model has been saved since it was loaded. `'on'` indicates the model has been saved.	string — `'on' | 'off'`
`SaveFinalState`	Save final states to workspace. Set by the **Final states** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box.	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 — `{'Dataset'} | 'Structure' | 'StructureWithTime'| 'Array'`
`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'}`

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'`
`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,

Parameter	Description	Values
		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`	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'`
`SFInvalidInputDataAccess-InChartInitDiag`	Select the diagnostic action to take when a chart: • Has the `ExecuteAtInitialization` property set to `true` • Accesses input data on a default transition or associated state entry actions, which execute at chart initialization Set by **Invalid input data access in chart initialization** on the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box.	string — `'none' | {'warning'} | 'error'`
`SFNoUnconditionalDefault-TransitionDiag`	Select the diagnostic action to take when a chart does not have an unconditional default transition to a state or a junction.	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	Set by **No unconditional default transitions** on the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box.	
`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'`
`SFTransitionActionBeforeC`	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'`
`SFTransitionOutsideNatura ParentDiag`	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**	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	pane of the Configuration Parameters dialog box.	
`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'`
`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**	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	pane of the Configuration Parameters dialog box.	
`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'`
`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'}`
`ShowLinearization-Annotations`	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'}`
`ShowLineDimensions-OnError`	For internal use.	
`ShowLineWidths`	Deprecated. Use `ShowLineDimensions` instead.	

Parameter	Description	Values
`ShowLoopsOnError`	Highlight invalid loops graphically.	string — `{'on'} | 'off'`
`ShowModelReference-BlockIO`	Toggles display of I/O mismatch on block. In the Simulink Editor, set by **Block I/O Mismatch for Referenced Model** on the **Display** > **Blocks** menu.	string — `'on' | {'off'}`
`ShowModelReference-BlockVersion`	Toggles display of version on block. In the Simulink Editor, set by **Block Version for Referenced Models** on the **Display** > **Blocks** menu.	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.	string — `'on' | {'off'}`

Parameter	Description	Values
	In the Simulink Editor, set by **Storage Class** on the **Format > Signals & Ports** menu.	
`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'`

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.	string — `{'Dataset'}` `'ModelDataLogs'` is supported for backward compatibility. However, when you open a model in R2016a or later, signal logging uses `Dataset` format, regardless of the setting of this parameter.
`SignalNameFromLabel`	Propagate signal names for Bus Creator block input signals whenever you change	string — `{''}`

Parameter	Description	Values
	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.	
`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'`
`SigSpecEnsureSample-TimeMsg`	Specifies diagnostic action to take when the sample time of the source port of a signal specified by a Signal Specification block differs from the signal's destination port. Set by **Enforce sample times specified by Signal Specification blocks** on the **Sample Time Diagnostics**	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	pane of the Configuration Parameters dialog box.	
`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 **All Parameters** tab of the Configuration Parameters dialog box.	string — `'on' | {'off'}`
`SimCtrlC`	Enables responsiveness checks in code generated for MATLAB Function blocks. Set by **"Ensure responsiveness"** on the **Simulation Target** pane of the Configuration Parameters dialog box.	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.	string — `{''}`

Parameter	Description	Values
	Set by **Header file** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	
`SimCustomInitializer`	Enter code statements that execute once at the start of simulation for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by **Initialize function** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	string — `{''}`
`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	string — `{''}`

Parameter	Description	Values
	Configuration Parameters dialog box.	
`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'`
`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**	string array — `{{}}`

Parameter	Description	Values
	> **Symbols** pane of the Configuration Parameters dialog box.	
`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, SIL, PIL, or External mode. In the Simulink Editor, set by the **Simulation > Mode** menu.	string — `{'normal'} | 'accelerator' | 'rapid-accelerator' | 'external' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'`
`SimulationStatus`	Indicates simulation status.	string — `{'stopped'} | 'updating' | 'initializing' | 'running' | 'paused' | 'terminating' | 'external'`
`SimulationTime`	Current time value for the simulation.	double — `{0}`
`SimStateInterfaceChecksum`	Check to ensure that the interface checksum is identical to the model checksum before loading the SimState.	string — `'none'|'warning'|error'`
`SimStateOlderReleaseMsg`	Check to report that the SimState was generated by an earlier version of Simulink.	string — `'error'|'warning'`

Parameter	Description	Values
	In the **Diagnostics** pane of the Configuration Parameters dialog box, configure the diagnostic to allow Simulink to report the message as error or warning.	
`SimUserDefines`	Enter a space-separated list of preprocessor macro definitions to be added to the generated code for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by **Defines** on the **Simulation Target** > **Custom Code** pane of the Configuration Parameters dialog box.	string — `{''}`
`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.	string — `{''}`

Parameter	Description	Values
	Set by **Libraries** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	
`SimUserSources`	Enter a space-separated list of source files to compile and link into the target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by **Source files** on the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.	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' |'ode8'| 'ode5' | 'ode4' | 'ode3' | 'ode2' | 'ode1' | 'ode14x'`

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'`
`SolverResetMethod`	This option appears when the solver type is `Variable-step` and the solver is `ode15s (stiff/NDF)`, `ode23t (Mod.`	string — `{'Fast'} | 'Robust'`

Parameter	Description	Values
	`stiff/Trapezoidal`), or `ode23tb (stiff/TR-BDF2)`. Set by **Solver reset method** on the **Solver** pane of the Configuration Parameters dialog box.	
`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 — `{''}`
`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	string — `'none' | {'warning'}`

Parameter	Description	Values
	for more than one state in the model. Set by **State name clash** on the **Solver Diagnostics** pane of the Configuration Parameters dialog box.	
`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'`
`StiffnessThreshold`	Threshold value to determine if the model is stiff. A model is stiff if the stiffness exceeds the `StiffnessThreshold` value. The default value for this parameter is 1000. For more information, see “Use Auto Solver to Select a Solver”.	string — `{''}`
`StopFcn`	Stop simulation callback. Set by **Simulation stop function** on the **Callbacks** pane of the Model Properties dialog box.	string — `{''}`

Parameter	Description	Values
	See “Create Model Callbacks” for more information.	
`StopTime`	Simulation stop time. Set by **Stop time** on the **Solver** pane of the Configuration Parameters dialog box.	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. For more information, see “Prevent Bus and Mux Mixtures”.	string — `{'ErrorLevel1'} | 'None' | 'Warning' | 'WarnOnBusTreatedAsVector'| 'ErrorOnBusTreatedAsVector'`
`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.	integer — `{8}`
`TargetBitPerInt`	Describes the length in bits of the C `int` data type supported	integer — `{32}`

Parameter	Description	Values
	by the hardware used to test generated code.	
`TargetBitPerLong`	Describes the length in bits of the C `long` data type supported by the hardware used to test generated code.	integer — `{32}`
`TargetBitPerLongLong`	Describes the length in bits of the C `long long` data type supported by the hardware used to test generated code. The value of this parameter must be greater than or equal to the value of `TargetBitPerLong`.	integer — `{64}`
`TargetBitPerShort`	Describes the length in bits of the C `short` data type supported by the hardware used to test generated code.	integer — `{16}`
`TargetEndianess`	Describes the significance of the first byte of a data word of the hardware used to test generated code.	string — `{'Unspecified'} | 'LittleEndian' | 'BigEndian'`
`TargetFcnLib`	For internal use.	
`TargetHWDeviceType`	Describes the characteristics of the hardware used to test generated code.	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.	string — `'Floor' | 'Zero' | {'Undefined'}`
`TargetLargestAtomicFloat`	Specify the largest floating-point data type that can be	string — `'Float' | 'Double' | {'None'}`

Parameter	Description	Values
	atomically loaded and stored on the hardware used to test code.	
`TargetLargestAtomicInteger`	Specify the largest integer data type that can be atomically loaded and stored on the hardware used to test code. Set this parameter to `'LongLong'` only if the test hardware supports the C `long long` data type and you have set `TargetLongLongMode` to `'on'`.	string — `{'Char'} | 'Short' | 'Int' | 'Long' | 'LongLong'`
`TargetLongLongMode`	Specify that your C compiler supports the C long long data type. Most C99 compilers support long long.	string — `'on' | {'off'}`
`TargetShiftRightIntArith`	Describes whether the C compiler that creates test code for this model implements a signed integer right shift as an arithmetic right shift.	string — `{'on'} | 'off'`
`TargetTypeEmulationWarnSuppressLevel`	Specifies whether Simulink Coder software displays or suppresses warning messages when emulating integer sizes in rapid prototyping environments.	integer — `{0}`
`TargetWordSize`	Describes the word length in bits of the hardware used to test generated code.	integer — `{32}`
`TasksWithSamePriorityMsg`	Specifies diagnostic action to take when tasks have equal priority.	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	Set by **Tasks with equal priority** on the **Sample Time Diagnostics** pane of the Configuration Parameters dialog box.	
`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 **All Parameters** 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'}`
`TLC...`	Parameters whose names begin with TLC are used for code generation. See the Simulink Coder documentation for more information.	

Parameter	Description	Values
`ToolBar`	Has no effect in Simulink Editor. This parameter will be removed in a future release. In the Simulink Editor, hide or display all toolbars with **Toolbars** on the **View** menu or, hide or display specific toolbars using **File > Simulink Preferences > Editor Default** toolbar options.	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 — `{''}`
`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.	string — `'none' | {'warning'} | 'error'`

Parameter	Description	Values
	Set by **Unconnected block input ports** on the **Connectivity Diagnostics** pane of the Configuration Parameters dialog box.	
`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'`
`UnderSpecifiedData-TypeMsg`	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'`
`UnderspecifiedInitial-izationDetection`	Select how Simulink software handles initialization of initial conditions for conditionally executed subsystems, Merge blocks, subsystem elapsed time, and `Discrete-Time Integrator` blocks.	string — `{'classic'}` | `'simplified'`

Parameter	Description	Values
	Set by **Underspecified initialization detection** on the **All Parameters** tab of the Configuration Parameters dialog box.	
`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'`
`UnknownTsInhSupMsg`	Detect blocks that have not set whether they allow the model containing them to inherit a sample time. Set by **Unspecified inheritability of sample time** on the **Diagnostics** > **Sample Time** pane of the Configuration Parameters dialog box.	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'`

Parameter	Description	Values
`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'`
`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.	
`UseDivisionForNetSlopeCom`	Use division to handle net slope computations when simplicity and accuracy conditions are met.	string — `{'off'} | 'on' | 'UseDivisionForReciprocalsOfInteger`
`VectorMatrix-ConversionMsg`	Detect vector-to-matrix or matrix-to-vector conversions. Set by **Vector/matrix block input conversion** on the **Type Conversion Diagnostics** pane of the	string — `{'none'} | 'warning' | 'error'`

Parameter	Description	Values
	Configuration Parameters dialog box.	
`Version`	Simulink version you are currently running, e.g., `'7.6'`. If you are using a service pack, the `ver` function returns an additional digit, e.g., 7.4.1 (R2009bSP1). To get version information without loading the block diagram into memory, see `Simulink.MDLInfo`.	double (read only)
`VersionLoaded`	Simulink version that last saved the model, e.g., `'7.6'`. If you are using a service pack, the `ver` function returns an additional digit, e.g., 7.4.1 (R2009bSP1). See also `SavedSinceLoaded`. To get version information without loading the block diagram into memory, see `Simulink.MDLInfo`.	double (read only)
`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	string — `{'UseLocalSettings'} | 'DisableAll' |`

Parameter	Description	Values
	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.	`'EnableAllAsWarning'` | `'EnableAllAsError'`
`WriteAfterWriteMsg`	Specifies diagnostic action to take when the model attempts to store data in a data store twice in succession in the current time step. Set by **Detect write after write** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.	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	string — `{'Nonadaptive'}` | `'Adaptive'`

Parameter	Description	Values
	Configuration Parameters dialog box.	
`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 `PostSaveFcn` callback.

```
set_param('mymodel','PostSaveFcn','my_save_cb')
```

Common Block Properties

In this section...

About Common Block Properties

This table lists the properties common to all Simulink blocks, including block callback properties (see "Callbacks for Customized Model Behavior"). Examples of commands that change these properties follow this table (see "Examples of Setting Block Properties" on page 6-100).

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, 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
`BlockDescription`	Block description shown in the Block Properties dialog box.	string
`BlockDiagramType`	Returns `model` if it is in an open Simulink block diagram. Returns `library` if it is a Simulink library.	`'model | 'library'`
`BlockType`	Block type (read only).	string
`ClipboardFcn`	Function called when block is copied to the clipboard (**Ctrl+C**) or when the menu item **Copy** is selected.	string
`CloseFcn`	Function called when `close_system` is run on block.	string
`Commented`	Exclude block from simulation.	`{'off'} | 'on' | 'through'`

`CompiledPort-ComplexSignals`	Complexity of port signals after updating diagram. You must compile the model before querying this property. For example:
	`vdp([],[],[],'compile');`
	`d = get_param(gcb,'CompiledP`
	`vdp([],[],[],'term');`
		structure array

`CompiledPortDataTypes`	Data types of port signals after updating diagram. You must compile the model before querying this property. See `CompiledPortComplexSignal`	structure array
`CompiledPortDesignMin`	Design minimum of port signals after updating diagram. You must compile the model before querying this property. For example: `feval(gcs, [],[],[],'compile`	structure array

Property	Description	Values

	`ports = get_param(gcb,'PortH`
	`min = get_param(ports.Outpor`
	`feval(model, [],[],[],'term'`	

`CompiledPortDesignMax`	Design maximum of port signals at compile time. You must compile the model before querying this property. For example:
	
	`feval(gcs, [],[],[],'compile`
	`ports = get_param(gcb,'PortH`
	`max = get_param(ports.Outpor`
	`feval(model, [],[],[],'term'`
		structure array

`CompiledPortDimensions`	Dimensions of port signals after updating diagram. You must compile the model before querying this property. See `CompiledPortComplexSignal`	structure array
`CompiledPortFrameData`	Frame mode of port signals after updating diagram. You must compile the model before querying this property. See `CompiledPortComplexSignal`	structure array
`CompiledPortWidths`	Structure of port widths after updating diagram. You must compile the model before querying this property. See `CompiledPortComplexSignal`	structure array
`CompiledSampleTime`	Block sample time after updating diagram. You must compile the model before querying this property. See `CompiledPortComplexSignal`	vector [`sample time`, `offset time`] or cell {[`sample time` 1, `offset time` 1]; [`sample time` 2, `offset time` 2];.....[`sample time` n, `offset time` n]}

Property	Description	Values
`ContinueFcn`	Function called at the restart of a simulation (after a pause).	string
`CopyFcn`	Function called when block is copied. See "Block Callback Parameters" in the Using Simulink documentation for details.	string
`DataTypeOverrideCompiled`	For internal use.	
`DeleteFcn`	Function called when block is deleted. See "Block Callback Parameters" in the Using Simulink documentation for details.	MATLAB expression
`DestroyFcn`	Function called when block is destroyed. See "Block Callback Parameters" in the Using Simulink documentation for details.	MATLAB expression
`Description`	Description of block. Set by the **Description** field in the **General** pane of the Block Properties dialog box.	text and tokens
`Diagnostics`	For internal use.	
`DialogParameters`	List of names/attributes of block-specific parameters for an unmasked block, or mask parameters for a masked block.	structure
`DropShadow`	Display drop shadow.	`{'off'} | 'on'`
`ExtModeLoggingSupported`	Enable a block to support uploading of signal data in external mode (for example, with a scope block).	`{'off'} | 'on'`

Property	Description	Values
`ExtModeLoggingTrig`	Enable a block to act as the trigger block for external mode signal uploading.	`{'off'} | 'on'`
`ExtModeUploadOption`	Enable a block to upload signal data in external mode when the **Select all** check box on the External Signal & Triggering dialog box is not selected. A value of `log` indicates the block uploads signals. A value of `none` indicates the block does not upload signals. The value `monitor` is currently not in use. If the **Select all** check box on the External Signal & Triggering dialog box is selected, it overrides this parameter setting.	`{'none'} | 'log' | 'monitor'`
`FontAngle`	Font angle.	`'normal' | 'italic' | 'oblique' | {'auto'}`
`FontName`	Font name.	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'}`
`ForegroundColor`	Foreground color of block's icon.	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.

Property	Description	Values
		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'`.
`Handle`	Block handle.	real
`HiliteAncestors`	For internal use.	
`InitFcn`	Initialization function for a block. Created on the **Callbacks** pane of the Model Properties dialog box. For more information, see "Create Model Callbacks". On non-masked blocks, updating the diagram or running the simulation call this function.	MATLAB expression
`InputSignalNames`	Names of input signals.	cell array
`IntrinsicDialogParameters`	List of names/attributes of block-specific parameters (regardless of whether the block is masked or unmasked). Use instead of `DialogParameters` if you want block-specific parameters for masked or unmasked blocks.	structure
`IOSignalStrings`	Block paths to objects that are connected to the Signal & Scope Manager. Simulink software	list

Property	Description	Values
	saves these paths when the model is saved.	
`IOType`	Signal & Scope Manager type. For internal use.	
`LibraryVersion`	For a linked block, the initial value of this property is the `ModelVersion` of the library at the time the link was created. The value updates with increments in the model version of the library.	string — {`'1.1'`}
`LineHandles`	Handles of lines connected to block.	structure
`LinkData`	Array of details about changes to the blocks inside the link that differ between a parameterized link and its library, listing the block names and parameter values. Use `[]` to reset to deparameterized, e.g., `set_param(gcb,'linkData',[])`. See "Parameterized Links" in the Using Simulink documentation.	cell array
`LinkStatus`	Link status of block. Updates out-of-date linked blocks when queried using `get_param`. See "Check and Set Link Status Programmatically" in the Using Simulink documentation.	`'none' | 'resolved' | 'unresolved' | 'implicit' | 'inactive' | 'restore' | 'propagate' | 'propagateHierarchy' | 'restoreHierarchy'`
`LoadFcn`	Function called when block is loaded.	MATLAB expression

Property	Description	Values
`MinMaxOverflow-Logging_Compiled`	For internal use.	
`ModelCloseFcn`	Function called when model is closed. The `ModelCloseFcn` is called prior to the block's `DeleteFcn` and `DestroyFcn` callbacks, if either are set.	MATLAB expression
`ModelParamTableInfo`	For internal use.	
`MoveFcn`	Function called when block is moved.	MATLAB expression
`Name`	Block name.	string
`NameChangeFcn`	Function called when block name is changed.	MATLAB expression
`NamePlacement`	Position of block name.	`{'normal'} | 'alternate'`
`ObjectParameters`	Names/attributes of block's parameters.	structure
`OpenFcn`	Function called when this Block Parameters dialog box opens.	MATLAB expression
`Orientation`	Where block faces.	`{'right'} | 'left' | 'up' | 'down'`
`OutputSignalNames`	Names of output signals.	cell array
`Parent`	Name of the system that owns the block.	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
`PauseFcn`	Function called at the pause of a simulation.	string
`PortConnectivity`	The value of this property is an array of structures, each of which describes one of the	structure array

Property	Description	Values
	block's input or output ports. Each port structure has the following fields: • `Type` Specifies the port's type and/or number. The value of this field can be: • *n*, where *n* is the number of the port for data ports • `'enable'` if the port is an enable port • `'trigger'` if the port is a trigger port • `'state'` for state ports • `'ifaction'` for action ports • `'LConn#'` for a left connection port where # is the port's number • `'RConn#'` for a right connection port where # is the port's number • `Position` The value of this field is a two-element vector, [*x* *y*], that specifies the port's position. • `SrcBlock` Handle of the block connected to this port. This field is null for output ports	

Property	Description	Values
	and `-1` for unconnected input ports. • `SrcPort` Number of the port connected to this port, starting at zero. This field is null for both output ports and unconnected input ports. • `DstBlock` Handle of the block to which this port is connected. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports. • `DstPort` Number of the port to which this port is connected, starting at zero. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports.	
`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.	structure array

Property	Description	Values
	• `Outport` Handles of the block's output ports. • `Enable` Handle of the block's enable port. • `Trigger` Handle of the block's trigger port. • `State` Handle of the block's state port. • `LConn` Handles of the block's left connection ports (for blocks that support Physical Modeling tools). • `RConn` Handles of the block's right connection ports (for blocks that support Physical Modeling tools). • `Ifaction` Handle of the block's action port. • `Reset` Handle of the block’s reset port.	

Property	Description	Values
`PortRotationType`	Type of port rotation used by this block. This is a read-only property.	`'default' | 'physical'`
`Ports`	A vector that specifies the number of each kind of port this block has. The order of the vector's elements corresponds to the following port types: • `Inport` • `Outport` • `Enable` • `Trigger` • `State` • `LConn` • `RConn` • `Ifaction` • `Reset`	vector
`Position`	Position of block in model window. To help with block alignment, the position you set can differ from the actual block position by a few pixels. Use `get_param` to return the actual position.	vector of coordinates, in pixels: `[left top right bottom]` The origin is the upper-left corner of the model window. The maximum absolute value for a coordinate is 32767. Positive values are to the right of and down from the origin. Negative values are to the left of and up from the origin.
`PostSaveFcn`	Function called after the block is saved.	MATLAB expression
`PreCopyFcn`	Function called before the block is copied. See “Block Callback Parameters” in the Using	MATLAB expression

Property	Description	Values
	Simulink documentation for details.	
`PreDeleteFcn`	Function called before the block is deleted. See "Block Callback Parameters" in the Using Simulink documentation for details.	MATLAB expression
`PreSaveFcn`	Function called before the block is saved. See "Block Callback Parameters" in the Using Simulink documentation for details.	MATLAB expression
`Priority`	Specifies the block's order of execution relative to other blocks in the same model. Set by the **Priority** field on the **General** pane of the Block Properties dialog box.	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. Intended only for use with user written S-functions. See the section "S-Function RTWdata" in the Simulink Coder documentation for details.	
`SampleTime`	Value of the sample time parameter. See "Specify Sample Time" in the Simulink documentation for more details.	string
`Selected`	Status of whether or not block is selected.	`{'on'} | 'off'`

Property	Description	Values
`ShowName`	Display block name.	`{'on'} | 'off'`
`StartFcn`	Function called at the start of a simulation.	MATLAB expression
`StatePerturbation-ForJacobian`	State perturbation size to use during linearization. See "Change Perturbation Level of Blocks Perturbed During Linearization" in the Simulink Control Design documentation for details.	string
`StaticLinkStatus`	Link status of block. Does not update out-of-date linked blocks when queried using `get_param`. See also `LinkStatus`.	`'none' | 'resolved' | 'unresolved' | 'implicit' | 'inactive' | 'restore' | 'propagate' | 'propagateHierarchy' | 'restoreHierarchy'`
`StopFcn`	Function called at the termination of a simulation.	MATLAB expression
`Tag`	Text that appears in the block label that Simulink software generates. Set by the **Tag** field on the **General** pane of the Block Properties dialog box.	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.

```
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 6-86). The type listed for masked blocks is the value of the `MaskType` parameter (see "Mask Parameters" on page 6-234).

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`)		
`CoefficientInTFapproxima`	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'`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'}`
`ContinuousStateAttribute`	State Name	string — `{''}` | user-defined
`WrapState`	Enable wrapping of states	string — `{'off'} | 'on'`
`WrappedStateUpperValue`	Upper value of wrapped state	scalar or vector — `{'pi'}`
`WrappedStateLowerValue`	Lower value of wrapped state	scalar or vector — `{'-pi'}`
Second-Order Integrator (`SecondOrderIntegrator`)		
`ICSourceX`	Initial condition source x	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
`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

Block (Type)/Parameter	Dialog Box Prompt	Values
StateNameDXDT	State name dx/dt	string — {} | user-defined
ExternalReset	External reset	string — {'none'} | 'rising' | 'falling' | 'either'
ZeroCross	Enable zero-crossing detection	string — {'on'} | 'off'
ReinitDXDTwhenXreachesSa	Reinitialize dx/dt when x reaches saturation	string — {'off'} | 'on'
IgnoreStateLimitsAndRese	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'}
AbsoluteTolerance	Absolute tolerance	string, scalar, or vector — {'auto'} | {'-1'} | any real scalar or vector
ContinuousStateAttribute	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
ContinuousStateAttribute	State Name	string — {''} | user-defined
Transport Delay (TransportDelay)		
DelayTime	Time delay	scalar or vector — {'1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'`
`ZeroDelay`	Handle zero delay	string — `{'off'} | 'on'`
`TransDelayFeedthrough`	Direct feedthrough of input during linearization	string — `{'off'} | 'on'`
`PadeOrder`	Pade order (for linearization)	string — `{'0'}`
`ContinuousStateAttribute`	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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
PadeOrder	Pade order (for linearization)	string — {'0'}
AbsoluteTolerance	Absolute tolerance	string, scalar, or vector — {'auto'} | {'-1'} | any positive real scalar or vector
ContinuousStateAttribute	State Name	string — {''} | user-defined
Zero-Pole (ZeroPole)		
Zeros	Zeros	vector — {'[1]'}
Poles	Poles	vector — {'[0 -1]'}
Gain	Gain	vector — {'[1]'}
AbsoluteTolerance	Absolute tolerance	string, scalar, or vector — {'auto'} | {'-1'} | any positive real scalar or vector
ContinuousStateAttribute	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'}
SaturateOnIntegerOverflo	Saturate on integer overflow	string — 'off' | {'on'}

Block (Type)/Parameter	Dialog Box Prompt	Values
`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)		
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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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 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 — {'[]'}
OutDataTypeStr	Output data type	string — 'Inherit: Inherit via back propagation' | {'Inherit: Same as input'} | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16'

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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)'`
`OutputDataTypeScalingMod`	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 on integer overflow	string — `{'off'} | 'on'`
Wrap To Zero `(Wrap To Zero)` (masked subsystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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'
RemoveProtectionDelay Length	Remove protection against out-of-range delay length in generated code	string — {'off'} | 'on'
DiagnosticForOutOfRangeD	Diagnostic for out-of-range delay length	string — {'None'} | 'Warning' | 'Error'
SampleTime	Sample time (–1 for inherited)	string — {'-1'}
StateName	State name	string — {''}

Block (Type)/Parameter	Dialog Box Prompt	Values
StateMustResolveToSignal	State name must resolve to Simulink signal object	string — {'off'} | 'on'
StateStorageClass	Code generation storage class	string — {'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'
CodeGenStateStorageTypeQ	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 — {'[]'}
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)'
OutputDataTypeScalingMod	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'

Block (Type)/Parameter	Dialog Box Prompt	Values
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 — {'[]'}
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)'
OutputDataTypeScalingM	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)		

Block (Type)/Parameter	Dialog Box Prompt	Values
`CoefSource`	Coefficient source	string — `{'Dialog parameters'} | 'Input port'`
`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)'`
`CoefDataTypeStr`	Coefficients data type	string — `{'Inherit: Same word length as`

Block (Type)/Parameter	Dialog Box Prompt	Values
		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)'
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)'

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'`
`SaturateOnIntegerOverflo`	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 — `{'[]'}`
`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' |`

Block (Type)/Parameter	Dialog Box Prompt	Values
		'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)'
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'

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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'`
`SaturateOnIntegerOverflo`	Saturate on integer overflow	string — `{'off'} | 'on'`
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveToSign`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`StateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageTypeQual`	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 — `{''}`
`StateMustResolveToSign`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`StateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' |`

Block (Type)/Parameter	Dialog Box Prompt	Values
		'ImportedExtern' | 'ImportedExternPointer'
RTWStateStorageTypeQua	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'}
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| 'int8' | 'int16' | 'int32' | '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)'`
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverfl`	Saturate on integer overflow	string — `{'off'} | 'on'`
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveToSignal`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`StateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageTypeQual`	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 — `{''}`
`StateMustResolveToSigna`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`StateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageType Qualifier`	Code generation storage type qualifier	string — `{''}`
Discrete-Time Integrator (`DiscreteIntegrator`)		
`IntegratorMethod`	Integrator method	string — `{'Integration: Forward Euler'} | 'Integration: Backward Euler' | 'Integration: Trapezoidal' |`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`'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'}`
`InitialConditionSetting`	Initial condition setting	string — `{'State (most efficient)'} | 'Output' | 'Compatibility'`
`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'`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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'`
`StateStorageClass`	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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`StateIdentifier`	State name	string — `{''}`
`StateMustResolveTo SignalObject`	State name must resolve to Simulink signal object	string — `{'off'} | 'on'`
`StateStorageClass`	Code generation storage class	string — `{'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'`
`RTWStateStorageType Qualifier`	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'}`
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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 — {''}

Block (Type)/Parameter	Dialog Box Prompt	Values
StateMustResolveTo SignalObject	State name must resolve to Simulink signal object	string — {'off'} | 'on'
StateStorageClass	Code generation storage class	string — {'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'
CodeGenStateStorageTyp Qualifier	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)		

Block (Type)/Parameter	Dialog Box Prompt	Values
`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)		
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'}`
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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)		
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'}

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'`
`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)'`

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'}
InternalRulePriority	Internal rule priority for lookup table	string — {'Speed'} | 'Precision'
Direct Lookup Table (n-D) (LookupNDDirect)		
NumberOfTableDimensions	Number of table dimensions	string — '1' | {'2'} | '3' | '4'
InputsSelectThisObjectF	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]'}
DiagnosticForOutOfRange	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	

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'}
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)'

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'`
`TableMin`	Table data > Minimum	string — `{'[]'}`
`TableMax`	Table data > Maximum	string — `{'[]'}`
`IntermediateResultsData`	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 — `{'[]'}`
`InternalRulePriority`	Internal rule priority	string — `{'Speed'} | 'Precision'`
`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'`
`SaturateOnIntegerOverfl`	Saturate on integer overflow	string — `{'off'} | 'on'`
`CheckIndexInCode`	Deprecated in R2011a	

Block (Type)/Parameter	Dialog Box Prompt	Values
n-D Lookup Table, 1-D Lookup Table, 2-D Lookup Table (`Lookup_n-D`)		
`NumberOfTableDimensions`	Number of table dimensions	string — `'1' | '2' | '3' | '4'`. Default is '1' for 1-D Lookup Table, '2' for 2-D Lookup Table, '3' for n-D Lookup Table.
`Table`	Table data	string — `{'reshape(repmat([4 5 6;16 19 20;10 18 23],1,2),[3,3,2])'}`
`BreakpointsSpecificatio`	Breakpoints specification	string — `{'Explicit values'} | 'Even spacing'`
`BreakpointsForDimension`	First point	string — `{'1'}`
`BreakpointsForDimension`	First point	string — `{'1'}`
`BreakpointsForDimension`	First point	string — `{'1'}`
...	...	...
`BreakpointsForDimension`	First point	string — `{'1'}`
`BreakpointsForDimension`	Spacing	string — `{'1'}`
`BreakpointsForDimension`	Spacing	string — `{'1'}`
`BreakpointsForDimension`	Spacing	string — `{'1'}`
...	...	...
`BreakpointsForDimension`	Spacing	string — `{'1'}`
`BreakpointsForDimension`	Breakpoints 1	string — `{'[10,22,31]'}`
`BreakpointsForDimension`	Breakpoints 2	string — `{'[10,22,31]'}`
`BreakpointsForDimension`	Breakpoints 3	string — `{'[5, 7]'}`
...	...	...
`BreakpointsForDimension`	Breakpoints 30	string — `{'[1:3]'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`InterpMethod`	Interpolation method	string — `'Flat' | {'Linear'} | 'Cubic spline'`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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 — {'[]'}
BreakpointsForDimensionDataTypeStr	Breakpoints 1 > Data Type	string — {'Inherit: Same as corresponding input'}

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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)'`
`BreakpointsForDimension Min`	Breakpoints 1 > Minimum	string — `{'[]'}`
`BreakpointsForDimension Max`	Breakpoints 1 > Maximum	string — `{'[]'}`
`BreakpointsForDimension 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)'`
`BreakpointsForDimension Min`	Breakpoints 2 > Minimum	string — `{'[]'}`
`BreakpointsForDimension Max`	Breakpoints 2 > Maximum	string — `{'[]'}`
`...`	`...`	`...`
`BreakpointsForDimension 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)'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'`
`BreakpointsForDimension Min`	Breakpoints 30 > Minimum	string — `{'[]'}`
`BreakpointsForDimension Max`	Breakpoints 30 > Maximum	string — `{'[]'}`
`FractionDataTypeStr`	Fraction > Data Type	string — `{'Inherit: Inherit via internal rule'} | 'double' | 'single' | 'fixdt(1,16,0)'`
`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 — `{'[]'}`
`InternalRulePriority`	Internal rule priority	string — `{'Speed'} | 'Precision'`
`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'`

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'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round'| 'Simplest' | 'Zero'`

Block (Type)/Parameter	Dialog Box Prompt	Values
DoSatur	Saturate to max or min when overflows occur	string — {'off'} | 'on'
Prelookup (PreLookup)		
BreakpointsSpecificatio	Specification	string — {'Explicit values'} | 'Even spacing'
BreakpointsFirstPoint	First point	string — {'10'}
BreakpointsSpacing	Spacing	string — {'10'}
BreakpointsNumPoints	Number of points	string — {'11'}
BreakpointsData	Value	string — {'[10:10:110]'}
BreakpointsDataSource	Source	string — {'Dialog'} | 'Input port'
IndexSearchMethod	Index search method	string — 'Evenly spaced points' | 'Linear search' | {'Binary search'}
BeginIndexSearchUsing PreviousIndexResult	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'
DiagnosticForOutOfRange Input	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'}
BreakpointDataTypeStr	Breakpoint > Data Type	string — {'Inherit: Same as input'} | 'Inherit: Inherit from 'Breakpoint data'' | 'double' | 'single' |

Block (Type)/Parameter	Dialog Box Prompt	Values
		`'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)'`
`NumDataPoints`	Number of data points for lookup table	string — `{'(2^5)+1'}`
`OutputWordLength`	Output word length	string — `{'16'}`
`InternalRulePriority`	Internal rule priority for lookup table	string — `{'Speed'} | 'Precision'`

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'
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'}

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'
RndMeth	Integer rounding mode	string — 'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
SaturateOnInteger	Saturate on integer overflow	string — {'off'} | 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
Overflow		
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'}
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
`SaturateOnInteger Overflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
Complex to Magnitude-Angle (`ComplexToMagnitudeAngle`)		
`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)'`

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'
SaturateOnInteger Overflow	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)'
OutputDataTypeScaling Mode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock output data type setting against changes by the fixed-point tools	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'}
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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 — {'[]'}
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)'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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'}`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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'`
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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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 — `{'[]'}`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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 — `{'[]'}`
`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`)		

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'`
`SaturateOnInteger Overflow`	Saturate on integer overflow	string — `'off' | {'on'}`
`IntermediateResults DataTypeStr`	Intermediate results data type	string — `{'Inherit: Inherit via internal rule'} | 'Inherit: Inherit from input' | 'Inherit: Inherit from output' | 'double' | 'single'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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'}`
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 — `{'[]'}`

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'}
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}`
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`

Block (Type)/Parameter	Dialog Box Prompt	Values
		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'
SaturateOnInteger Overflow	Saturate on integer overflow	string — 'off' | {'on'}
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'}
Squeeze (Squeeze) (masked subsystem)		
None	None	None

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}
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'

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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 (`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 — `{'[]'}`

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'}`
`InputSameDT`	Require all inputs to have the same data type	string — `{'off'} | 'on'`
`AccumDataTypeStr`	Accumulator data type	string — `{'Inherit: Inherit via internal rule'} | 'Inherit:`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`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'`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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`)		
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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'}`
`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 — `{''}`

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 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 — `{''}`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'`
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)		
`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 — `{''}`

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'`
`PropagateVarSize`	Propagate sizes of variable-size signals	string — `{'Only when execution is resumed'} | 'During execution'`
Atomic Subsystem (`SubSystem`)		

Block (Type)/Parameter	Dialog Box Prompt	Values
ShowPortLabels	Show port labels **Note:** The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	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 — {''}
PermitHierarchical Resolution	Permit hierarchical resolution	string — {'All'} | 'ExplicitOnly' | 'None'
TreatAsAtomicUnit	Treat as atomic unit	string — 'off' | {'on'}
TreatAsGroupedWhenPropag	Treat as grouped when propagating variant conditions	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	Function packaging	string — {'Auto'} | 'Inline' | 'Nonreusable function' | 'Reusable function'

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWFcnNameOpts`	Function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	File name (no extension)	string — `{''}`
`FunctionInterfaceSpec`	Function interface This parameter requires a license for Embedded Coder software and an ERT-based system target file.	string — `{'void_void'} | 'Allow arguments'`
`FunctionWithSeparateData`	"Function with separate data" on page 1-1924 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 1-1926 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
`RTWMemSecFuncExecute`	"Memory section for execution functions" on page 1-1927 This parameter requires a license for Embedded Coder	string — `{'Inherit from model'}` | `'Default'` | list of memory sections from model's package

Block (Type)/Parameter	Dialog Box Prompt	Values
	software and an ERT-based system target file.	
`RTWMemSecDataConstants`	"Memory section for constants" on page 1-1928 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 1-1930 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 1-1932 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'`
`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'`
`SimViewingDevice`	No dialog box prompt	string — `{'off'}` | `'on'`

Block (Type)/Parameter	Dialog Box Prompt	Values
	If set to `'on'`, designates the block as a Signal Viewing Subsystem — an atomic subsystem that encapsulates processing and viewing of signals received from the target system in External mode. For more information, see "Signal Viewing Subsystems".	
`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 — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`
`MinAlgLoopOccurrences`	Minimize algebraic loop occurrences	string — `{'off'} | 'on'`

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`	Function packaging	string — `'Auto' | 'Inline' | 'Nonreusable function' | {'Reusable function'}`
`RTWFcnNameOpts`	Function name options	string — `'Auto' | {'Use subsystem name'} | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `'Auto' | {'Use subsystem name'} | 'Use function name' | 'User specified'`
`RTWFileName`	File name (no extension)	string — `{''}`
`RTWMemSecFuncInitTerm`	"Memory section for initialize/terminate functions" on page 1-1926 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
`RTWMemSecFuncExecute`	"Memory section for execution functions" on page 1-1927 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 1-1928	string — `{'Inherit from model'} | 'Default'` | list of

Block (Type)/Parameter	Dialog Box Prompt	Values
	This parameter requires a license for Embedded Coder software and an ERT-based system target file.	memory sections from model's package
`RTWMemSecDataInternal`	"Memory section for internal data" on page 1-1930 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 1-1932 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'`
`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	string — `'none' | {'FromPortIcon'} |`

Block (Type)/Parameter	Dialog Box Prompt	Values
	Note: The values `'off'` and `'on'` are for backward compatibility only and should not be used in new models or when updating existing models.	`'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 — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`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`	Function packaging	string — `{'Auto'} | 'Inline' | 'Nonreusable function' | 'Reusable function'`
`RTWFcnNameOpts`	Function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `{'Auto'} | 'Use subsystem name' | 'Use`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`function name' | 'User specified'`
`RTWFileName`	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'`
`SimViewingDevice`	No dialog box prompt If set to `'on'`, designates the block as a Signal Viewing Subsystem — an atomic subsystem that encapsulates processing and viewing of signals received from the target system in External mode. For more information, see “Signal Viewing Subsystems”.	string — `{'off'} | 'on'`
`DataTypeOverride`	No dialog box prompt Specifies data type used to override fixed-point data types. Set by **Data type override** on the Fixed-Point Tool.	string — `{'UseLocalSettings'} | 'ScaledDouble' | 'Double' | 'Single' | 'Off'`
`MinMaxOverflowLogging`	No dialog box prompt Setting for fixed-point instrumentation. Set by **Fixed-**	string — `{'UseLocalSettings'} | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`

Block (Type)/Parameter	Dialog Box Prompt	Values
	point instrumentation mode on the Fixed-Point Tool.	
`IsSubsystemVirtual`	No dialog box prompt	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 — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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`	Function packaging	string — `{'Auto'} | 'Inline' | 'Nonreusable function' | 'Reusable function'`
`RTWFcnNameOpts`	Function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		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'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`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`	Function packaging	string — `{'Auto'} | 'Inline' | 'Nonreusable function' | 'Reusable function'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWFcnNameOpts`	Function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	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
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
`OutputConcatenationDimer`	Concatenation dimension of output signal	cell array of strings
For Iterator (`ForIterator`)		

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
ShowIterationPort	Show iteration variable	string — 'off' | {'on'}
IndexMode	Index mode	string — 'Zero-based' | {'One-based'}
IterationVariable DataType	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 — {''}
PermitHierarchical Resolution	Permit hierarchical resolution	string — {'All'} | 'ExplicitOnly' | 'None'
TreatAsAtomicUnit	Treat as atomic unit	string — 'off' | {'on'}

Block (Type)/Parameter	Dialog Box Prompt	Values
`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`	Function packaging	string — `{'Auto'} | 'Inline' | 'Nonreusable function' | 'Reusable function'`
`RTWFcnNameOpts`	Function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		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 — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`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`	Function packaging	string — `{'Auto'} | 'Inline' | 'Nonreusable`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`function' | 'Reusable function'`
`RTWFcnNameOpts`	Function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	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
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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
ErrorFcn	Name of error callback function	string — {''}
PermitHierarchical Resolution	Permit hierarchical resolution	string — {'All'} | 'ExplicitOnly' | 'None'
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	Function packaging	string — {'Auto'} | 'Inline' | 'Nonreusable function' | 'Reusable function'

Block (Type)/Parameter	Dialog Box Prompt	Values
`RTWFcnNameOpts`	Function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`
`RTWFcnName`	Function name	string — `{''}`
`RTWFileNameOpts`	File name options	string — `{'Auto'} | 'Use subsystem name' | 'Use function name' | 'User specified'`
`RTWFileName`	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' | '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'`
`LatchByDelaying OutsideSignal`	Latch input by delaying outside signal	string — `{'off'} | 'on'`
`LatchInputFor FeedbackSignals`	Latch input for feedback signals of function-call subsystem outputs	string — `{'off'} | 'on'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`Interpolate`	Interpolate data	string — `'off' | {'on'}`
`UseBusObject`	Specify properties via bus object	string — `{'off'} | 'on'`
`BusObject`	Bus object for specifying bus properties	string — `{'BusObject'}`
`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'`
`Unit`	Specify physical unit of the input signal to the block.	string — `{‘inherit’}| '<Enter unit>'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
`SamplingMode`	Sampling mode	string — `{'auto'} | 'Sample based' | 'Frame based'`
Model (`ModelReference`)		
`ModelNameDialog`	The name of the referenced model exactly as you typed it in, with any surrounding	string — `{'<Enter Model Name>'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
	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`.	
`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.
`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.

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'`
`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.ParameterArgument`	string — `{''}`

Block (Type)/Parameter	Dialog Box Prompt	Values
	`Values` – The values to supply for the model arguments when this variant is the active variant.	
	`variant.SimulationMode` – The execution mode to use when this variant is the active variant.	string — `{'Accelerator'} | 'Normal' | 'Processor-in-the-loop (PIL)'`
`VariantConfigurationObje`	Specifies the variant configuration object that is associated with the model.	string — `{''}` The value is an empty string if no configuration object is associated; otherwise, it is the name of a `Simulink.VariantConfigurationData` object.
`OverrideUsingVariant`	Whether to override the variant conditions and make a specified variant the active variant, and if so, the name of that variant.	string — `{''}` The value is the empty string if no overriding variant object is specified; or the name of the overriding object.
`ActiveVariant`	The variant that is currently active, either because its variant condition is `true` or `OverrideUsingVariant` has overridden the variant conditions and specified this variant.	string — `{''}` The value is the empty string if no variant is active; or the name of the active variant.
`GeneratePreprocessor Conditionals`	Locally controls whether generated code contains preprocessor conditionals. This parameter applies only to Simulink Coder code generation and has no effect on the behavior of a model in Simulink.	string — `{'off'} | 'on'`

Block (Type)/Parameter	Dialog Box Prompt	Values
	The parameter is available only for ERT targets. For more information, see "Variant Systems".	
`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'`
`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>'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`Unit`	Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.	string — `{'inherit'}| '<Enter unit>'`
`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 — `{'[]'}`
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 — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `{'off'} | 'on'`

Block (Type)/Parameter	Dialog Box Prompt	Values
TreatAsGroupedWhenPropag	Treat as grouped when propagating variant conditions	string — 'off' | {'on'}
VariantControl	Variant control	string — {'Variant'} | '(default)'
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' | 'Nonreusable 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-**	string — {'UseLocalSettings'} | 'MinMaxAndOverflow'

Block (Type)/Parameter	Dialog Box Prompt	Values
	point instrumentation mode on the Fixed-Point Tool.	`| '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}'}`
`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 — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`TreatAsAtomicUnit`	Treat as atomic unit	string — `'off' | {'on'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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' | 'Nonreusable 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'`
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		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 — `{'[]'}`
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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 — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`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' | 'Nonreusable function' | 'Reusable function'`
`RTWFcnNameOpts`	Code generation function name options	string — `{'Auto'} | 'Use subsystem name' | 'User specified'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'`
`IsSubsystemVirtual`		boolean — `{'off'} | 'on'` Read-only
Unit Conversion		
`OutDataTypeStr`	Output data type	string — `{'Inherit via internal rule'} | 'Inherit via back propagation'`
Unit System Configuration		
`AllowAllUnitSystems`	Allow or restrict unit systems.	boolean — `{'on'} | 'off'`
`UnitSystems`	Displays allowed unit system.	cell array of strings — `{'SI','English',SI (extended)','CGS'}`
While Iterator (`WhileIterator`)		

Block (Type)/Parameter	Dialog Box Prompt	Values
`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 — `{''}`
`TemplateBlock`	Template block	string — `{''}`
`MemberBlocks`	Member blocks	string — `{''}`
`Permissions`	Read/Write permissions	string — `{'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'`
`ErrorFcn`	Name of error callback function	string — `{''}`
`PermitHierarchical Resolution`	Permit hierarchical resolution	string — `{'All'} | 'ExplicitOnly' | 'None'`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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' | 'Nonreusable 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'`
`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 — `{'[]'}`
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| 'Nearest' | 'Round' | 'Simplest' | 'Zero'`
`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'}`

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'}
BiasBase	2.2.1. Bias: Base	string — {'Bias1'} | 'Bias2' | 'max([Bias1 Bias2])' | 'min([Bias1 Bias2])' | 'Bias1*Bias2' | 'Bias1/Bias2' | 'Bias1+Bias2' | 'Bias1-Bias2'

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}
OutPortSampleTimeOpt	Output port sample time options	string — {'Specify'} | 'Inherit' | 'Multiple of input port sample time'
OutPortSampleTimeMultipl	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)		

Block (Type)/Parameter	Dialog Box Prompt	Values
`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>'`
`BusOutputAsStruct`	Require nonvirtual bus	string — `{'off'} | 'on'`
`Unit`	Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.	string — `{'inherit'}| '<Enter unit>'`
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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)		
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 — {'{}'}

Block (Type)/Parameter	Dialog Box Prompt	Values
Bus Creator (`BusCreator`)		
`InheritFromInputs`	Override bus signal names from inputs	string — `{'on'} | 'off'` If set to `'on'`, overrides bus signal names from inputs. Otherwise, inherits bus signal names from a bus object.
`Inputs`	Number of inputs	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>'`
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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
InitialValue	Initial value	string — {'0'}
StateMustResolveTo SignalObject	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 — {''}
DataLoggingLimit DataPoints	Limit data points to last	string — 'off' | {'on'}
DataLoggingMaxPoints	Limit data points to last	non-zero integer {5000}
DataLoggingDecimateDat	Decimation	string — 'off' | {'on'}
DataLoggingLimit DataPoints	Decimation	non-zero integer {2}
StateStorageClass	Code generation storage class	string — {'Auto'} | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer'
RTWStateStorageType Qualifier	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>'

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'}`
`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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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	
`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'`
`SaturateOnInteger Overflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`AllowDiffInputSizes`	Allow different data input sizes (Results in variable-size output signal)	string — `{'off'} | 'on'`

Block (Type)/Parameter	Dialog Box Prompt	Values
Manual Switch (`Manual Switch`) (masked subsystem)		
`varsize`	Allow different input sizes (Results in variable-size output signal)	string — `{'off'} | 'on'`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
Merge (`Merge`)		
`Inputs`	Number of inputs	string — `{'2'}`
`InitialOutput`	Initial output	string — `{'[]'}`
`AllowUnequalInput PortWidths`	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 — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `{'Inherit: Inherit via internal rule'} | 'Inherit: Inherit via`

Block (Type)/Parameter	Dialog Box Prompt	Values
		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'}
IndexMode	Index mode	string — 'Zero-based' | {'One-based'}
IndexOptionArray	Index Option	string — 'Select all' | {'Index vector (dialog)'} | 'Index vector (port)' |

Block (Type)/Parameter	Dialog Box Prompt	Values
		`'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'}`
`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'`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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'`
`SaturateOnInteger Overflow`	Saturate on integer overflow	string — `{'off'} | 'on'`
`AllowDiffInputSizes`	Allow different input sizes (Results in variable-size output signal)	string — `{'off'} | 'on'`
Variant Source (`VariantSource`)		
`VariantControls`	Variant control	string — `{'Variant'} | '(default)'`
`OverrideUsingVariant`	Override variant conditions and use the following variant	string — `{''}`
`AllowZeroVariantContro`	Allow zero active variant controls	string — `{'off'} | 'on'`
`ShowConditionOnBlock`	Show variant condition on block	string — `{'off'} | 'on'`
`GeneratePreprocessorCo`	Analyze all choices during update diagram and generate preprocessor conditionals	string — `{'off'} | 'on'`
Variant Sink (`VariantSink`)		
`VariantControls`	Variant control	string — `{'Variant'} | '(default)'`
`OverrideUsingVariant`	Override variant conditions and use the following variant	string — `{''}`

Block (Type)/Parameter	Dialog Box Prompt	Values
AllowZeroVariantContro	Allow zero active variant controls	string — {'off'} | 'on'
ShowConditionOnBlock	Show variant condition on block	string — {'off'} | 'on'
GeneratePreprocessorCo	Analyze all choices during update diagram and generate preprocessor conditionals	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'

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}
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'

Block (Type)/Parameter	Dialog Box Prompt	Values
`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 — `{'[]'}`
Scope (`Scope`)		
`Floating`		string — `{'off'} | 'on'`
Location		vector — `{'[188 390 512 629]'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'}
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
MatrixName	Variable name	string — {'ans'}
SaveFormat	Save format	string — {'Timeseries'} | 'Array'
Decimation	Decimation	string — {'1'}
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]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
VectorParams1D	Interpret vector parameters as 1-D	string — 'off' | {'on'}
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'}
SampleTime	Sampling time	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>'

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'}`
`ExtrapolationBeforeFirst`	Data extrapolation before first data point	string — `{'Linear extrapolation'} | 'Hold first value' | 'Ground value'`
`InterpolationWithinTimeR`	Data interpolation within time range	string — `{'Linear interpolation'} | 'Zero order hold'`

Block (Type)/Parameter	Dialog Box Prompt	Values
ExtrapolationAfterLastDa	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'}
IconDisplay	Icon display	string — 'Signal name' | {'Port number'} | 'Port number and signal name'
BusOutputAsStruct	Output as nonvirtual bus	string — {'off'} | 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'`
`Unit`	Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.	string — `{‘inherit’}| '<Enter unit>'`
`SignalType`	Signal type	string — `{'auto'} | 'real' | 'complex'`
`SamplingMode`	Sampling mode	string — `{'auto'} | 'Sample based' | 'Frame based'`
`LatchByDelaying OutsideSignal`	Latch input by delaying outside signal	string — `{'off'} | 'on'`
`LatchInputFor FeedbackSignals`	Latch input for feedback signals of function-call subsystem outputs	string — `{'off'} | 'on'`

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'}`
`Variance`	Variance	string — `{'1'}`
`Seed`	Seed	string — `{'0'}`
`SampleTime`	Sample time	string — `{'0.1'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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)'`
`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].''}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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'
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'}

Block (Type)/Parameter	Dialog Box Prompt	Values
`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'}`
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'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
`Maximum`	Maximum	string — `{'1'}`
`Seed`	Seed	string — `{'0'}`
`SampleTime`	Sample time	string — `{'0.1'}`
`VectorParams1D`	Interpret vector parameters as 1-D	string — `'off' | {'on'}`
Waveform Generator (`WaveformGenerator`)		
`OutMin`	Output minimum	string — `{'[]'}`
`OutMax`	Output maximum	string — `{'[]'}`
`OutDataTypeStr`	Output data type	string — `'Inherit: Inherit via back propagation' | {'Inherit: Inherit from table data'} | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'`
`LockScale`	Lock output data type setting against changes by the fixed-point tools	string — `{'off'} | 'on'`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | 'Floor' | {'Nearest'} | 'Round' | 'Simplest' | 'Zero'`
`SaturateOnIntegerOverflo`	Saturate on integer overflow	string — `{'off'} | 'on'`
`SelectedSignal`	Output signal	string — `{'1'}`
`SampleTime`	Sample time	string — `{'0'}`

User-Defined Functions Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
`MATLAB Function` (`Stateflow`) (masked subsystem)		
`MATLAB System` (`MATLABSystem`)		
`System`	System object class name	string — `{''}`
`Fcn` (`Fcn`)		
`Expr`	Expression	string — `{'sin(u(1)*exp(2.3*(-u(2))))'}`
`SampleTime`	Sample time (-1 for inherited)	string — `{'-1'}`
`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]'}`
`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'}`
`RndMeth`	Integer rounding mode	string — `'Ceiling' | 'Convergent' | {'Floor'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
		`| '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)		
`tsamp`	Sample time	string — `{'-1'}`
Unit Delay Resettable (`Unit Delay Resettable`) (masked subsystem)		
`vinit`	Initial condition	string — `{'0.0'}`
`tsamp`	Sample time	string — `{'-1'}`

Block (Type)/Parameter	Dialog Box Prompt	Values
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)		

Block (Type)/Parameter	Dialog Box Prompt	Values
Increment Real World (`Real World Value Increment`) (masked subsystem)		
Increment Stored Integer (`Stored Integer Value Increment`) (masked subsystem)		

Mask Parameters

About Mask Parameters

This section lists parameters that describe masked blocks. You can use these descriptive parameters with `get_param` and `set_param` to obtain and specify the properties of a block mask.

The descriptive mask parameters listed in this section apply to all masks, and provide access to all mask properties. Be careful not to confuse these descriptive mask parameters with the mask-specific parameters defined for an individual mask in the Mask Editor **Parameters** pane.

See "Masking Fundamentals" and "Mask Editor Overview" for information about block masks and the Mask Editor.

Mask Parameters

Parameter	Description/Prompt	Values
`Mask`	Turns mask on or off.	`{'on'} | 'off'`
`MaskCallbackString`	Mask parameter callbacks that are executed when the respective parameter is changed on the dialog. Set by the **Dialog callback** field on the **Parameters** pane of the Mask Editor dialog box. For more information, see "Mask Code Execution".	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 `{''}`
`MaskDisplay`	Drawing commands for the block icon. Set by the **Icon**	string `{''}`

Parameter	Description/Prompt	Values
	Drawing commands field on the **Icon & Ports** pane of the Mask Editor dialog box.	
`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. Set by the **Icon Transparency** option on the **Icon & Ports** pane of the Mask Editor dialog box.	`{'opaque'} | 'transparent' | 'opaque-with-ports'`
`MaskIconRotate`	Set the rotation of the icon (Rotates is `on`, Fixed is `off`). Set by the **Icon Rotation** option on the **Icon & Ports** pane of the Mask Editor dialog box.	`'on' | {'off'}`
`MaskIconUnits`	Set the units for the drawing commands. Set by the **Icon**	`'pixel' | {'autoscale'} | 'normalized'`

Parameter	Description/Prompt	Values
	Units option on the **Icon & Ports** pane of the Mask Editor dialog box.	
`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'`
`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 `{''}`
`MaskPropertyName String`	Pipe-delimited version of `MaskNames`.	string `{''}`
`MaskRunInitForIconRedraw`	For internal use only.	

Parameter	Description/Prompt	Values
`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.	
`MaskToolTipsDisplay`	Determines which mask dialog parameters to display in the tooltip for this masked block. Specify as a cell array of `'on'` or `'off'` values, each of which indicates whether to display the parameter named at the corresponding position in the cell array returned by `MaskNames`.	cell array of `'on'` and `'off'` {''}
`MaskToolTipString`	Comma-delimited version of `MaskToolTipsDisplay`.	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 `{'[]'}`

Parameter	Description/Prompt	Values
`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 `{''}`
`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

Fixed-Point Tool

Fixed-Point Tool Parameters and Dialog Box

The Fixed-Point Tool includes the following components:

- **Main** toolbar
- **Model Hierarchy** pane
- **Contents** pane
- **Dialog** pane

Main Toolbar

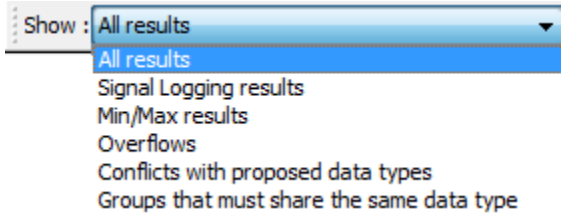
The Fixed-Point Tool's main toolbar appears near the top of the Fixed-Point Tool window under the Fixed-Point Tool's menu.

The toolbar contains the following buttons that execute commonly used Fixed-Point Tool commands:

Button	Usage
	Open the Fixed-Point Advisor to prepare the model for conversion to fixed point.
	Simulate a model and store the run results.
	Pause a simulation.
	Stop a simulation.
	Analyze model and store derived minimum and maximum results.
	Propose data types. Propose fraction lengths for specified word lengths or propose word lengths for specified fraction lengths.
	Apply accepted data types.
	Compare selected runs.
	Create a difference plot for the selected signals.
	Plot the selected signal.

Button	Usage
	Create a histogram plot for the selected signal.

The toolbar also contains the **Show** option:



The **Show** option specifies the type of results to display in the **Contents** pane. The **Contents** pane displays information only after you simulate a system or propose fraction lengths. If there are no results that satisfy a particular filter option, the list will be blank.

Show Option	Result
All results	Displays all results for the selected tree node.
Signal Logging results	For the selected tree node, displays blocks whose output ports have logged signal data. The Fixed-Point tool marks these blocks with the logged signal icon  . **Note:** You can plot simulation results associated with logged signal data using the Simulation Data Inspector.
Min/Max results	For the selected tree node, displays blocks that record design Min/Max, simulation Min/Max, and overflow data. **Prerequisites**: **Fixed-point instrumentation mode** should not be set to `Force Off`.
Overflows	For the selected tree node, displays blocks that have non-zero overflows recorded. If a block has its **Saturate on integer overflow** option selected, overflow information appears in the **Saturations** column, otherwise it appears in the **OverflowWraps** column.

Show Option	Result
Conflicts with proposed data types	For the selected tree node, displays results that have potential data typing or scaling issues. **Prerequisites**: This information is available only after you propose data types. The Fixed-Point Tool marks these results with a yellow or red icon, as shown here:  The proposed data type poses potential issues for this object. Open the Result Details tab to review these issues.  The proposed data type will introduce errors if applied to this object. Open the Result Details tab for details about how to resolve these issues.
Groups that must share the same data type	For the selected tree node, displays blocks that must share the same data type because of data type propagation rules. **Prerequisites**: This information is available only after you propose fraction lengths. The Fixed-Point Tool allocates an identification tag to blocks that must share the same data type. This identification tag is displayed in the **DTGroup** column as follows: • If the selected tree node is the model root All results for the model are listed. The **DTGroup** column is sorted by default so that you can easily view all blocks in a group. • If the selected tree node is a subsystem The identification tags have a suffix that indicates the total number of results in each group. For example, `G2(2)` means group `G2` has 2 members. This information enables you to see how many members of a group belong to the selected subsystem and which groups share data types across subsystem boundaries.

Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model hierarchy. The first node in the pane represents a Simulink model. Expanding the root node displays subnodes that represent the model's subsystems, `MATLAB Function` blocks, Stateflow charts, and referenced models.

The Fixed-Point Tool's **Contents** pane displays elements that comprise the object selected in the **Model Hierarchy** pane. The **Dialog** pane provides parameters for specifying the selected object's data type override and fixed-point instrumentation mode. You can also specify an object's data type override and fixed-point instrumentation mode by right-clicking on the object. The **Model Hierarchy** pane indicates the value of these parameters by displaying the following abbreviations next to the object name:

Abbreviation	Parameter Value
Fixed-point instrumentation mode	
`mmo`	`Minimums, maximums and overflows`
`o`	`Overflows only`
`fo`	`Force off`
Data type override	
`scl`	`Scaled double`
`dbl`	`Double`
`sgl`	`Single`
`off`	`Off`

Contents Pane

The **Contents** pane displays a tabular view of objects that log fixed-point data in the system or subsystem selected in the **Model Hierarchy** pane. The table rows correspond to model objects, such as blocks, block parameters, and Stateflow data. The table columns correspond to attributes of those objects, such as the data type, design minimum and maximum values, and simulation minimum and maximum values.

The **Contents** pane displays information only after you simulate a system, analyze the model to derive minimum and maximum values, or propose fraction lengths.

You can control which of the following columns the Fixed-Point Tool displays in this pane. For more information, see “Customizing the Contents Pane View” on page 7-8.

Column Label	Description
Accept	Check box that enables you to selectively accept the Fixed-Point Tool's data type proposal.
CompiledDesignMax	Compile-time information for **DesignMax**.
CompiledDesignMin	Compile-time information for **DesignMin**.
CompiledDT	Compile-time data type. This data type appears on the signal line in `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.
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.
InitValueMin	Minimum initial value for a signal or parameter. Some model objects provide parameters that allow you to

Column Label	Description
	specify the initial values of their signals. For example, the `Constant` block includes a **Constant value** that initializes the block output signal. **Note:** The Fixed-Point Tool uses this parameter when it proposes data types.
LogSignal	Check box that allows you to enable or disable signal logging for an object.
ModelRequiredMin	Minimum value of a parameter used during simulation. For example, the `n-D Lookup Table` block uses the **Breakpoints** and **Table data** parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets **ModelRequiredMin** to the minimum of the minimum values of all these parameters. **Note:** The Fixed-Point Tool uses this parameter when it proposes data types.
ModelRequiredMax	Maximum value of a parameter used during simulation. For example, the `n-D Lookup Table` block uses the **Breakpoints** and **Table data** parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets **ModelRequiredMax** to the maximum of the maximum values of all these parameters. **Note:** The Fixed-Point Tool uses this parameter when it proposes data types.
Name	Identifies path and name of block.
OverflowWraps	Number of overflows that wrap during simulation.
ProposedDT	Data type that the Fixed-Point Tool proposes.
ProposedMax	Maximum value that results from the data type the Fixed-Point Tool proposes.

Column Label	Description
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".
SimMax	Maximum value that occurs during simulation.
SimMin	Minimum value that occurs during simulation.
SpecifiedDT	Data type the block specifies in its parameter dialog box, for example, the value of its **Output data type** parameter.

Customizing the Contents Pane View

You can customize the **Contents** pane in the following ways:

- "Using Column Views" on page 7-8
- "Changing Column Order and Width" on page 7-10
- "Sorting by Columns" on page 7-10

Using Column Views

The Fixed-Point Tool provides the following standard Column Views:

View Name	Columns Provided	When Does the Fixed-Point Tool Display this View?
`Simulation View` (default)	**Name**, **Run**, **CompiledDT**, **SpecifiedDT**, **SimMin**, **SimMax**, **DesignMin**, **DesignMax**, **OverflowWraps**, **Saturations**	After a simulating minimum and maximum values.
`Automatic Data Typing View`	**Name**, **Run**, **CompiledDT**, **CompiledDesignMax**, **CompiledDesignMin**,**Accept**,	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?
	ProposedDT, **SpecifiedDT**, **DesignMin**, **DesignMax**, **DerivedMin**, **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.
`Data Collection View`	**Name**, **Run**, **CompiledDT**, **SpecifiedDT**, **DerivedMin**, **DerivedMax**, **SimMin**, **SimMax**, **OverflowWraps**, **Saturations**	After simulating or deriving minimum and maximum values if the results have simulation min/max, derived min/max, and design min/max.
`Derived Min/Max View`	**Name**, **Run**, **CompiledDesignMax**, **CompiledDesignMin**, **DerivedMin**, **DerivedMax**	After deriving minimum and maximum values.

By selecting **Show Details**, you can:

- Customize the standard column views
- Create your own column views
- Export and import column views saved in MAT-files, which you can share with other users

- Reset views to factory settings

  If you upgrade to a new release of Simulink, and the column views available in the Fixed-Point Tool do not match the views described in the documentation, reset your views to factory settings. When you reset all views, the Model Explorer removes all the custom views you have created. Before you reset views to factory settings, export any views that you will want to use in the future.

You can prevent the Fixed-Point Tool from automatically changing the column view of the contents pane by selecting **View > Lock Column View** in the Fixed-Point Tool menu. For more information on controlling views, see "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.

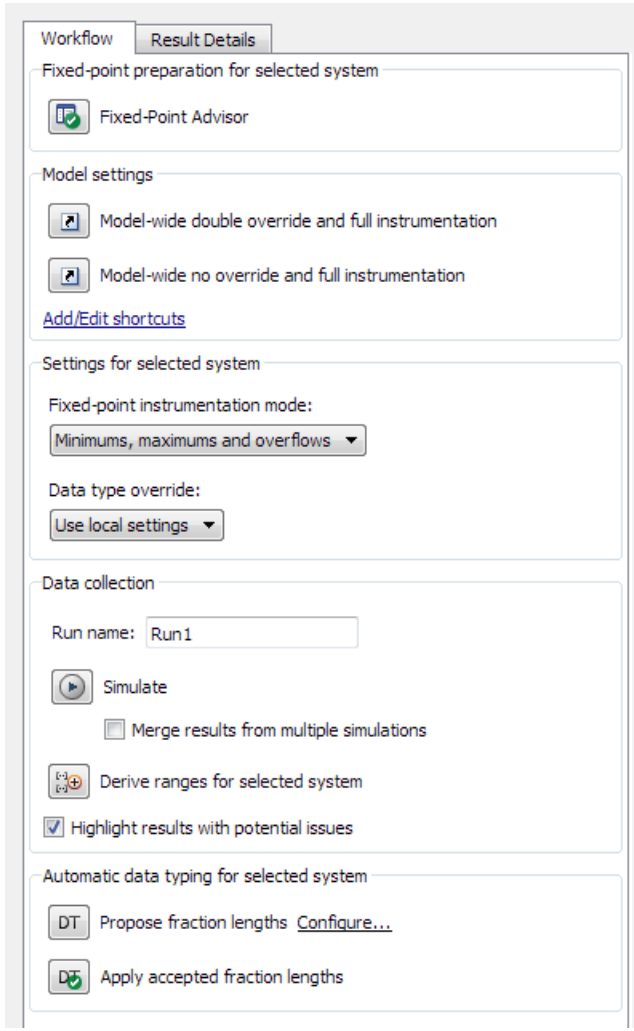
Sorting by Columns

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 system under design.



The Dialog pane includes the following components:

Component	Description
System under design	Displays the system under design for conversion. You can change the selected system by clicking **Change**.

Component	Description
Fixed-point preparation	Contains the **Fixed-Point Advisor** button. Use this button to open the Fixed-Point Advisor to guide you through the tasks to prepare your floating-point model for conversion to fixed point. For more information, see "Fixed-Point Advisor" on page 7-13.
Configure model settings	Contains default configurations that set up run parameters, such as the run name and data type override settings, by clicking a button. For more information, see "Configure model settings" on page 7-14.
Range collection	Contains controls to collect simulation or derived minimum and maximum data for your model.
Automatic data typing	Contains controls to propose and, optionally, accept data type proposals.
Result Details tab	Use this tab to view data type information about the object selected in the **Contents** pane.

Tips

From the Fixed-Point Tool **View** menu, you can customize the layout of the **Dialog** pane. Select:

- **Show Fixed-Point Preparation** to show/hide the **Fixed-Point Advisor** button. By default, the Fixed-Point Tool displays this button.
- **Show Dialog View** to show/hide the **Dialog** pane. By default, the Fixed-Point Tool displays this pane.
- **Settings for selected system** to show/hide the **Settings for selected system** pane. By default, the Fixed-Point Tool displays this pane.

Fixed-Point Advisor

Open the Fixed-Point Advisor to guide you through the tasks to prepare a floating-point model for conversion to fixed point. Use the Fixed-Point Advisor if your model contains blocks that do not support fixed-point data types.

Configure model settings

Use the configurations to set up model-wide data type override and instrumentation settings prior to simulation. The Fixed-Point Tool provides:

- Frequently-used factory default configurations
- The ability to add and edit custom configurations

Note: The factory default configurations apply to the whole model. You cannot use these shortcuts to configure subsystems.

Factory Defaults

Factory Default Configuration	Description
Range collection using double override	Use this configuration to observe ideal numeric behavior of the model and collect ranges for data type proposals. This configuration sets: • **Run name** to `DoubleOverride` • **Fixed-point instrumentation mode** to `Minimums, maximums and overflows` • **Data type override** to `Double` • **Data type override applies to** to `All numeric types` By default, a button for this configuration appears in the **Configure model settings** pane.
Range collection with specified data types	Use this configuration to collect ranges of actual model and to validate current behavior. This configuration sets: • **Run name** to `NoOverride` • **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`

Factory Default Configuration	Description
	• **Data type override** to `Use local settings` By default, a button for this shortcut appears in the **Configure model settings** pane.
Remove overrides and disable range collection	Use this configuration to cleanup settings after finishing fixed-point conversion and to restore maximum simulation speed. This configuration sets: • **Fixed-point instrumentation mode** to `Off` • **Data type override** to `Use local settings` By default, a button for this shortcut appears in the **Configure model settings** pane.

Advanced settings

Use **Advanced settings** to add new configurations or edit existing user-defined configurations.

Run name

Specifies the run name

If you use a default configuration to set up a run, the Fixed-Point Tool uses the run name associated with this configuration. You can override the run name by entering a new name in this field.

Tips

- To store data for multiple runs, provide a different run name for each run. Running two simulations with the same run name overwrites the original run unless you select **Merge results from multiple simulations**.
- You can edit the run name in the Contents pane **Run** column.

For more information, see “Run Management”.

Simulate

Simulates model and stores results.

Action

Simulates the model and stores the results with the run name specified in **Run name**. The Fixed-Point Tool displays the run name in the **Run** column of the **Contents** pane.

Merge instrumentation results from multiple simulations

Control how simulation results are stored

Settings

Default: Off

 On

Merges new simulation minimum and maximum results with existing simulation results in the run specified by the run name parameter. Allows you to collect complete range information from multiple test benches. Does not merge signal logging results.

 Off

Clears all existing simulation results from the run specified by the run name parameter before displaying new simulation results.

Command-Line Alternative

Parameter: `'MinMaxOverflowArchiveMode'`
Type: string
Value: `'Overwrite' | 'Merge'`
Default: `'Overwrite'`

Tip

Select this parameter to log simulation minimum and maximum values captured over multiple simulations. For more information, see “Propose Data Types Using Multiple Simulations”.

Derive ranges for selected system

Derive minimum and maximum values for signals for the selected system.

The Fixed-Point Tool analyzes the selected system to compute derived minimum and maximum values based on design minimum and maximum values specified on blocks. For example, using the **Output minimum** and **Output maximum** for block outputs.

Action

Analyzes the selected system to compute derived minimum and maximum information based on the design minimum and maximum values specified on blocks.

By default, the Fixed-Point Tool displays the `Derived Min/Max View` with the following information in the **Contents** pane.

Command-Line Alternative

No command line alternative available.

Dependencies

Range analysis:

- Requires a Fixed-Point Designer license.

Propose

Signedness

Select whether you want The Fixed-Point Tool to propose signedness for results in your model. The Fixed-Point Tool proposes signedness based on collected range data and block constraints. By default, the **Signedness** check box is selected.

When the check box is selected, signals that are always strictly positive get an unsigned data type proposal. If you clear the check box, the Fixed-Point Tool proposes a signed data type for all results that currently specify a floating-point or an inherited output data type unless other constraints are present. If a result specifies a fixed-point output data type, the Fixed-Point Tool will propose a data type with the same signedness as the currently specified data type unless other constraints are present.

Word length or fraction length

Select whether you want the Fixed-Point Tool to propose word lengths or fraction lengths for the objects in your system.

- If you select **Word length**, the Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length to avoid overflows.
- If you select **Fraction length**, the Fixed-Point Tool proposes a data type with the specified word length and best-precision fraction length while avoiding overflows.

If a result currently specifies a fixed-point data type, that information will be used in the proposal. If a result specifies a floating-point or inherited output data type, and the **Inherited** and **Floating point** check boxes are selected, the Fixed-Point Tool uses the settings specified under **Automatic data typing** to make a data type proposal.

Propose for

Inherited

Propose data types for results that specify one of the inherited output data types.

Floating-point

Propose data types for results that specify floating-point output data types.

Default fraction length

Specify the default fraction length for objects in your model. The Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length that avoids overflows.

Command-Line Alternative

No command line alternative available.

Default word length

Specify the default word length for objects in your model. The Fixed-Point Tool will propose best-precision fraction lengths based on the specified default word length.

Command-Line Alternative

No command line alternative available.

When proposing types use

Specify the types of ranges to use for data type proposals.

Design and derived ranges

The Fixed-Point Tool uses the design ranges in conjunction with derived ranges to propose data types. Design ranges take precedence over derived ranges.

Design and simulation ranges

The Fixed-Point Tool uses the design ranges in conjunction with collected simulation ranges to propose data types. Design ranges take precedence over simulation ranges.

The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. For more information, see “Safety margin for simulation min/max (%)” on page 7-25

All collected ranges

The Fixed-Point Tool uses design ranges in addition to derived and simulation ranges to propose data types.

Design minimum and maximum values take precedence over simulation and derived ranges.

Command-Line Alternative

No command line alternative available.

Safety margin for simulation min/max (%)

Specify safety factor for simulation minimum and maximum values.

Settings

Default: 0

The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. The specified safety margin must be a real number greater than `-100`. For example, a value of 55 specifies that a range *at least* 55 percent larger is desired. A value of -15 specifies that a range *up to* 15 percent smaller is acceptable.

Dependencies

Before performing automatic data typing, you must specify design minimum and maximum values or run a simulation to collect simulation minimum and maximum data, or collect derived minimum and maximum values.

Command-Line Alternative

No command line alternative available.

Advanced Settings

In this section...

Advanced Settings Overview

Use the Advanced Settings dialog to control the fixed-point instrumentation mode, and data type override settings. You can also use the Advanced Settings dialog to add or edit user-defined configurations. You cannot modify the factory default configurations. If you add a new configuration and want it to appear as a button on the Fixed-Point Tool **Configure model settings** pane, use the controls in the **Shortcuts** tab.

Fixed-point instrumentation mode

Control which objects log minimum, maximum and overflow data during simulation.

Settings

Default: `Use local settings`

`Use local settings`

Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

`Minimums, maximums and overflows`

Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

`Overflows only`

Logs only overflow data for all blocks in the current system or subsystem.

`Force off`

Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

Tips

- You cannot change the instrumentation mode for linked subsystems or referenced models.

Dependencies

The value of this parameter for parent systems controls min/max logging for all child subsystems, unless `Use local settings` is selected.

Command-Line Alternative

Parameter: `'MinMaxOverflowLogging'`
Type: string
Value: `'UseLocalSettings' | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'`
Default: `'UseLocalSettings'`

Data type override

Control data type override of objects that allow you to specify data types in their dialog boxes.

Settings

Default: `Use local settings`

The value of this parameter for parent systems controls data type override for all child subsystems, unless `Use local settings` is selected.

`Use local settings`

Overrides data types according to the setting of this parameter for each subsystem.

`Scaled double`

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

`Double`

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

`Single`

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

`Off`

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

Tips

- Set this parameter to `Double` or `Single` and the **Data type override applies to** parameter to `All numeric types` to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.
- You cannot change the **Data type override** setting on linked subsystems or referenced models.
- Data type override never applies to `boolean` data types.
- When you set the **Data type override** parameter of a parent system to `Double`, `Single`, `Scaled double` or `Off`, this setting also applies to all child subsystems and

you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is `Use local settings`, you can set the **Data type override** parameter for individual children.

- Use this parameter with the **Data type override applies to** parameter. The following table details how these two parameters affect the data types in your model.

Fixed-Point Tool Settings		Block Local Settings	
Data type override	**Data type override applies to**	**Floating-point types**	**Fixed-point types**
Use local settings/Off	N/A	Unchanged	Unchanged
Double	All numeric types	Double	Double
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Double
Single	All numeric types	Single	Single
	Floating-point	Single	Unchanged
	Fixed-point	Unchanged	Single
Scaled double	All numeric types	Double	Scaled double equivalent of fixed-point type
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Scaled double equivalent of fixed-point type

Dependencies

- The following Simulink blocks allow you to set data types in their block masks, but ignore the **Data type override** setting:
  - `Probe`
  - `Trigger`
  - `Width`

Command-Line Alternative

Parameter: `'DataTypeOverride'`
Type: string

Value: `'UseLocalSettings' | 'ScaledDouble' | 'Double' | 'Single' | 'Off'`
Default: `'UseLocalSettings'`

Data type override applies to

Specifies which data types the Fixed-Point Tool overrides

Settings

Default: `All numeric types`

`All numeric types`

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to `boolean` or enumerated data types.

`Floating-point`

Data type override applies only to floating-point data types, that is, `double` and `single`.

`Fixed-point`

Data type override applies only to fixed-point data types, for example, `uint8`, `fixdt`.

Tips

- Use this parameter with the **Data type override** parameter.
- Data type override never applies to `boolean` or enumerated data types or to buses.
- When you set the **Data type override** parameter of a parent system to `Double`, `Single`, `Scaled double` or `Off`, this setting also applies to all child subsystems and you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is `Use local setting`, you can set the **Data type override** parameter for individual children.
- The following table details how these two parameters affect the data types in your model.

Fixed-Point Tool Settings		Block Local Settings	
Data type override	**Data type override applies to**	**Floating-point types**	**Fixed-point types**
Use local settings/Off	N/A	Unchanged	Unchanged
Double	All numeric types	Double	Double
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Double

Fixed-Point Tool Settings		Block Local Settings	
Data type override	**Data type override applies to**	**Floating-point types**	**Fixed-point types**
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'`

Name of shortcut

Enter a unique name for your shortcut. By default, the Fixed-Point Tool uses this name as the **Run name** for this shortcut.

If the shortcut name already exists, the new settings overwrite the existing settings.

See Also

- "Run Management"

Allow modification of fixed-point instrumentation settings

Select whether to change the model fixed-point instrumentation settings when you apply this shortcut to the model.

Settings

Default: On

 On

When you apply this shortcut to the model, changes the fixed-point instrumentation settings of the model and its subsystems to the setting defined in this shortcut.

 Off

Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.

Tip

If you want to control data type override settings without altering the fixed-point instrumentation settings on your model, clear this option.

See Also

- “Run Management”

Allow modification of data type override settings

Select whether to change the model data type override settings when you apply this shortcut to the model

Settings

Default: On

 On

When you apply this shortcut to the model, changes the data type override settings of the model and its subsystems to the settings defined in this shortcut .

 Off

Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.

Allow modification of run name

Select whether to change the run name on the model when you apply this shortcut to the model

Settings

Default: On

 On

Changes the run name to the setting defined in this shortcut when you apply this shortcut to the model.

 Off

Does not change the run name when you apply this shortcut to the model.

Run name

Specify the run name to use when you apply this shortcut.

By default, the run name uses the name of the shortcut. Run names are case sensitive.

Dependency

Allow modification of run name enables this parameter.

Capture system settings

Copy the model and subsystem fixed-point instrumentation mode and data type override settings into the Shortcut editor.

Fixed-point instrumentation mode

Control which objects in the shortcut editor log minimum, maximum and overflow data during simulation.

This information is stored in the shortcut. To use the current model setting, click **Capture system settings**.

Settings

Default: Same as model setting

`Use local settings`

Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

`Minimums, maximums and overflows`

Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

`Overflows only`

Logs only overflow data for all blocks in the current system or subsystem.

`Force off`

Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

Dependency

Allow modification of fixed-point instrumentation settings enables this parameter.

Data type override

Control data type override of objects that allow you to specify data types in their dialog boxes.

This information is stored in the shortcut. To use the current model settings, click **Capture system settings**.

Settings

Default: Same as model

The value of this parameter for parent systems controls data type override for all child subsystems, unless `Use local settings` is selected.

`Use local settings`

Overrides data types according to the setting of this parameter for each subsystem.

`Scaled double`

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

`Double`

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

`Single`

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

`Off`

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

Dependency

Allow modification of data type override settings enables this parameter.

Data type override applies to

Specifies which data types to override when you apply this shortcut.

This information is stored in the shortcut. To use the current model setting, click **Capture system settings**.

Settings

Default: `All numeric types`

`All numeric types`

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to `boolean` or enumerated data types.

`Floating-point`

Data type override applies only to floating-point data types, that is, `double` and `single`.

`Fixed-point`

Data type override applies only to fixed-point data types, for example, `uint8`, `fixdt`.

Dependency

Allow modification of data type override settings enables this parameter.

8

Model Advisor Checks

Simulink Checks

In this section...
“Simulink Check Overview” on page 8-5
“Migrating to Simplified Initialization Mode Overview” on page 8-5
“Identify unconnected lines, input ports, and output ports” on page 8-7
“Check root model Inport block specifications” on page 8-8
“Check optimization settings” on page 8-9
“Check diagnostic settings ignored during accelerated model reference simulation” on page 8-12
“Check for parameter tunability information ignored for referenced models” on page 8-13
“Check for implicit signal resolution” on page 8-14
“Check for optimal bus virtuality” on page 8-15
“Check for Discrete-Time Integrator blocks with initial condition uncertainty” on page 8-16
“Identify disabled library links” on page 8-17
“Identify parameterized library links” on page 8-18
“Identify unresolved library links” on page 8-19
“Identify model reference variants and variant subsystems that override variant choice” on page 8-20
“Identify configurable subsystem blocks for converting to variant subsystem blocks” on page 8-21
“Check usage of function-call connections” on page 8-21
“Check model for upgradable Simulink Scope blocks” on page 8-22
“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues” on page 8-23
“Check if read/write diagnostics are enabled for data store blocks” on page 8-25
“Check data store block sample times for modeling errors” on page 8-27
“Check for potential ordering issues involving data store access” on page 8-28
“Check structure parameter usage with bus signals” on page 8-30

In this section...

In this section...

Simulink Check Overview

Use the Simulink Model Advisor checks to configure your model for simulation.

See Also

- "Run Model Checks"
- "Simulink Coder Checks"
- "Simulink Verification and Validation Checks"

Migrating to Simplified Initialization Mode Overview

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. This mode is especially important for models that do not specify initial conditions for conditionally executed subsystem output ports. See "Address Classic Mode Issues by Using Simplified Mode" for more information.

Use the Model Advisor checks in **Migrating to Simplified Initialization Mode** to help migrate your model to simplified initialization mode.

See Also

- "Address Classic Mode Issues by Using Simplified Mode"
- "Underspecified initialization detection"
- "Check usage of Merge blocks" on page 8-41
- "Check usage of Outport blocks" on page 8-44
- "Check usage of Discrete-Time Integrator blocks" on page 8-55
- "Check model settings for migration to simplified initialization mode" on page 8-57

Identify unconnected lines, input ports, and output ports

Check ID: `mathworks.design.UnconnectedLinesPorts`

Check for unconnected lines or ports.

Description

This check lists unconnected lines or ports. These can have difficulty propagating signal attributes such as data type, sample time, and dimensions.

Note: Ports connected to ground/terminator blocks will pass this test.

Results and Recommended Actions

Condition	Recommended Action
Lines, input ports, or output ports are unconnected.	Connect the signals. Double-click the list of unconnected items to locate failure.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

Tips

Use the `PortConnectivity` command to obtain an array of structures describing block input or output ports.

See Also

"Common Block Properties" on page 6-86 for information on the `PortConnectivity` command.

"What Is a Model Advisor Exclusion?"

Check root model Inport block specifications

Check ID: `mathworks.design.RootInportSpec`

Check that root model Inport blocks fully define dimensions, sample time, and data type.

Description

Using root model Inport blocks that do not fully define dimensions, sample time, or data type can lead to undesired simulation results. Simulink software back-propagates dimensions, sample times and data types from downstream blocks unless you explicitly assign them values.

Results and Recommended Actions

Condition	Recommended Action
Root-level Inport blocks have undefined attributes.	Fully define the attributes of the root-level Inport blocks.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- “About Data Types in Simulink”.
- “Determine Output Signal Dimensions”.
- “Specify Sample Time”.
- “What Is a Model Advisor Exclusion?”

Check optimization settings

Check ID: `mathworks.design.OptimizationSettings`

Check for optimizations that can lead to non-optimal code generation and simulation.

Description

This check reviews the status of optimizations that can improve code efficiency and simulation time.

Results and Recommended Actions

Condition	Recommended Action
The specified optimizations are off.	Select the following optimization check boxes on the **Optimization** pane in the Configuration Parameters dialog box: • **"Remove root level I/O zero initialization"** • **"Remove internal data zero initialization"** Select the following optimization check boxes on the **Optimization > Signals and Parameters** pane in the Configuration Parameters dialog box: • **"Inline invariant signals"** (only if you have a Simulink Coder license) Select the following optimization check boxes on the **All Parameters** tab in the Configuration Parameters dialog box: • **"Block reduction"** • **"Conditional input branch execution"** • **"Implement logic signals as Boolean data (vs. double)"** • **"Use memset to initialize floats and doubles to 0.0"** • **"Remove code from floating-point to integer conversions that wraps out-of-range values"** (only if you have a Simulink Coder license) • **"Signal storage reuse"** • **"Enable local block outputs"** • **"Reuse local block outputs"**

Condition	Recommended Action
	• **"Eliminate superfluous local variables (Expression folding)"** 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: • **All Parameters > Solver data inconsistency** • **All Parameters > Array bounds exceeded** • **Diagnostics > Data Validity > Simulation range checking**
The specified Embedded Coder parameters are off.	If you have an Embedded Coder license and you are using an ERT-based system target file: • Select **All Parameters > Single output/update function**. For details, see "Single output/update function". • Select **All Parameters > Ignore test point signals**. For details, see "Ignore test point signals". • Set **Optimization > Signals and Parameters > Pass reusable subsystem outputs as** to `Individual argutments`. For details, see "Pass reusable subsystem outputs as".

Tips

If the system contains `Model` blocks and the referenced model is in Accelerator mode, simulating the model requires generating and compiling code.

See Also

- "Optimization Pane: General".

Check diagnostic settings ignored during accelerated model reference simulation

Check ID: `mathworks.design.ModelRefSIMConfigCompliance`

Checks for referenced models for which Simulink changes configuration parameter settings during accelerated simulation.

Description

For models referenced in Accelerator mode, Simulink ignores the settings of the following configuration parameters that you set to a value other than `None`.

- **All Parameters > Array bounds exceeded**
- **Diagnostics > Data Validity > Inf or NaN block output**
- **Diagnostics > Data Validity > Simulation range checking**
- **Diagnostics > Data Validity > Division by singular matrix**
- **Diagnostics > Data Validity > Wrap on overflow**

Also, for models referenced in Accelerator mode, Simulink ignores the following **Configuration Parameters > Diagnostics > Data Validity > Data Store Memory Block** parameters if you set them to a value other than `Disable all`. For details, see "Data Store Diagnostics".

- **Detect read before write**
- **Detect write after read**
- **Detect write after write**

Results and Recommended Actions

Condition	Recommended Action
You want to see the results of running the identified diagnostics with settings to produce warnings or errors.	Simulate the model in Normal mode and resolve diagnostic warnings or errors.

Check for parameter tunability information ignored for referenced models

Check ID: `mathworks.design.ParamTunabilityIgnored`

Checks if parameter tunability information is included in the Model Parameter Configuration dialog box.

Description

Simulink software ignores tunability information specified in the Model Parameter Configuration dialog box. This check identifies those models containing parameter tunability information that Simulink software will ignore if the model is referenced by other models.

Results and Recommended Actions

Condition	Recommended Action
Model contains ignored parameter tunability information.	Click the links to convert to equivalent Simulink parameter objects in the MATLAB workspace.

See Also

"Parameter Storage in the Generated Code".

Check for implicit signal resolution

Check ID: `mathworks.design.ImplicitSignalResolution`

Identify models that attempt to resolve named signals and states to `Simulink.Signal` objects.

Description

Requiring Simulink software to resolve all named signals and states is inefficient and slows incremental code generation and model reference. This check identifies those signals and states for which you may turn off implicit signal resolution and enforce resolution.

Results and Recommended Actions

Condition	Recommended Action
Not all signals and states are resolved.	Turn off implicit signal resolution and enforce resolution for each signal and state that does resolve.

See Also

"Resolve Signal Objects for Output Data".

Check for optimal bus virtuality

Check ID: `mathworks.design.OptBusVirtuality`

Identify virtual buses that could be made nonvirtual. Making these buses nonvirtual improves generated code efficiency.

Description

This check identifies blocks incorporating virtual buses that cross a subsystem boundary. Changing these to nonvirtual improves generated code efficiency.

Results and Recommended Actions

Condition	Recommended Action
Blocks that specify a virtual bus crossing a subsystem boundary.	Change the highlighted bus to nonvirtual.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- “Signal Basics”.
- “Virtual and Nonvirtual Buses”.
- “What Is a Model Advisor Exclusion?”

Check for Discrete-Time Integrator blocks with initial condition uncertainty

Check ID: `mathworks.design.DiscreteTimeIntegratorInitCondition`

Identify Discrete-Time Integrator blocks with state ports and initial condition ports that are fed by neither an Initial Condition nor a Constant block.

Description

Discrete-Time Integrator blocks with state port and initial condition ports might not be suitably initialized unless they are fed from an Initial Condition or Constant block. This is more likely to happen when Discrete-Time Integrator blocks are used to model second-order or higher-order dynamic systems.

Results and Recommended Actions

Condition	Recommended Action
Discrete-Time Integrator blocks are not initialized during the model initialization phase.	Add a Constant or Initial Condition block to feed the external Initial Condition port.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- IC block
- `Discrete-Time Integrator` block
- Constant block
- "What Is a Model Advisor Exclusion?"

Identify disabled library links

Check ID: `mathworks.design.DisabledLibLinks`

Search model for disabled library links.

Description

Disabled library links can cause unexpected simulation results. Resolve disabled links before saving a model.

Note: This check may overlap with "Check model for block upgrade issues" on page 8-72.

Results and Recommended Actions

Condition	Recommended Action
Library links are disabled.	Click the **Library Link > Resolve link** option in the context menu.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

Tips

- Use the Model Browser to find library links.
- To enable a broken link, right-click a block in your model to display the context menu. Select **Library Link > Resolve link**.

See Also

"Restore Disabled or Parameterized Links".

"What Is a Model Advisor Exclusion?"

Identify parameterized library links

Check ID: `mathworks.design.ParameterizedLibLinks`

Search model for parameterized library links.

Description

Parameterized library links that are unintentional can result in unexpected parameter settings in your model. This can result in improper model operation.

Results and Recommended Actions

Condition	Recommended Action
Parameterized links are listed.	Verify that the links are intended to be parameterized.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

Tips

- Right-click a block in your model to display the context menu. Choose `Link Options` and click `Go To Library Block` to see the original block from the library.
- To parameterize a library link, choose `Look Under Mask`, from the context menu and select the parameter.

See Also

"Restore Disabled or Parameterized Links".

"What Is a Model Advisor Exclusion?"

Identify unresolved library links

Check ID: `mathworks.design.UnresolvedLibLinks`

Search the model for unresolved library links, where the specified library block cannot be found.

Description

Check for unresolved library links. Models do not simulate while there are unresolved library links.

Results and Recommended Actions

Condition	Recommended Action
Library links are unresolved.	Locate missing library block or an alternative.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

"Fix Unresolved Library Links"

"What Is a Model Advisor Exclusion?"

Identify model reference variants and variant subsystems that override variant choice

Check ID: `mathworks.design.VariantOverride`

Identify model or subsystem for model reference variants and variant subsystems that specify variant choice using the override option instead of using the active variant object.

Results and Recommended Actions

Condition	Recommended Action
Model reference variants or variant subsystems that override variant choice are identified.	Specify variant choice using active variant object.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

"Activation States of Variant Choices"

"What Is a Model Advisor Exclusion?"

Identify configurable subsystem blocks for converting to variant subsystem blocks

Check ID: `mathworks.design.CSStoVSSConvert`

Search the model to identify configurable subsystem blocks at the model or subsystem level.

Results and Recommended Actions

Condition	Recommended Action
Configurable subsystem blocks are identified.	Convert these blocks to variant subsystem blocks to avoid compatibility issues. See `Configurable Subsystem`.

Capabilities and Limitations

You can run this check on your library models.

See Also

"Set up Model Variants"

Check usage of function-call connections

Check ID: `mathworks.design.CheckForProperFcnCallUsage`

Check model diagnostic settings that apply to function-call connectivity and that might impact model execution.

Description

Check for connectivity diagnostic settings that might lead to non-deterministic model execution.

Results and Recommended Actions

Condition	Recommended Action
Diagnostics > Connectivity > Invalid function-call connection is set to `none`	Set **Diagnostics > Connectivity > Invalid function-call connection** to `error`.

Condition	Recommended Action
or `warning`. This might lead to non-deterministic model execution.	
Diagnostic > Connectivity > Context-dependent inputs is set to `Disable All` or `Use local settings`. This might lead to non-deterministic model execution.	Set **Diagnostics > Connectivity > Context-dependent inputs** to `Enable all as errors`.

See Also

`Function-Call Subsystem`

Check model for upgradable Simulink Scope blocks

Check model for Simulink Scope blocks that you can upgrade to Simulink Time Scope blocks

Description

In a future release, Simulink Scope blocks will be removed and replaced with Simulink Time Scope blocks.

Results and Recommended Actions

Condition	Recommended Action
Model does not have Simulink Scope blocks.	No action required.
Model includes at least one Simulink Scope block.	Select the **Check model for upgradable Simulink Scope blocks** check, click the **Run This Check** button, review the list of scope blocks, and then click the **Upgrade** button.

Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues

Check ID: `mathworks.design.DataStoreMemoryBlkIssue`

Look for modeling issues related to `Data Store Memory` blocks.

Description

Checks for multitasking data integrity, strong typing, and shadowing of data stores of higher scope.

Results and Recommended Actions

Condition	Recommended Action
The **Duplicate data store names** check is set to `none` or `warning`.	Consider setting the "Duplicate data store names" check to `error` in the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane.
The data store variable names are not strongly typed in one of the following: • **Signal Attributes** pane of the **Block Parameters** dialog for the `Date Store Memory` block • Global data store name	Specify a data type other than auto by taking one of the following actions: • Choose a data type other than `Inherit: auto` on the **Signal Attributes** pane of the **Block Parameters** dialog for the `Date Store Memory` block. • If you are using a global data store name, then specify its data type in the Simulink.Signal object.
The **Multitask data store** check is set to `none` or `warning`.	Consider setting the "Multitask data store" check to `error` in the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- “Local and Global 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”
- “What Is a Model Advisor Exclusion?”

Check if read/write diagnostics are enabled for data store blocks

Check ID: `mathworks.design.DiagnosticDataStoreBlk`

For data store blocks in the model, enable the read-and-write diagnostics order checking to detect run-time issues.

Description

Check for the read-and-write diagnostics order checking. By enabling the read-and-write diagnostics, you detect potential run-time issues.

Results and Recommended Actions

Condition	Recommended Action
The **Detect read before write** check is disabled.	Consider enabling "Detect read before write" in the Configuration Parameter dialog box **Diagnostics> Data Validity** pane.
The **Detect write after read** check is disabled.	Consider enabling "Detect write after read" in the Configuration Parameter dialog box **Diagnostics> Data Validity** pane.
The **Detect write after write** check is disabled.	Consider enabling "Detect write after write" in the Configuration Parameter dialog box **Diagnostics> Data Validity** pane.

Capabilities and Limitations

Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

Tips

.

- The run-time diagnostics can slow simulations down considerably. Once you have verified that Simulink does not generate warnings or errors during simulation, set them to `Disable all`.

See Also

- "Local and Global Data Stores"
- `Data Store Memory`
- `Data Store Read`
- `Data Store Write`
- "Detect read before write"
- "Detect write after read"
- "Detect write after write"
- "Check for potential ordering issues involving data store access" on page 8-28
- "What Is a Model Advisor Exclusion?"

Check data store block sample times for modeling errors

Check ID: `mathworks.design.DataStoreBlkSampleTime`

Identify modeling errors due to the sample times of data store blocks.

Description

Check data store blocks for continuous or fixed-in-minor-step sample times.

Results and Recommended Actions

Condition	Recommended Action
Data store blocks in your model have continuous or fixed-in-minor-step sample times.	Consider making the listed blocks discrete or replacing them with either `Memory` or `Goto` and `From` blocks.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- "Local and Global Data Stores"
- `Data Store Memory`
- `Data Store Read`
- `Data Store Write`
- "Fixed-in-Minor-Step"
- "What Is a Model Advisor Exclusion?"

Check for potential ordering issues involving data store access

Check ID: `mathworks.design.OrderingDataStoreAccess`

Look for read/write issues which may cause inaccuracies in the results.

Description

During an **Update Diagram**, identify potential issues relating to read-before-write, write-after-read, and write-after-write conditions for data store blocks.

Results and Recommended Actions

Condition	Recommended Action
Reading and writing (read-before-write or write-after-read condition) occur out of order.	Consider restructuring your model so that the Data Store Read block executes before the Data Store Write block.
Multiple writes occur within a single time step.	Change the model to write data only once per time step or refer to the following Tips section.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

Tips

This check performs a static analysis which might not identify every instance of improper usage. Specifically, Function-Call Subsystems, Stateflow Charts, MATLAB for code generation, For Iterator Subsystems, and For Each Subsystems can cause both missed detections and false positives. For a more comprehensive check, consider enabling the following diagnostics on the **Diagnostics > Data Validity** pane in the Configuration Parameters dialog box: “Detect read before write”, “Detect write after read”, and “Detect write after write”.

See Also

- “Local and Global Data Stores”
- `Data Store Memory`

- `Data Store Read`
- `Data Store Write`
- "Detect read before write"
- "Detect write after read"
- "Detect write after write"
- "What Is a Model Advisor Exclusion?"

Check structure parameter usage with bus signals

Check ID: `mathworks.design.MismatchedBusParams`

Identify blocks and `Simulink.Signal` objects that initialize bus signals by using mismatched structures.

Description

In a model, you can use a MATLAB structure to initialize a bus signal. For example, if you pass a bus signal through a `Unit Delay` block, you can set the **Initial condition** parameter to a structure. For basic information about initializing buses by using structures, see "Specify Initial Conditions for Bus Signals".

Run this check to generate efficient and readable code by matching the shape and numeric data types of initial condition structures with those of bus signals. Matching these characteristics avoids unnecessary explicit typecasts and replaces field-by-field structure assignments with, for example, calls to `memcpy`.

Partial Structures

This check lists blocks and `Simulink.Signal` objects that initialize bus signals by using partial structures. During the iterative process of creating a model, you can use partial structures to focus on a subset of signal elements in a bus. For a mature model, use full structures to:

- Generate readable and efficient code.
- Support a modeling style that explicitly initializes unspecified signals. When you use partial structures, Simulink implicitly initializes unspecified signals.

For more information about full and partial structures, see "Full and Partial IC Structures".

Data Type Mismatches

This check lists blocks and `Simulink.Signal` objects whose initial condition structures introduce data type mismatches. The fields of these structures have numeric data types that do not match the data types of the corresponding bus signal elements.

When you configure an initial condition structure to appear as a tunable global structure in the generated code, avoid unnecessary explicit typecasts by matching the data types. See "Generate Tunable Initial Condition Structure for Bus Signal".

Results and Recommended Actions

Condition	Recommended Action
Block or signal object uses partial structure	Consider using the function `Simulink.Bus.createMATLABStructure` to create a full initial condition structure.
Data types of structure fields do not match data types of corresponding signal elements	Consider defining the structure as a `Simulink.Parameter` object, and creating a `Simulink.Bus` object to use as the data type of the bus signal and of the parameter object. To control numeric data types, use the `Simulink.BusElement` objects in the bus object.

See Also

- "Specify Initial Conditions for Bus Signals"
- "Generate Tunable Initial Condition Structure for Bus Signal"
- "Data Stores with Signal Objects"
- `Simulink.Bus.createMATLABStruct`
- `Simulink.Signal`

Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition

Check ID: `mathworks.design.ReplaceZOHDelayByRTB`

Identify `Delay`, `Unit Delay`, or `Zero-Order Hold` blocks that are used for rate transition. Replace these blocks with actual `Rate Transition` blocks.

Description

If a model uses `Delay`, `Unit Delay`, or `Zero-Order Hold` blocks to provide rate transition between input and output signals, Simulink makes a hidden replacement of these blocks with built-in `Rate Transition` blocks. In the compiled block diagram, a yellow symbol and the letters "RT" appear in the upper-left corner of a replacement block. This replacement can affect the behavior of the model, as follows:

- These blocks lose their algorithmic design properties to delay a signal or implement zero-order hold. Instead, they acquire rate transition behavior.
- This modeling technique works only in specific transition configurations (slow-to-fast for `Delay` and `Unit Delay blocks`, and fast-to-slow for `Zero-Order Hold` block).

Set the block sample time to be equal to the slower rate (source for the `Delay` and `Unit Delay` blocks and destination for the `Zero-Order Hold` block).

- When the block sample time of a downstream or upstream block changes, these `Delay`, `Unit Delay` and `Zero-Order Hold` blocks might not perform rate transition. For example, setting the source and destination sample times equal stops rate transition. The blocks then assume their original algorithmic design properties.
- The block sample time shows incomplete information about sample time rates. The block code runs at two different rates to handle data transfer. However, the block sample time and sample time color show it as a single-rate block. Tools and MATLAB scripts that use sample time information base their behavior on this information.

An alternative is to replace `Delay`, `Unit Delay`, or `Zero-Order Hold` blocks with actual `Rate Transition` blocks.

- The technique ensures unambiguous results in block behavior. `Delay`, `Unit Delay`, or `Zero-Order Hold` blocks act according to their algorithmic design to delay and hold signals respectively. Only `Rate Transition` blocks perform actual rate transition.
- Using an actual `Rate Transition` block for rate transition offers a configurable solution to handle data transfer if you want to specify deterministic behavior or the type of memory buffers to implement.

Use this check to identify instances in your model where `Delay`, `Unit Delay` or `Zero-Order Hold` blocks undergo hidden replacement to provide rate transition between signals. Click **Upgrade Model** to replace these blocks with actual `Rate Transition` blocks.

Results and Recommended Actions

Condition	Recommended Action
Model has no instances of `Delay`, `Unit Delay`, or `Zero-Order Hold` blocks used for rate transition.	No action required.
Model has instances of `Delay`, `Unit Delay`, or `Zero-Order Hold` blocks used for rate transition.	The check identifies these instances and allows you to upgrade the model. 1 Click **Upgrade Model** to replace with actual `Rate Transition` blocks.

Condition	Recommended Action
	2 Save changes to your model.

If you do not choose to replace the `Delay`, `Unit Delay`, and/or `Zero-Order Hold` blocks with actual `Rate Transition` blocks, Simulink continues to perform a hidden replacement of these blocks with built-in rate transition blocks.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- "Run Model Checks"
- "Model Upgrades"
- `Rate Transition`
- "What Is a Model Advisor Exclusion?"

Check for calls to slDataTypeAndScale

Check ID: `mathworks.design.CallslDataTypeAndScale`

Identify calls to the internal function `slDataTypeAndScale`.

Description

In some previous versions of Simulink, opening a model that had been saved in an earlier version triggers an automatic upgrade to code for data type handling. The automatic upgrade inserts calls to the internal function `slDataTypeAndScale`. Although Simulink continues to support some uses of the function, if you eliminate calls to it, you get cleaner and faster code.

Simulink does not support calls to `slDataTypeAndScale` when:

- The first argument is a `Simulink.AliasType` object.
- The first argument is a `Simulink.NumericType` object with property `IsAlias` set to true.

Running **Check for calls to slDataTypeAndScale** identifies calls to `slDataTypeAndScale` that are required or recommended for replacement. In most cases, running the check and following the recommended action removes the calls. You can ignore calls that remain. Run the check unless you are sure there are not calls to `slDataTypeAndScale`.

Results and Recommended Actions

Condition	Recommended Action
Required Replacement Cases	Manually or automatically replace calls to `slDataTypeAndScale`. Cases listed require you to replace calls to `slDataTypeAndScale`.
Recommended Replacement Cases	For the listed cases, it is recommended that you manually or automatically replace calls to `slDataTypeAndScale`.
Manual Inspection Cases	Inspect each listed case to determine whether it should be manually upgraded.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

Tips

- Do not manually insert a call to `slDataTypeAndScale` into a model. The function was for internal use only.
- Running **Check for calls to slDataTypeAndScale** calls the Simulink function `slRemoveDataTypeAndScale`. Calling this function directly provides a wider range of conversion options. However, you very rarely need more conversion options.

See Also

- For more information about upgrading data types and scales, in the MATLAB Command Window, execute the following:
  - `help slDataTypeAndScale`
  - `help slRemoveDataTypeAndScale`
- "What Is a Model Advisor Exclusion?"

Check bus usage

Check ID: `mathworks.design.MuxBlkAsBusCreator`

Identify Mux blocks used as a bus creator and bus signal that Simulink treats as a vector.

Description

Models cannot 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:

- Replace Mux blocks with Bus Creator blocks.
- Insert a Bus to Vector block at the input ports 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.

See Also

- "Prevent Bus and Mux Mixtures"
- "Bus to Vector Block Compatibility Issues"
- Bus to Vector block
- "Mux blocks used to create bus signals"
- "Bus signal treated as vector"
- "Migrating to Simplified Initialization Mode Overview" on page 8-5
- `Simulink.BlockDiagram.addBusToVector`

Check for potentially delayed function-call subsystem return values

Check ID: `mathworks.design.DelayedFcnCallSubsys`

Identify function-call return values that might be delayed because Simulink software inserted an implicit Signal Conversion block.

Description

So that signals reside in contiguous memory, Simulink software can automatically insert an implicit Signal Conversion block in front of function-call initiator block input ports. This can result in a one-step delay in returning signal values from calling function-call subsystems. The delay can be avoided by ensuring the signal originates from a signal block within the function-call system. Or, if the delay is acceptable, insert a Unit Delay block in front of the affected input ports.

Results and Recommended Actions

Condition	Recommended Action
The listed block input ports could have an implicit Signal Conversion block.	Decide if a one-step delay in returning signal values is acceptable for the listed signals. • If the delay is not acceptable, rework your model so that the input signal originates from within the calling subsystem. • If the delay is acceptable, insert a Unit Delay block in front of each listed input port.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

Signal Conversion block

Unit Delay block

"What Is a Model Advisor Exclusion?"

Identify block output signals with continuous sample time and non-floating point data type

Check ID: `mathworks.design.OutputSignalSampleTime`

Find continuous sample time, non-floating-point output signals.

Description

Non-floating-point signals might not represent continuous variables without loss of information.

Results and Recommended Actions

Condition	Recommended Action
Signals with continuous sample times have a non-floating-point data type.	On the identified signals, either change the sample time to be discrete or fixed-in-minor-step ([0 1]).

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

“What Is Sample Time?”.

“What Is a Model Advisor Exclusion?”

Check usage of Merge blocks

Check ID: `mathworks.design.MergeBlkUsage`

Identify Merge blocks with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Note: Run this check along with the other checks in the "Migrating to Simplified Initialization Mode Overview" on page 8-5.

Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. For more information, see "Address Classic Mode Issues by Using Simplified Mode".

This Model Advisor check identifies settings in the Merge blocks in your model that can cause problems if you use classic initialization mode. It also recommends settings for consistent behavior of Merge blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.

Results and Recommended Actions

Condition	Recommended Action
Check the run-time diagnostic setting of the Merge block.	1 In the Configuration Parameters dialog box, on the **All Parameters** tab, set "Detect multiple driving blocks executing at the same time step" to `error`. 2 Verify that the model simulates without errors before running this check again.
Check for Model blocks that are using the PIL simulation mode.	The simplified initialization mode does not support the Processor-in-the-loop (PIL) simulation for model references.

Condition	Recommended Action
Check for library blocks with instances that cannot be migrated.	Examine the failed subcheck results for each block to determine the corrective actions.
Check for single-input Merge blocks.	Replace both the `Mux` block used to produce the input signal and the `Merge` block with one multi-input `Merge` block. Single-input `Merge` blocks are not supported in the simplified initialization mode.
Check for root Merge blocks that have an unspecified **Initial output** value.	If you do not specify an explicit value for the **Initial output** parameter of *root Merge blocks*, then Simulink uses the default initial value of the output data type. A root `Merge` block is a `Merge` block with an output port that does not connect to another `Merge` block. For information on the default initial value, see "Initializing Signal Values".
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.

Condition	Recommended Action
Check for Merge blocks that have unconnected inputs or that have inputs from non-conditionally executed subsystems.	Set the **Number of inputs** parameter of the `Merge` block to the number of Merge block inputs. You must connect each input to a signal. Verify that each `Merge` block input is driven by a conditionally executed subsystem. `Merge` blocks cannot be driven directly by an `Iterator Subsystem` or a block that is not a conditionally executed subsystem.
Check for Merge blocks with inputs that are combined or reordered outside of conditionally executed subsystems.	Verify that combinations or reordering of `Merge` block input signals takes place within a conditionally executed subsystem. Such designs may use `Mux`, `Bus Creator`, or `Selector` blocks.
Check for Merge blocks with inconsistent input sample times.	Verify that input signals to each `Merge` block have the same **Sample time**. Failure to do so could result in unpredictable behavior. Consequently, the simplified initialization mode does not allow inconsistent sample times.
Check for Merge blocks with multiple input ports that are driven by a single source.	Verify that the `Merge` block does not have multiple input signals that are driven by the same conditionally executed subsystem or conditionally executed `Model` block.

Condition	Recommended Action
Check for Merge blocks that use signal objects to specify the **Initial output** value.	Verify that the following behavior is acceptable. In the simplified initialization mode, signal objects cannot specify the **Initial output** parameter of the `Merge` block. While you can still initialize the output signal for a `Merge` block using a signal object, the initialization result may be overwritten by that of the `Merge` block. **Note:** Simulink generates a warning that the initial value of the signal object has been ignored.

See Also

- "Migrating to Simplified Initialization Mode Overview" on page 8-5
- "What Is a Model Advisor Exclusion?"

Check usage of Outport blocks

Check ID: `mathworks.design.InitParamOutportMergeBlk`

Identify Outport blocks and conditional subsystems with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Note: Run this check along with the other checks in the "Migrating to Simplified Initialization Mode Overview" on page 8-5.

Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. This mode is especially important for models that do not specify initial conditions for conditionally executed subsystem output ports. For more information, see "Address Classic Mode Issues by Using Simplified Mode".

This Model Advisor check identifies Outport blocks and conditional subsystems in your model that can cause problems if you use the simplified initialization mode. It also recommends settings for consistent behavior of Outport blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode. Warning statements identify issues or changes in behavior can might occur after migration.

Results and Recommended Actions

Condition	Recommended Action
Check for blocks inside of the Iterator Subsystem that require elapsed time.	Within an Iterator Subsystem hierarchy, do not use blocks that require a service that maintains the time that has elapsed between two consecutive executions. Since an Iterator Subsystem can execute multiple times at a given time step, the concept of elapsed time is not well-defined between two such executions. Using these blocks inside of an Iterator Subsystem can cause unexpected behavior.
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 violations, see the example model `sl_subsys_semantics`.

Condition	Recommended Action
	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 are driven by a bus signal and whose **Initial output** value is not scalar.	For `Outport` blocks driven by bus signals, classic initialization mode does not support Initial Condition (IC) structures, while simplified initialization mode does. Hence, when migrating a model from classic to simplified mode, specify a scalar for the **Initial Output** parameter. After migration completes, to specify different initial values for different elements of the bus signal, use IC structures. For more information, see "Create Initial Condition (IC) Structures".

Condition	Recommended Action
Check for Outport blocks that require an explicit signal copy.	An explicit copy of the bus signal driving the `Outport` block is required for the initialization of the output signal of the corresponding subsystem. Insert a `Signal Conversion` block before the `Outport` block, then set the Output parameter of the `Signal Conversion` block to `Bus copy`. A standard data signal may require an additional signal copy for one or more of the following reasons: • A block with output that cannot be overwritten is driving the `Outport` block. The `Ground` block and the `Constant` block are examples of such blocks. • The `Outport` block shares the same signal source with another `Outport` block in the same subsystem or in one nested within the current subsystem but having a different initial output value. • The `Outport` block connects to the input of a `Merge` block • One of the input signals of the `Outport` block is specifying a Simulink.Signal object with an explicit initial value.
Check for merged Outport blocks that inherit the **Initial Output** value from Outport blocks that have been configured to reset when the blocks become disabled.	When `Outport` blocks are driving a Merge block, do not set their **Output when disabled** parameters to `reset`.

Condition	Recommended Action
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`.
Check for Outport blocks that have an undefined **Initial output** value with invalid initial condition sources.	Verify that the following behavior is acceptable. When the **Initial output** parameter is unspecified ([ ]), it inherits the initial output from the source blocks. If at least one of the sources of the `Outport` block is not a valid source to inherit the initial value, the block uses the default initial value for that data type. For simplified initialization mode, valid sources an `Outport` blocks can inherit the **Initial output** value from are: `Constant`, `Initial Condition`, `Merge` (with initial output), Stateflow chart, function-call model reference, or conditionally executed subsystem blocks.

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. 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 and have an undefined **Initial output** value.	Verify that the following behavior is acceptable. Specify the **Initial output** parameter for the `Outport` block. Set this value to `[]` (empty matrix) to use the default initial value of the output data type.
Check the **Initial output** setting of Outport blocks that reset when they are disabled.	Specify the **Initial output** parameter of the `Outport` block. You must specify the **Initial output** value for blocks that are configured to reset when they become disabled.
Check the **Initial output** setting for Outport blocks that pass through a function-call data dependency signal.	You cannot specify an **Initial output** value for the `Outport` block because function-call data dependency signals are passing through it. To set the **Initial output** value: 1 Set the **Initial output** parameter of the `Outport` block to `[]`. 2 Provide the initial value at the source of the data dependency signal rather than at the `Outport` block.

Condition	Recommended Action
Check for Outport blocks that use signal objects to specify the **Initial output** value.	Verify that the following behavior is acceptable. In the simplified initialization mode, signal objects cannot specify the **Initial output** parameter of an `Outport` block. You can still initialize the input or output signals for an `Outport` block using signal objects, but the initialization results may be overwritten by those of the `Outport` block. **Note:** If you are working with a conditionally executed subsystem `Outport` block, Simulink generates a warning that the initial value of the signal object has been ignored.
Check for library blocks with instances that have warnings.	Examine the warning subcheck results for each block before migrating to the simplified initialization mode.
Check for merged Outport blocks that are either unconnected or connected to a Ground block.	Verify that the following behavior is acceptable. The `Outport` block is driving a `Merge` block, but its inputs are either unconnected or connected to `Ground` blocks. In the classic initialization mode, unconnected or grounded outports do not update the merge signal even when their parent conditionally executed subsystems are executing. In the simplified initialization mode, however, these outports will update the merge signal with a value of zero when their parent conditionally executed subsystems are executing.

Condition	Recommended Action
Check for Outport blocks that obtain the **Initial output** value from an input signal when they are migrated.	Verify that the following behavior is acceptable. The **Initial output** parameter of the `Outport` block is not specified. As a result, the simplified initialization mode will assume that the **Initial output** value for the `Outport` block is derived from the input signal. This assumption may result in different initialization behavior. If this behavior is not acceptable, modify your model before you migrate to the simplified initialization mode.
Check for outer `Outport` blocks that have an explicit **Initial output**.	Verify that the following behavior is acceptable. In classic initialization mode, the **Initial output** and **Output when disabled** parameters of the `Outport` block must match those of their source `Outport` blocks. In simplified initialization mode, Simulink sets the **Initial output** parameter of outer Outport blocks to `[ ]` (empty matrix) and **Output when disabled** parameter to `held`.

Condition	Recommended Action
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.
Check for blocks that read input from conditionally executed subsystems during initialization.	Verify that the following behavior is acceptable. Some blocks, such as the `Discrete-Time Integrator` block, read their inputs from conditionally executed subsystems during initialization in the classic initialization mode. Simulink performs this step as an optimization technique. This optimization is not allowed in the simplified initialization mode because the output of a conditionally executed subsystem at the first time step after initialization may be different than the initial value declared in the corresponding `Outport` block. In particular, this discrepancy occurs if the subsystem is active at the first time step.

Condition	Recommended Action
Check for a migration conflict for Outport blocks that use a `Dialog` as the **Source of initial output value**.	Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the **Check for library blocks with instances that cannot be migrated** to learn about the different migration paths for other instances of each `Outport` block. The `Outport` block will maintain its current settings and use its specified **Initial output** value.
Check for a migration conflict for Outport blocks that use `Input signal` as the **Source of initial output** value.	Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the **Check for library blocks with instances that cannot be migrated** to learn about the different migration paths for other instances of each `Outport` block. The `Outport` block currently specifies an **Initial output** of `[]` (empty matrix), and **the Output when disabled** as `held`. This means that each outport does not perform initialization, but implicitly relies on source blocks to initialize its input signal. After migration, the parameter **Source of initial output value** will be set to `Input signal` to reflect this behavior.

Condition	Recommended Action
Check for a migration conflict for Outport blocks that have SimEvents® semantics.	Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the **Check for library blocks with instances that cannot be migrated** to learn about the different migration paths for other instances of each `Outport` block. The `Outport` blocks will continue to use an **Initial output** value of `[ ]` (empty matrix) and an **Output when disabled** setting of `held`. Simulink will maintain these settings because their parent conditionally executed subsystems are connected to SimEvents blocks.
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.

Condition	Recommended Action
Check for a migration conflict for Outport blocks that use a default ground value as the Initial output.	The parameter **Initial output** is set to `[]` (empty matrix) and the source of the `Outport` is an invalid initial condition source. Thus, the block uses the default initial value as the initial output in the simplified initialization mode. Other instances of Outport blocks with the same library link either have errors or are being migrated differently.
Check for a migration conflict for merged Outport blocks without explicit specification of **Initial output**.	Review the results from the subcheck **Check for library blocks with instances that cannot be migrated** to learn about different migration paths for other instances of each Outport block. For the remaining Outport blocks, **Initial output** is set to `[]` (empty matrix) and **Output when disabled** is set to `held` respectively, in simplified initialization mode.

See Also

- "Migrating to Simplified Initialization Mode Overview" on page 8-5
- "What Is a Model Advisor Exclusion?"

Check usage of Discrete-Time Integrator blocks

Check ID: `mathworks.design.DiscreteBlock`

Identify Discrete-Time Integrator blocks with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Note: Run this check along with the other checks in the "Migrating to Simplified Initialization Mode Overview" on page 8-5.

Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. For more information, see "Address Classic Mode Issues by Using Simplified Mode".

This Model Advisor check identifies settings in Discrete-Time Integrator blocks in your model that can cause problems if you use the simplified initialization mode. It also recommends settings for consistent behavior of Discrete-Time Integrator blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.

Results and Recommended Actions

Condition	Recommended Action
Check for Discrete-Time Integrator blocks whose parameter **Initial condition setting** is set to `Output`.	Determine if the new behavior of the `Discrete-Time Integrator` blocks is acceptable. If it is not acceptable, modify the model to account for the new behavior before migrating to the simplified initialization mode.
Check for Discrete-Time Integrator blocks whose **Initial condition setting** parameter is set to `State (most efficient)` and are in a subsystem that uses triggered sample time.	Use periodic sample time for the block, or set **Initial Condition setting** to `Output`.
Check for blocks inside of the Iterator Subsystem that require elapsed time.	Within an Iterator Subsystem hierarchy, do not use blocks that require a service that maintains the time that has elapsed between two consecutive executions. Since an Iterator Subsystem can execute multiple times at a given time step, the concept of elapsed time is not well-defined between two such executions. Using these blocks inside of an Iterator Subsystem can cause unexpected behavior.

See Also

- "Migrating to Simplified Initialization Mode Overview" on page 8-5
- "What Is a Model Advisor Exclusion?"

Check model settings for migration to simplified initialization mode

Note: Do not run this check in isolation. Run this check along with the other checks in the "Migrating to Simplified Initialization Mode Overview" on page 8-5.

Check ID: `mathworks.design.ModelLevelMessages`

Identify settings in Model blocks and model configuration parameters that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Description

Simplified initialization mode was introduced in R2008b to improve consistency of simulation results. For more information, see "Address Classic Mode Issues by Using Simplified Mode".

This Model Advisor check identifies issues in the model configuration parameters and `Model` blocks in your model that can cause problems when you migrate to simplified initialization mode. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.

Note: Before running this check, verify that your block diagram conforms to the modeling standards set by this diagnostic.

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` and click **OK**.

For more information, see "Diagnostics Pane: Connectivity".

After running this Model Advisor consistency check, if you click **Explore Result** button, the messages pertain only to blocks that are not library-links.

Note: Because it is difficult to undo these changes, select **File > Save Restore Point As** to back up your model before migrating to the simplified initialization mode.

Results and Recommended Actions

Condition	Recommended Action
Verify that all Model blocks are using the simplified initialization mode.	Migrate the model referenced by the Model block to the simplified initialization mode, then migrate the top model.
Verify simplified initialization mode setting	Set **Configuration Parameters > All Parameters > Underspecified initialization detection** to `Simplified`.

Action Results

Clicking **Modify Settings** causes the following:

- The Model parameter is set to `simplified`
- If an Outport block has the **Initial output** parameter set to the empty 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

- “Migrating to Simplified Initialization Mode Overview” on page 8-5
- “What Is a Model Advisor Exclusion?”

Check for non-continuous signals driving derivative ports

Check ID: `mathworks.design.NonContSigDerivPort`

Identify noncontinuous signals that drive derivative ports.

Description

Noncontinuous signals that drive derivative ports cause the solver to reset every time the signal changes value, which slows down simulation.

Results and Recommended Actions

Condition	Recommended Action
There are noncontinuous signals in the model driving derivative ports.	• Make the specified signals continuous. • Replace the continuous blocks receiving these signals with discrete state versions of the blocks.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- "Modeling Dynamic Systems"
- "Simulation Phases in Dynamic Systems"
- "What Is a Model Advisor Exclusion?"

Runtime diagnostics for S-functions

Check ID: `mathworks.design.DiagnosticSFcn`

Check array bounds and solver consistency if `S-Function` blocks are in the model.

Description

Validates whether `S-Function` blocks adhere to the ODE solver consistency rules that Simulink applies to its built-in blocks.

Results and Recommended Actions

Condition	Recommended Action
Solver data inconsistency is set to `none`.	In the Configuration Parameters dialog box, on the **All Parameters** tab, set **Solver data inconsistency** to `warning` or `error`.
Array bounds exceeded is set to `none`.	In the Configuration Parameters dialog box, on the **All Parameters** tab, set **Array bounds exceeded** to `warning` or `error`

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- "What Is an S-Function?"
- "How S-Functions Work"
- "What Is a Model Advisor Exclusion?"

Check model for foreign characters

Check ID: `mathworks.design.characterEncoding`

Check for characters that are incompatible with the current encoding

Description

Check for characters in the model file that cannot be represented in the current encoding. These can cause errors during simulation, and may be corrupted when saving the model.

Results and Recommended Actions

Condition	Recommended Action
Incompatible characters found	Change the current encoding to the encoding specified in the model file, using `slCharacterEncoding`. To change the current encoding you need to close the models, and this closes the Model Advisor.

Tips

The Upgrade Advisor report shows the encoding you need, or you can retrieve the encoding from the model using the command:

```
get_param(modelname,'SavedCharacterEncoding')
```

Use `slCharacterEncoding` to change the encoding. This setting applies to the current MATLAB session, so if you restart MATLAB and want to open the same model, you will need to make the same change to the current encoding again.

For more information see:

- `slCharacterEncoding`
- “Open a Model with Different Character Encoding”
- “Save Models with Different Character Encodings”

See Also

- “Consult the Upgrade Advisor”.

- “Model Upgrades”

Identify unit mismatches in the model

Identify instances of unit mismatches between ports in the model.

Description

Check for instances of unit mismatches between ports in the model.

Results and Recommended Actions

Condition	Recommended Action
Unit mismatches found	Change one of the mismatched unit settings to match the unit settings for the other port.

See Also

- "Unit Specification in Simulink Models".

Identify automatic unit conversions in the model

Identify instances of automatic unit mismatches in the model.

Description

Identify instances of automatic unit mismatches in the model.

Results and Recommended Actions

Condition	Recommended Action
Automatic unit conversions found	Check that the converted units are expected for the model.

See Also

- "Unit Specification in Simulink Models".

Identify disallowed unit systems in the model

Identify instances of disallowed unit systems in the model.

Description

Identify instances of disallowed unit systems in the model.

Results and Recommended Actions

Condition	Recommended Action
Disallowed unit systems found	Either choose a unit that conforms to the configured unit system, or select another unit system. For more information, see “Restricting Unit Systems”.

See Also

- “Unit Specification in Simulink Models”.

Identify undefined units in the model

Identify instances of unit specifications, not defined in the unit database, in the model.

Description

Identify instances of unit specifications, not defined in the unit database, in the model.

Results and Recommended Actions

Condition	Recommended Action
Undefined units found	Change the unit to one that Simulink supports.

See Also

- “Unit Specification in Simulink Models”.
- Allowed Units

Check model for block upgrade issues

Check ID: `mathworks.design.Update`

Check for common block upgrade issues.

Description

Check blocks in the model for compatibility issues resulting from using a new version of Simulink software.

Results and Recommended Actions

Condition	Recommended Action
Blocks with compatibility issues found.	Click **Modify** to fix the detected block issues.
Check update status for the Level 2 API S-functions.	Consider replacing Level 1 S-functions with Level 2.

Action Results

Clicking **Modify** replaces blocks from a previous release of Simulink software with the latest versions.

See Also

- “Write Level-2 MATLAB S-Functions”.
- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check model for block upgrade issues requiring compile time information

Check ID: `mathworks.design.UpdateRequireCompile`

Check for common block upgrade issues.

Description

Check blocks for compatibility issues resulting from upgrading to a new version of Simulink software. Some block upgrades require the collection of information or data when the model is in the compile mode. For this check, the model is set to compiled mode and then checked for upgrades.

Results and Recommended Actions

Condition	Recommended Action
Model contains `Lookup Table` or `Lookup Table (2-D)` blocks and some of the blocks specify **Use Input Nearest** or **Use Input Above** for a lookup method.	Replace `Lookup Table` blocks and `Lookup Table (2-D)` blocks with `n-D Lookup Table` blocks. Do not apply **Use Input Nearest** or **Use Input Above** for lookup methods; select another option.
Model contains `Lookup Table` or `Lookup Table (2-D)` blocks and some blocks perform multiplication first during interpolation.	Replace `Lookup Table` blocks and `Lookup Table (2-D)` blocks with `n-D Lookup Table` blocks. However, because the `n-D Lookup Table` block performs division first, this replacement might cause a numerical difference in the result.
Model contains `Lookup Table` or `Lookup Table (2-D)` blocks. Some of these blocks specify **Interpolation-Extrapolation** as the **Lookup method** but their input and output are not the same floating-point type.	Replace `Lookup Table` blocks and `Lookup Table (2-D)` blocks with `n-D Lookup Table` blocks. Then change the extrapolation method or the port data types for block replacement.
Model contains `Unit Delay` blocks with **Sample time** set to `-1` that inherit a continuous sample time.	Replace `Unit Delay` blocks with `Memory` blocks.

Action Results

Clicking **Modify** replaces blocks from a previous release of Simulink software with the latest versions.

See Also

- n-D Lookup Table
- `Unit Delay`
- "Consult the Upgrade Advisor"
- "Model Upgrades"

Check that the model is saved in SLX format

Check ID: `mathworks.design.UseSLXFile`

Check that the model is saved in SLX format.

Description

Check whether the model is saved in SLX format.

Results and Recommended Actions

Condition	Recommended Action
Model not saved in SLX format	Consider upgrading to the SLX file format to use the latest features in Simulink.

Capabilities and Limitations

You can run this check on your library models.

Tips

Simulink Projects can help you upgrade models to SLX format and preserve file revision history in source control. See “Upgrade Model Files to SLX and Preserve Revision History”.

See Also

- “Save Models in the SLX File Format”
- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check model for SB2SL blocks

Check ID: `mathworks.simulink.SB2SL.Check`

Check that the model does not have outdated SB2SL blocks.

Description

Check if the model contains outdated SB2SL blocks.

Results and Recommended Actions

Condition	Recommended Action
Model contains outdated SB2SL blocks	Consider upgrading the model to current SB2SL blocks.

Action Results

Clicking **Update SB2SL Blocks** replaces blocks with the latest versions.

See Also

- "Consult the Upgrade Advisor".

Check Model History properties

Check ID: `mathworks.design.SLXModelProperties`

Check for edited model history properties

Description

Check models for edited Model History property values that could be used with source control tool keyword substitution. This keyword substitution is incompatible with SLX file format.

In the MDL file format you can configure some model properties to make use of source control tool keyword substitution. If you save your model in SLX format, source control tools cannot perform keyword substitution. Information in the model file from such keyword substitution is cached when you first save the MDL file as SLX, and is not updated again. The Model Properties History pane and Model Info blocks in your model show stale information from then on.

Results and Recommended Actions

Condition	Recommended Action
Edited model history properties	Manually or automatically reset the properties to the default values. Click the button to reset, or to inspect and change these properties manually, open the Model Properties dialog and look in the History pane.

Capabilities and Limitations

You can run this check on your library models.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Identify Model Info blocks that can interact with external source control tools

Check ID: `mathworks.design.ModelInfoKeywordSubstitution`

Use this check to find Model Info blocks that can be altered by external source control tools through keyword substitution.

Description

This check searches for strings in the Model Info block enclosed within dollar signs that can be overwritten by an external source control tool. Using third-party source control tool keyword expansion might corrupt your model files when you submit them. Keyword substitution is not available in SLX model file format.

For a more flexible interface to source control tools, use a Simulink project instead of the Model Info block. See “About Source Control with Projects”.

Results and Recommended Actions

Condition	Recommended Action
The Model Info block contains fields like this: `$keyword$`	Review the list of fields in the report, then remove the keyword strings from the Model Info block.

See Also

- “Consult the Upgrade Advisor”.
- “About Source Control with Projects”

Identify Model Info blocks that use the Configuration Manager

Check ID: `mathworks.design.ModelInfoConfigurationManager`

Use this check to find Model Info blocks that use the Configuration Manager.

Description

Model Info blocks using the Configuration Manager allow risky keyword substitution using external source control tools. Using third-party source control tool keyword expansion might corrupt your model files when you submit them. Keyword substitution is not available in SLX model file format. The Configuration Manager for the Model Info block will be removed in a future release.

For a more flexible interface to source control tools, use a Simulink project instead of the Model Info block. See “About Source Control with Projects”.

Results and Recommended Actions

Condition	Recommended Action
A Model Info block is using the Configuration Manager.	Click **Remove the Configuration Manager**.

See Also

- “Consult the Upgrade Advisor”.
- “About Source Control with Projects”

Check for Mux blocks used to create bus signals

Check ID: `mathworks.design.SLXModelProperties`

Identify Mux blocks used as a bus creator.

Description

Models cannot contain Mux blocks that output bus signals. Instead, replace those Mux blocks with Bus Creator blocks.

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`.

Action Results

Clicking **Modify** replaces Mux blocks with Bus Creator blocks.

Tip

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.

See Also

- "Prevent Bus and Mux Mixtures"
- "Address Compatibility Issues After Running Upgrade Advisor"
- "Bus to Vector Block Compatibility Issues"
- Bus to Vector block
- "Mux blocks used to create bus signals"

Check bus usage

Check virtual bus usage and for Mux blocks used to create bus signals.

Description

Models cannot contain bus signals that Simulink software implicitly converts to vectors. **Check bus usage** checks virtual bus usage and for Mux blocks used to create bus signals.

Results and Recommended Actions

Condition	Recommended Action
Model uses Mux blocks to create bus signals.	In the Upgrade Advisor, click **Modify**.
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`.
Virtual bus signal input to these blocks: • Assignment • Delay (if you specify an initial condition from the dialog that is a MATLAB structure or zero and the value for **State name** is not empty) • Permute Dimension • Reshape • Selector • Unit Delay (if you specify an initial condition that is a MATLAB structure or zero and the value for **State name** is not empty) • Vector Concatenate	In the Upgrade Advisor, click **Modify**. The check inserts a Bus to Vector block to attempt to convert virtual bus input signals to vector signals. For issues that the Upgrade Advisor identifies but cannot fix, modify the model manually. For details, see “Correct Buses Used as Muxes” and “Prevent Bus and Mux Mixtures”.

Tips

- Run this check before running the **Check consistency of initialization parameters for Outport and Merge blocks** check.
- Starting in R2015b, virtual bus signal inputs to blocks that require nonbus or nonvirtual bus input can cause an error. An error occurs when a virtual bus input signal is generated by a block that specifies a bus object as its output data type. Examples of blocks that can specify a bus object as their output data type include a Bus Creator block or a root Inport block. The blocks that cause an error when they have a virtual bus input in this situation are:

  - Assignment
  - Delay

    This block causes an error only if you set an initial condition from the dialog that is a MATLAB structure or zero and you specify a value for **State name**.
  - Permute Dimension
  - Reshape
  - Selector
  - Unit Delay

    This block causes an error only if you set an initial condition from the dialog that is a MATLAB structure or zero and you specify a value for **State name**.
  - Vector Concatenate

See Also

- "Prevent Bus and Mux Mixtures"
- "Bus to Vector Block Compatibility Issues"
- Bus to Vector block
- "Mux blocks used to create bus signals"
- "Bus signal treated as vector"
- `Simulink.BlockDiagram.addBusToVector`

Check model for legacy 3DoF or 6DoF blocks

Check ID: `mathworks.design.Aeroblks.CheckDOF`

Lists 3DoF and 6DoF blocks are outdated.

Description

This check searches for 3DoF and 6DoF blocks from library versions prior to 3.13 (R2014a).

Results and Recommended Actions

Condition	Recommended Action
Blocks configured with old versions of 3DoF or 6DoF blocks found.	Click **Replace 3DoF and 6DoF Blocks** to replace the blocks with latest versions.

Action Results

Clicking **Replace 3DoF and 6DoF Blocks** replaces blocks with the latest versions.

See Also

- "Equations of Motion"

Check model and local libraries for legacy Aerospace Blockset blocks

Check ID: `mathworks.design.Aeroblks.CheckFG`

Lists blocks configured to use FlightGear versions that are outdated or not supported.

Description

This check searches and lists blocks configured to use FlightGear versions that are outdated or not supported.

Results and Recommended Actions

Condition	Recommended Action
Blocks configured with old versions of FlightGear are found.	Click **Update FlightGear blocks** to change block settings to latest supported version of FlightGear. Then, download latest version of FlightGear that MATLAB supports.

Action Results

Clicking **Update FlightGear blocks** changes block settings to the latest supported version of FlightGear.

See Also

- “Flight Simulator Interfaces”

Check and update masked blocks in library to use promoted parameters

Check ID: `mathworks.design.CheckAndUpdateOldMaskedBuiltinBlocks`

Check for libraries that should be updated to use promoted parameters.

Description

This check searches libraries created before R2011b for masked blocks that should be updated to use promoted parameters. Since R2011b, if a block parameter is not promoted, its value in the linked block is locked to its value in the library block. This check excludes blocks of type Subsystem, Model reference, S-Function and M-S-Function.

Results and Recommended Actions

Condition	Recommended Action
Libraries that need to be updated are found	Click **Update**. Once the libraries have been updated, run the check again

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- "Consult the Upgrade Advisor".
- "Model Upgrades"
- "What Is a Model Advisor Exclusion?"

Check and update mask image display commands with unnecessary imread() function calls

Check ID: `mathworks.design.CheckMaskDisplayImageFormat`

Check identifies masks using image display commands with unnecessary calls to the `imread()` function.

Description

This check searches for the mask display commands that make unnecessary calls to the `imread()` function, and updates them with mask display commands that do not call the `imread()` function. Since 2013a, a performance and memory optimization is available for mask images specified using the image path instead of the RGB triple matrix.

Results and Recommended Actions

Condition	Recommended Action
Mask display commands that make unnecessary calls to the `imread()` function are found.	Click **Update**. Once the blocks have been updated, run the check again.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”
- “What Is a Model Advisor Exclusion?”

Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter

Check ID: `mathworks.design.CheckAndUpdateOldMaskTabnames`

This check identifies masked blocks that specify tabs in mask dialog using the `MaskTabNames` parameter.

Description

This check identifies masked blocks that use the `MaskTabNames` parameter to programmatically create tabs in the mask dialog. Since R2013b, dialog controls are used to group parameters in a tab on the mask dialog.

Results and Recommended Actions

Condition	Recommended Action
Masked blocks commands that use the `MaskTabNames` parameter to programmatically create tabs in the mask dialog are found.	Click **Upgrade**. Once the blocks have been updated, run the check again.

Capabilities and Limitations

You can run this check on your library models.

See Also

- "Consult the Upgrade Advisor".
- "Model Upgrades"

Identify questionable operations for strict single-precision design

Check ID: `mathworks.design.StowawayDoubles`

For a strict single-precision design, this check identifies the blocks that introduce double-precision operations, and non-optimal model settings.

Description

For a strict single-precision design, this check identifies the blocks that introduce double-precision operations, and non-optimal model settings.

Results and Recommended Actions

Condition	Recommended Action
Double-precision floating-point operations found in model.	Verify that: • Block input and output data types are set correctly. • In the Configuration Parameters dialog box, on the **Optimization** pane, **Default for underspecified data type** is set to `single`.
Model uses a library standard that is not optimal for strict-single designs.	Verify that: • All target-specific math libraries used by the model support single-precision implementations. Set **Configuration Parameters > All Parameters > Standard math library** to `C99 (ISO)`.
Logic signals are not implemented as boolean data.	Verify that: • In the Configuration Parameters dialog box, on the **All Parameters** tab, **Implement**

Condition	Recommended Action
	logic signals as Boolean data is selected.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- "Consult the Upgrade Advisor".
- "Model Upgrades"
- "What Is a Model Advisor Exclusion?"

Check get_param calls for block CompiledSampleTime

Check ID: `mathworks.design.CallsGetParamCompiledSampleTime`

Use this check to identify MATLAB files in your working environment that contain `get_param` function calls to return the block `CompiledSampleTime` parameter.

Description

For multi-rate blocks (including subsystems), Simulink returns the block compiled sample time as a cell array of the sample rates in the block. The return value is a cell array of pairs of doubles. MATLAB code that accepts this return value only as pairs of doubles can return an error when called with a multi-rate block. Use this check to identify such code in your environment. Modify these instances of code to accept a cell array of pairs of doubles instead.

For example, consider a variable blkTs, which has been assigned the compiled sample time of a multi-rate block.

```
blkTs = get_param(block,'CompiledSampleTime');
```

Here are some examples in which the original code works only if blkTs is a pair of doubles and the block is a single-rate block:

- Example 1

```
if isinf(blkTs(1))
    disp('found constant sample time')
end
```

Since blkTs is now a cell array, Simulink gives this error message:

```
Undefined function 'isinf' for input arguments of type 'cell'
```

Instead, use this code, for which blkTs can be a cell array or a pair of doubles.

```
if isequal(blkTs, [inf,0])
    disp('found constant sample time')
end
```

- Example 2

```
if all(blkTs == [-1,-1])
    disp('found triggered sample time')
end
```

For the above example, since blkTs is now a cell array, Simulink gives this error:

```
Undefined function 'eq' for input arguments of type 'cell'
```

Instead, use this code, for which blkTs can be a cell array or a pair of doubles.

```
if isequal(blkTs, [-1,-1])
    disp('found triggered sample time')
end
```

- Example 3

```
if (blkTs(1) == -1)
    disp('found a triggered context')
end
```

Again, since blkTs is now a cell array, Simulink gives this error:

```
Undefined function 'eq' for input arguments of type 'cell'
```

Instead, use this code.

```
if ~iscell(blkTs)
    blkTs = {blkTs};
end
for idx = 1:length(blkTs)
    thisTs = blkTs{idx};
```

```
    if (thisTs(1) == -1)
        disp('found a triggered context')
    end
end
```

The above code checks for a triggered type sample time (triggered or async). In cases in which a block has constant sample time ([inf,0]) in addition to triggered or async or when a block has multiple async rates, this alternative property detects the triggered type sample time.

This check scans MATLAB files in your environment. If the check finds instances of MATLAB code that contain `get_param` calls to output the block compiled sample time, Upgrade Advisor displays these results. It suggests that you modify code that accepts the block compiled sample time from multi-rate blocks.

Results and Recommended Actions

Condition	Recommended Action
No MATLAB files call `get_param(block,CompiledSampleTime`	None
Some MATLAB files call `get_param(block,CompiledSampleTime`	If files use the block `CompiledSampleTime` parameter from multi-rate blocks, modify these files to accept the parameter as a cell array of pairs of doubles.

See Also

- "Sample Times in Subsystems"
- "Block Compiled Sample Time"

Check model for parameter initialization and tuning issues

Check ID: `mathworks.design.ParameterTuning`

Use this check to identify issues in the model that occur when you initialize parameters or tune them.

Description

This check scans your model for parameter initialization and tuning issues like:

- Rate mismatch between blocks
- Divide by zero issue in conditionally executed subsystems
- Invalid control port value in `Index Vector` blocks

Results and Recommended Actions

Condition	Recommended Action
The model has rate transition issues.	Select **Automatically handle rate transition for data transfer** in the **Solver** pane of the model configuration parameters.
The model has a divide by zero issue in a conditionally executed subsystem with a control port.	At the command prompt, run `set_param(control_port,'DisallowConstTsAndPrmTs'`
The model has an invalid control port value in a conditionally executed subsystem.	At the command prompt, run `set_param(control_port,'DisallowConstTsAndPrmTs'`

Action Results

Select **Upgrade model** to resolve issues in the model related to parameter initialization and tuning.

See Also

- "Automatic Rate Transition"

Check virtual bus across model reference boundaries

Check ID: `mathworks.design.CheckVirtualBusAcrossModelReference`

Check virtual bus signals that cross model reference boundaries.

Description

This check identifies root-level Inport and Outport blocks in a referenced models and Model blocks with virtual bus outputs that require updates to change to nonvirtual bus signals.

If the check identifies issues, click **Update Model** to apply these changes:

- For root-level Inport blocks — Enable the **Output as nonvirtual bus** parameter and insert a Signal Conversion block after the Inport block. The Signal Conversion block is configured to output a virtual bus.
- For root-level Outport blocks — Enable the **Output as nonvirtual bus in parent model** parameter.
- For Model blocks — For ports whose Outport blocks were updated to address issues, insert a Signal Conversion block after the corresponding ports of the Model block. The Signal Conversion block is configured to output a virtual bus.

Recommended Action and Results

To resolve issues, click **Upgrade Model**.

Note: Run the **Analyze model hierarchy and continue upgrade sequence** check on the top-level model and then down through the model reference hierarchy.

For virtual bus signals that cross model reference boundaries, the check updates the model in these situations:

- Function prototype control — If you use function prototype controls, clicking **Upgrade Model** updates root-level Inport and Outport blocks that have virtual bus inputs or outputs.
- C++ code generation with `I/O arguments step method` — You can set the function specification to `I/O arguments step method` using these steps: Set **Language** parameter to `C++`, `I/O arguments step method` as the function specification for generating C++ code,

  1. In the **Configuration Parameters** > **Code Generation** pane, set **Language** to `C++`.
  2. In the **Configuration Parameters** > **Code Generation** > **Interface** pane, click **Configure C++ Class Interface**.
  3. In the dialog box, set the **Function specification** parameter to `I/O arguments step method`.

  With that model setting, when you click **Upgrade Model**, the advisor updates root-level Inport and Outport blocks that have virtual bus inputs or outputs. Alternatively, you can change the C++ code generation function specification setting to `Default step method`.

- Buses with variable-dimension signals — For virtual buses used by root-level Inport and Outport blocks that contain signals with variable-dimensions in the bus hierarchy, clicking **Upgrade Model** updates the affected Inport and Outport block ports.
- Non-auto storage class for Outport block signals — For input signals to root-level Outport blocks that have an associated non-auto storage class, clicking **Upgrade Model** updates the affected Outport block ports.
- Model blocks that reference models containing fixed Outport blocks — Clicking **Upgrade Model** updates Model blocks referencing the models that had Outport blocks fixed by the **Analyze model hierarchy and continue upgrade sequence** check.

See Also

- "Bus Data Crossing Model Reference Boundaries"

Check model for custom library blocks that rely on frame status of the signal

Check ID: `mathworks.design.DSPFrameUpgrade`

This check identifies custom library blocks in the model that depend on the frame status of the signal.

Description

This check searches for the custom library blocks in a model that depend on the frame status of the signal. The check analyzes the blocks, recommends fixes, and gives reasons for the fixes. You must make the fixes manually.

Results and Recommended Actions

Condition	Recommended Action
The check finds custom library blocks that depend on the frame status of the signal.	Follow the recommendation given by the Upgrade Advisor.

Capabilities and Limitations

You can run this check only on custom library blocks in your model.

You must make the fixes manually.

This check appears only if you have the DSP System Toolbox installed.

See Also

"Frame-based processing"

Check Rapid Accelerator signal logging

Check ID: `mathworks.design.CheckRapidAcceleratorSignalLogging`

When simulating your model in Rapid Accelerator mode, use this check to find signals logged in your model that are globally disabled. Rapid Accelerator mode supports signal logging. Use this check to enable signal logging globally.

Description

This check scans your model to see if a simulation is in Rapid Accelerator mode and whether the model contains signals with signal logging. If the check finds an instance and signal logging is globally disabled, an option to turn on signal logging globally appears.

Results and Recommended Actions

Condition	Recommended Action
Simulation mode is not Rapid Accelerator.	None You can enable signal logging in Rapid Accelerator mode.
Simulation mode is Rapid Accelerator. Upgrade Advisor did not find signals with signal logging enabled.	NoneThe model does not use signal logging. Enable signal logging for signals and globally if you want to log signals.
Simulation mode is Rapid Accelerator. Upgrade Advisor found signals with signal logging enabled. However, global setting for signal logging was disabled.	Enable signal logging globally if you want to log signals with signal logging enabled.
Signal logging was already globally enabled.	None

Action Results

Selecting **Modify** enables signal logging globally in your model.

See Also

- “Signal Logging in Rapid Accelerator Mode”
- “Consult the Upgrade Advisor”.

Check for root outports with constant sample time

Check ID: `mathworks.design.CheckConstRootOutportWithInterfaceUpgrade`

Use this check to identify root outports with a constant sample time used with an AUTOSAR target, Function Prototype Control, or the model C++ class interface.

Description

Root outports with constant sample time are not supported when using an AUTOSAR target, Function Prototype Control, or the model C++ class interface. Use this check to identify root Outport blocks with this condition and modify the blocks as recommended.

Results and Recommended Actions

Condition	Recommended Action
Root outport with constant sample time used with an AUTOSAR target, Function Prototype Control or the model C++ class interface.	Consider one of the following: • Set the sample time of the block to the fundamental sample time. • Identify the source of the constant sample time and set its sample time to the fundamental sample time. • Place a Rate Transition block with inherited sample time (-1) before the block.

See Also

- “Consult the Upgrade Advisor”.

Analyze model hierarchy and continue upgrade sequence

Check ID: `com.mathworks.Simulink.UpgradeAdvisor.UpgradeModelHierarchy`

Check for child models and guide you through upgrade checks.

Description

This check identifies child models of this model, and guides you through upgrade checks to run both non-compile and compile checks. The Advisor provides tools to help with these tasks:

- If the check finds child models, it offers to run the Upgrade Advisor upon each child model in turn and continue the upgrade sequence. If you have a model hierarchy you need to check and update each child model in turn.
- If there are no child models, you still need to continue the check sequence until you have run both non-compile and compile checks.

You must run upgrade checks in this order: first the checks that do not require compile time information and do not trigger an Update Diagram, then the compile checks.

Click **Continue Upgrade Sequence** to run the next checks. If there are child models, this will open the next model. Keep clicking **Continue Upgrade Sequence** until the check passes.

Results and Recommended Actions

Condition	Recommended Action
Child models found	Click **Continue Upgrade Sequence** to run the next checks. If there are child models, this will close the current Upgrade Advisor session, and open Upgrade Advisor for the next model in the hierarchy.
No child models, but more checks to run	If there are no child models, click **Continue Upgrade Sequence** to refresh the Upgrade Advisor with compilation checks selected. The compile checks trigger an Update Diagram (marked with ^). Run

Condition	Recommended Action
	the next checks and take advised actions. When you return to this check, click **Continue Upgrade Sequence** until this check passes.

Tips

Best practice for upgrading a model hierarchy is to check and upgrade each model starting at the leaf end and working up to the root model.

When you click **Continue Upgrade Sequence**, the Upgrade Advisor opens the leaf model as far inside the hierarchy as it can find. Subsequent steps guide you through upgrading your hierarchy from leaf to root model.

When you open the Upgrade Advisor, the checks that are selected do not require compile time information and do not trigger an Update Diagram. Checks that trigger an Update Diagram are not selected to run by default, and are marked with ^. When you use the Upgrade Advisor on a hierarchy, keep clicking **Continue Upgrade Sequence** to move through this sequence of analysis:

1. The Upgrade Advisor opens each model and library in turn, from leaf to root, and selects the non-compile checks. Run the checks, take any advised actions, then click **Continue Upgrade Sequence** to open the next model and continue.
2. When you reach the root end of the hierarchy, the Upgrade Advisor then opens each model again in the same order (but not libraries) and selects only the checks that require a model compile. Run the checks, take any advised actions, then click **Continue Upgrade Sequence** to open the next model. Continue until you reach the end of the hierarchy and this check passes.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Model Reference Conversion Advisor

Model Reference Conversion Advisor

Check Conversion Input Parameters

Use input parameters to configure the actions the advisor performs and the output it produces.

You can use the default parameters to run the advisor without changing any parameters.

Input Parameter	Description
New model name	The advisor provides a model name that is based on the Subsystem block name and is unique in the MATLAB path. The model name cannot exceed 59 characters. **Tip** If the advisor generates an error indicating that the target referenced model already exists, then use the **New model name** parameter to specify a new file name.
Conversion data file name	The advisor creates a file for storing data created during the conversion. By default, the advisor uses the model name at the beginning of the file name and appending `_conversion_data.mat`. For example, if the subsystem is in a model named `myModel`, the conversion file name is `myModel_conversion_data.mat`. You can save the conversion data in a MAT-file (default) or a MATLAB file. If you use a `.m` file extension, the advisor serializes all variables to a MATLAB file.
Fix errors automatically	By default, if an advisor check finds any errors and the advisor can fix the error, the advisor provides a **Fix** button that you can click to have the advisor fix the issue. If you enable this parameter, the advisor fixes all conversion errors that it can, without displaying the **Fix** button.
Replace content of a subsystem with a Model block	By default, the advisor updates the original model by inserting a Model block in the model. The advisor action depends on whether you use the automatic fix options. • If you use the automatic fixes, then the advisor replaces the Subsystem block with a Model block unless the automatic

Input Parameter	Description
	fixes change the input or output ports. If the ports change, then the advisor includes the contents of the subsystem in a Model block that is inserted in the Subsystem block. • If you do not use the automatic fixes, then the advisor replaces the Subsystem block with a Model block. Clear this parameter to have the advisor open a new Simulink Editor window that contains only a Model block that references the newly created referenced model. The advisor does not update the original model in the other Simulink Editor window. **Note:** If you are converting a variant subsystem, do not use this option. See "Convert Each Variant Subsystem".
Create a wrapper subsystem	Insert a wrapper subsystem to preserve the layout of a model. The subsystem wrapper contains the Model block from the conversion. If the automatic fixes change the subsystem input or output ports, the conversion process enables this option.

Input Parameter	Description
Check simulation results after conversion	Compare the results of simulating the top model for the referenced model to the results of simulating the baseline model that has the subsystem. To use this option, before performing the conversion, enable signal logging for the subsystem output signals of interest in the model. Set these advisor options: • **Model block simulation mode** — Use the same simulation mode as in the original model. • **Replace content of a subsystem with a Model block** — Enable this option. • **Stop time** — Specify when you want the simulations to end. The default is 60 seconds. • **Absolute tolerance** — Specify a value if you do not want to use the default of `'1e-06'`. • **Relative tolerance** — Specify a value if you do not want to use the default of `'1e-03'`. To see the results after the conversion is complete, click **View comparison results**. The advisor displays the results of the comparison in the Simulation Data Inspector. For more information, see "Compare Simulation Results Before and After Conversion".
Stop time	By default, the advisor uses the stop time of the top model, unless the stop time of the top model is `Inf`. If the stop time of the top model is `Inf`, the advisor uses a default stop time of `10`. You can specify a different stop time. For details, see "Specify Simulation Start and Stop Time". To use this option, select **Check simulation results after conversion**.
Absolute tolerance	The absolute signal tolerance for the simulation run comparison. The default is `1e-06`. To use this option, select **Check simulation results after conversion**.

Input Parameter	Description
Relative tolerance	The relative signal tolerance for the simulation run comparison. The default is `1e-03`. To use this option, select **Check simulation results after conversion**.
Model block simulation mode	Simulation mode for the new Model block that references the referenced model. • `Normal` (default) • `Accelerator`

After you configure the advisor, to start the conversion checks, click **Run this task**.

10

Performance Advisor Checks

Simulink Performance Advisor Checks

In this section...

Simulink Performance Advisor Check Overview

Use Performance Advisor checks to improve model simulation time.

See Also

"Improve Simulation Performance Using Performance Advisor "

Baseline

Establish a measurement to compare the performance of a simulation after Performance Advisor implements improvements.

See Also

"Create a Performance Advisor Baseline Measurement"

Checks that Require Update Diagram

These checks require that **Update Diagram** occurs in order to run.

See Also

"Improve Simulation Performance Using Performance Advisor "

Checks that Require Simulation to Run

These checks require simulation to run in order to collect sufficient performance data. Performance Advisor reports the results after simulation completes.

See Also

"Improve Simulation Performance Using Performance Advisor "

Check Simulation Modes Settings

These checks evaluate simulation modes (Normal, Accelerator, Rapid Accelerator, Rapid Accelerator with up-to-date check off) and identify the optimal mode to achieve fastest simulation.

See Also

"What Is Acceleration?"

Check Compiler Optimization Settings

Use these checks to select compiler optimization settings for improved performance.

See Also

"Compiler optimization level"

Create baseline

Select this check to create a baseline when Performance Advisor runs. You can also create a baseline manually. A baseline is the measurement of simulation performance before you run checks in Performance Advisor. The baseline includes the time to run the simulation and the simulation results (signals logged). Before you create a baseline for a model, in the **Data Import/Export** pane of the Configuration Parameters dialog box:

- Select the **States** check box.
- Set the **Format** parameter to `Structure with time`.

See Also

"Create a Performance Advisor Baseline Measurement"

Identify resource-intensive diagnostic settings

To improve simulation speed, disable diagnostics where possible. For example, some diagnostics, such as **Solver data inconsistency** or **Array bounds exceeded**, incur run-time overheads during simulations.

See Also

- "Diagnostics"
- "Improve Simulation Performance Using Performance Advisor "

Check optimization settings

To improve simulation speed, enable optimizations where possible. For example, if some optimizations, such as Block Reduction, are disabled, enable these optimizations to improve simulation speed.

You can also trade off compile-time speed for simulation speed by setting the compiler optimization level. Compiler optimizations for accelerations are disabled by default. Enabling them accelerates simulation runs but results in longer build times. The speed

and efficiency of the C compiler used for Accelerator and Rapid Accelerator modes also affects the time required in the compile step.

See Also

- "Optimization Pane: General"
- "Improve Simulation Performance Using Performance Advisor "

Identify inefficient lookup table blocks

To improve simulation speed, use properly configured lookup table blocks.

See Also

- "Lookup Tables"
- "Optimize Generated Code for Lookup Table Blocks"
- "Optimize Breakpoint Spacing in Lookup Tables"
- "Improve Simulation Performance Using Performance Advisor "

Check MATLAB System block simulation mode

In general, to improve simulation speed, choose `Code generation` for the **Simulate using** parameter of the `MATLAB System` block. Because data exchange between MATLAB and Simulink passes through several software layers, `Interpreted execution` usually slows simulations, particularly if the model needs many data exchanges.

This check identifies which MATLAB System blocks can generate code and changes the **Simulate using** parameter value to `Code generation` where possible.

While `Code generation` does not support all MATLAB functions, the subset of the MATLAB language that it does support is extensive. By using this `Code generation`, you can improve performance.

See Also

- `MATLAB System`
- "Simulation Modes"

- “Improve Simulation Performance Using Performance Advisor ”

Identify Interpreted MATLAB Function blocks

To improve simulation speed, replace `Interpreted MATLAB Function` blocks with `MATLAB Function` blocks where possible. Because data exchange between MATLAB and Simulink passes through several software layers, `Interpreted MATLAB Function` blocks usually slow simulations, particularly if the model needs many data exchanges.

Additionally, because you cannot compile an `Interpreted MATLAB Function`, an `Interpreted MATLAB Function` block impedes attempts to use an acceleration mode to speed up simulations.

While `MATLAB Function` blocks do not support all MATLAB functions, the subset of the MATLAB language that it does support is extensive. By replacing your interpreted MATLAB code with code that uses only this embeddable MATLAB subset, you can improve performance.

See Also

- `MATLAB Function`
- “Improve Simulation Performance Using Performance Advisor ”

Identify simulation target settings

To improve simulation speed, disable simulation target settings where possible. For example, in the Configuration Parameters dialog box, clear the **Simulation Target** > **Echo expression without semicolons** check box to improve simulation speed.

See Also

- “Simulation Target Pane: General”
- “Improve Simulation Performance Using Performance Advisor ”

Check model reference rebuild setting

To improve simulation speed, in the Configuration Parameters dialog box, verify that the **Model Referencing** > **Rebuild** parameter is set to `If any changes in known dependencies detected`.

See Also

- "Rebuild"
- "Improve Simulation Performance Using Performance Advisor "

Identify Scope blocks

Opened and uncommented Scope blocks can impact simulation performance. To improve simulation performance, close and comment out Scope blocks. Right-click a scope block, and then select **Comment Out**.

For opened Scopes, you can improve simulation speed by reducing updates. From the Scope **Simulation** menu, select **Reduce Updates to Improve Performance**.

See Also

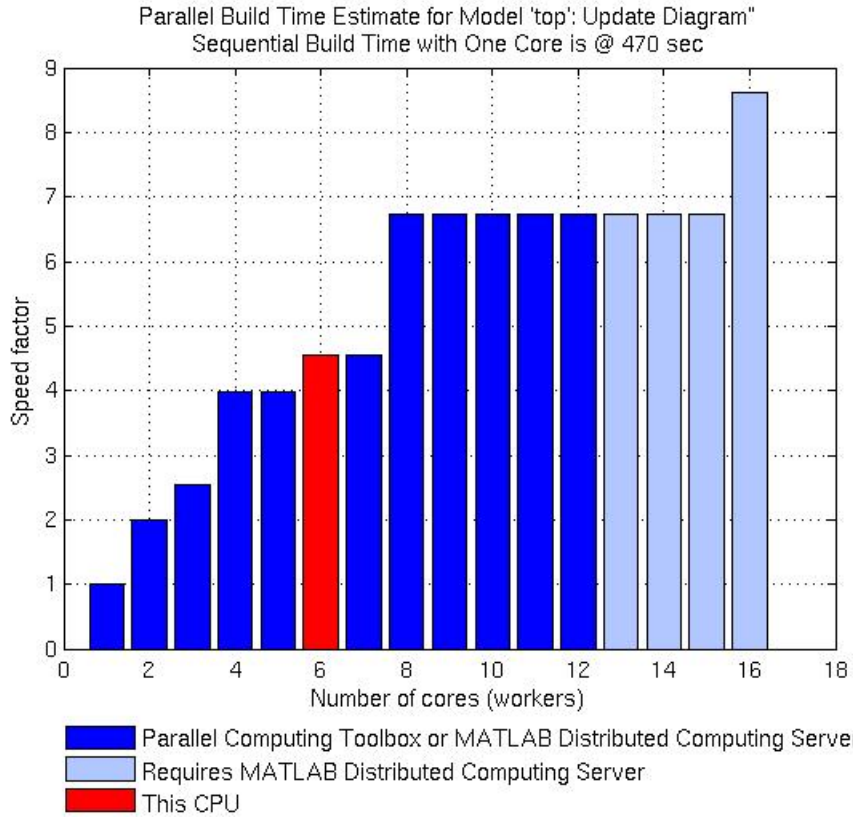
- "Improve Simulation Performance Using Performance Advisor "

Check model reference parallel build

To improve simulation, verify the number of referenced models in the model. If there are two or more referenced models, build the model in parallel if possible.

Performance Advisor analyzes the model and estimates the build time on the current computer as if it were using several cores. It also estimates the parallel build time for the model in the same way an estimation would be performed if Parallel Computing Toolbox or MATLAB Distributed Computing Server™ software were installed on the computer. Performance Advisor performs this estimate as follows:

1. Search the model for referenced models that do not refer to other referenced models.
2. Calculate the average number of blocks in each of the referenced models that do not refer to other referenced models.
3. Of the list of referenced models that do not refer to others, select a referenced model whose number of blocks is closest to the calculated average.
4. Build this model to obtain the build time.
5. Based on the number of blocks and the build time for this referenced model, estimate the build time for all other referenced models.
6. Based on these build times, estimate the parallel build time for the top model.



To calculate the overhead time introduced by the parallel build mechanism, set the Parallel Build Overhead Time Estimation Factor. Performance Advisor calculates the estimated build time with overhead as:

(1 + Parallel Build Overhead Time Estimation Factor)*(Build time on a single machine)

See Also

- "Enable parallel model reference builds"
- "Improve Simulation Performance Using Performance Advisor "

Check Delay block circular buffer setting

To improve simulation, check that each `Delay` block in the model uses the appropriate buffer type. By default, the block uses an array buffer (the **Use circular buffer for state** option is not selected). However, when the delay length is large, a circular buffer can improve execution speed by keeping the number of copy operations constant.

If the `Delay` block is currently using an array buffer, and all of the following conditions are true, Performance Advisor selects a circular buffer:

- The `Delay` block is in sample-based mode, i.e,either the **Input processing** parameter is set to `Elements as channels (sample based)`, or the input signal type is set to `Sample based`.
- The value or upper limit of the delay length is 10 or greater.
- The size of the state—equal to the delay length multiplied by the total of all output signal widths—is 1000 or greater.

See Also

- `Delay`
- "Improve Simulation Performance Using Performance Advisor "

Check 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

- "Solvers"
- "Speed Up Simulation"
- "Improve Simulation Performance Using Performance Advisor "

Select simulation mode

To achieve fastest simulation time, use this check to evaluate the following modes and identify the optimal selection:

- Normal
- Accelerator
- Rapid Accelerator
- Rapid Accelerator with up-to-date check off

In Normal mode,Simulink interprets your model during each simulation run. If you change the model frequently, this is generally the preferred mode to use because it

requires no separate compilation step. It also offers the most flexibility to make changes to your model.

In Accelerator mode,Simulink compiles a model into a binary shared library or DLL where possible, eliminating the block-to-block overhead of an interpreted simulation in Normal mode. Accelerator mode supports the debugger and profiler, but not runtime diagnostics.

In Rapid Accelerator mode, simulation speeds are fastest but this mode only works with models where C-code is available for all blocks in the model. Also, this mode does not support the debugger or profiler.

When choosing Rapid Accelerator with up-to-date check off, Performance Advisor does not perform an up-to-date check during simulation. You can run the Rapid Accelerator executable repeatedly while tuning parameters without incurring the overhead of up-to-date checks. For instance, if you have a large model or a model that makes extensive use of model reference, this method of execution can increase efficiency.

For models with 3–D signals, Normal or Accelerator modes work best.

See Also

- "How Acceleration Modes Work"
- "Choosing a Simulation Mode"
- "Comparing Performance"
- "Run Simulation Using the sim Command"

Select compiler optimizations on or off

Use this check to determine whether performing compiler optimization can help improve simulation speed. The optimization can only be performed in Accelerator or Rapid Accelerator modes.

Note: This check will be skipped if MATLAB is not configured to use an optimizing compiler.

See Also

- "How Acceleration Modes Work"

- “Choosing a Simulation Mode”
- “Comparing Performance”
- “Improve Simulation Performance Using Performance Advisor ”

Final Validation

This check validates the overall performance improvement of simulation time and accuracy in a model. If the performance is worse than the original model, Performance Advisor discards all changes to the model and loads the original model.

Global settings for validation do not apply to this check. If you have not validated the performance improvement from changes resulting from other checks, use this check to perform a final validation of all changes to a model.

See Also

- “Comparing Performance”
- “Improve Simulation Performance Using Performance Advisor ”

11

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